Gene Expression Programming in Problem Solving

Cândida Ferreira

Departamento de Ciências Agrárias
Universidade dos Açores
9701-851, Angra do Heroísmo, Portugal
candidaf@gene-expression-programming.com
http://www.gene-expression-programming.com/author.asp

In R. Roy, M. Köppen, S. Ovaska, T. Furuhashi, and F. Hoffmann, eds., *Soft Computing and Industry – Recent Applications,* pages 635-654, Springer-Verlag, 2002.

Gene expression programming is a full fledged genotype/phenotype system that evolves computer programs encoded in linear chromosomes of fixed length. The structural organization of the linear chromosomes allows the unconstrained and fruitful (in the sense that no invalid phenotypes will follow) operation of important genetic operators such as mutation, transposition, and recombination as the expression of each gene always results in valid programs. Although simple, the genotype/phenotype system of gene expression programming is the first artificial genotype/phenotype system with a complex and sounding translation mechanism. Indeed, the interplay between genotype (chromosomes) and phenotype (expression trees) is at the core of the tremendous increase in performance observed in gene expression programming. Furthermore, gene expression programming shares with genetic programming the same kind of tree representation and, therefore, with GEP it is possible, for one thing, to retrace easily the steps undertaken by genetic programming and, for another, to explore easily new frontiers opened up by the crossing of the phenotype threshold. In this tutorial, the fundamental differences between gene expression programming and its predecessors, genetic algorithms and genetic programming, are briefly summarized so that the evolutionary advantages of gene expression programming could be better understood. The work proceeds with a detailed description of the main players in this new algorithm, focusing mainly on the interactions between them and how the simple yet revolutionary structure of the chromosomes allows the efficient, unconstrained exploration of the search space.

1. Genetic algorithms at large

The aim of this introduction is to bring into focus the basic differences between gene expression programming (GEP) and its predecessors, genetic algorithms (GAs) and genetic programming (GP). According to Mitchell (1996), gene expression programming is, like GAs and GP, a genetic algorithm as it uses populations of individuals, selects them according to fitness, and introduces genetic variation using one or more genetic operators. The fundamental difference between the three algorithms resides in the nature of the individuals: in GAs the individuals are symbolic strings of fixed length (chromosomes); in GP the individuals are non-linear entities of different sizes and shapes (parse trees); and in GEP the individuals are encoded as symbolic strings of fixed length (chromosomes) which are then expressed as non-linear entities of different sizes and shapes (expression trees).

1.1. Genetic algorithms

Genetic algorithms, invented by J. Holland in the 1960s, applied biological evolution theory to computer systems (Holland 1975). Like all evolutionary computer systems, GAs are an

oversimplification of biological evolution. In this case, solutions to a problem are encoded in character strings (usually 0's and 1's), and a population of these solutions is left to evolve in order to find a solution to the problem at hand. Populations, and therefore solutions, evolve because individual solutions (chromosomes) reproduce with modification. This is obviously the prerequisite for evolution to occur. Modification in the original GA was introduced by mutation, crossover, and inversion. In addition, for evolution to occur, individuals must pass the sieve of selection. They are selected according to fitness, being the fitness rigorously determined and its value used to reproduce them proportionately. The higher the fitness, the higher the probability of leaving more offspring.

The chromosomes of GAs are simple replicators (e.g., Dawkins 1995), and therefore they survive by virtue of their properties alone. This is equivalent to say that they function simultaneously as genome and phenome. So, the chromosomes are not only keepers of the genetic information that is replicated and transmitted with modification to the next generation, but are also the object of selection. The variety of functions GAs' chromosomes are able to play is severely limited by this dual role and by their structural organization, specially the simple language of chromosomes and their fixed length. This very much resembles a simple RNA World, where the linear RNA genome is also capable of exhibiting structural diversity. In this case, the whole structure of the RNA molecule determines the functionality and, therefore, the fitness of the individual. For instance, it wouldn't be possible in such systems to use only a particular region of the genome as a solution to the problem: the whole genome is always the solution. Obviously these systems are severely constrained.

1.2. Genetic programming

Genetic programming, invented by Cramer in 1985 and further developed by Koza (1992), solved the problem of fixed length solutions by creating non-linear entities with different sizes and shapes. The alphabet used to create these entities was also more varied, creating a richer, more versatile system of representation. However, the created individuals lacked a simple, autonomous genome, functioning simultaneously both as genome and phenome. Again, in the jargon of evolutionary theory, the entities of GP are simple replicators that survive by virtue of their own properties. The non-linear entities (parse trees) of GP resemble protein molecules in their use of a richer alphabet and in their complex, hierarchical representation. Thus, GP entities are capable of exhibiting a great variety of functionalities. But these entities are very difficult to reproduce with modification because the genetic modifications are done directly on the parse tree itself. Consequently, most modifications generate structural impossibilities. As a comparison, it is worth noticing that, in nature, the expression of any protein gene results always in a valid protein structure (in nature, there is no such thing as a structurally incorrect protein).

So, in GP, the genetic operators act directly on the parse tree and, although at first sight this might appear advantageous, it greatly limits this technique (it is impossible to make an orange tree produce mangos only by grafting and pruning). Furthermore, the pallet of genetic operators available to GP is very limited, because most of them would result in invalid parse trees. Consequently, GP uses almost exclusively a special kind of recombination that operates at the level of parse trees. In this GP-specific crossover, selected branches are exchanged between two parent parse trees to create offspring. The idea behind its implementation was to exchange smaller, mathematically concise blocks in order to evolve more complex, hierarchical solutions composed of smaller building blocks.

The mutation operator in GP also differs from point mutations in nature in order to guarantee the creation of syntactically correct programs. The mutation operator selects a

node in the parse tree and replaces the branch beneath that node by a randomly generated branch. Again, the overall shape of the tree is not greatly changed by this kind of mutation.

Permutation is the third operator used in GP and, like recombination and mutation, is greatly constrained: the arguments to a chosen function are selected and exchanged. In this case the overall shape of the tree remains unchanged.

Although J. Koza described these three operators as the basic GP operators, crossover is practically the only genetic operator used in most GP implementations. Not surprisingly, in GP, huge populations of parse trees are used with the aim of creating all the necessary building blocks with the inception of the initial population in order to guarantee the discovery of a solution only by moving these initial building blocks around.

Finally, due to the dual function of the parse trees (genome and phenome), and like GAs, GP is incapable of a simple, rudimentary expression: in all cases, the entire parse tree is the solution.

1.3. Gene expression programming

Gene expression programming was invented by myself in 1999 (Ferreira 2001), and is the natural development of GAs and GP. GEP uses the same kind of diagram representation of GP, but the entities evolved by GEP (expression trees) are the expression of a genome. Therefore, with GEP, the second evolutionary threshold - the Phenotype Threshold - is crossed, providing new and efficient solutions to evolutionary computation.

The great insight of GEP consisted in the invention of chromosomes capable of representing any expression tree. For that a new language was created so that the information of GEP chromosomes could be read and expressed. Also important is that the structure of GEP chromosomes allows the easy implementation of multiple genes, each encoding a sub-program. On the other hand, the structural and functional organization of GEP genes and their interplay with expression trees, always guarantees the production of valid programs, no matter how much or how profoundly we modify the chromosomes.

In contrast to its analogous cellular gene expression, the expression of the genetic information in GEP is rather simple. The main players in GEP are only two: the chromosomes and the expression trees (ETs), being the latter the expression of the genetic information encoded in the former. As in nature, the process of information decoding is called translation. And this translation implies obviously a kind of code and a set of rules. The genetic code is very simple: a one-to-one relationship between the symbols of the chromosome and the functions or terminals they represent. The rules are also very simple: they determine the spatial organization of the functions and terminals in the ETs and the type of interaction between sub-ETs.

In GEP there are therefore two languages: the language of genes and the language of ETs, and knowing the sequence or structure of one, is knowing the other. In nature, despite being possible to infer the sequence of proteins given the sequence of genes and vice versa, we practically know nothing about the rules that determine the three-dimensional structure of proteins. But in GEP, thanks to the simple rules that determine the structure of ETs and their interactions, it is possible to infer immediately the phenotype given the sequence of a gene, and vice versa. This bilingual and unequivocal system is called *Karva* language.

The tutorial proceeds with the presentation of the structural and functional organization of GEP chromosomes; how the chromosomes are translated into expression trees; how the chromosomes function as genotype and the expression trees as phenotype; and how an individual program is created, matured, and reproduced, leaving offspring with new properties, thus, capable of adaptation.

2. The genome of GEP individuals

In GEP, the genome or chromosome consists of a linear, symbolic string of fixed length composed of one or more genes. Despite their fixed length, we will see that GEP chromosomes code for ETs with different sizes and shapes.

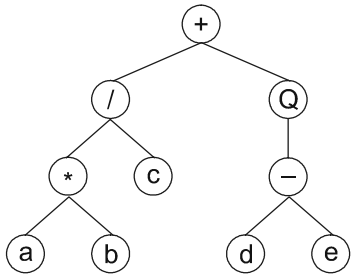
2.1. Open reading frames and genes

The structural organization of GEP genes is better understood in terms of open reading frames (ORFs). In biology, an ORF or coding sequence of a gene begins with the 'start' codon, continues with the amino acid codons, and ends at a termination codon. However, a gene is more than the respective ORF, with sequences upstream of the start codon and sequences downstream of the stop codon. Although in GEP the start site is always the first position of a gene, the termination point not always coincides with the last position of a gene. It is common for GEP genes to have non-coding regions downstream of the termination point. For now we won't consider these non-coding regions, because they don't interfere with the product of expression.

Consider, for example, the algebraic expression:

$$\frac{a \cdot b}{c} + \sqrt{d - e} \tag{2.1}$$

It can also be represented as a diagram:



where 'Q' represents the square root function.

This kind of diagram representation is in fact the phenotype of GEP chromosomes, being the genotype easily inferred from the phenotype as follows:

```
0123456789
+/Q*c-abde
```
(2.2)

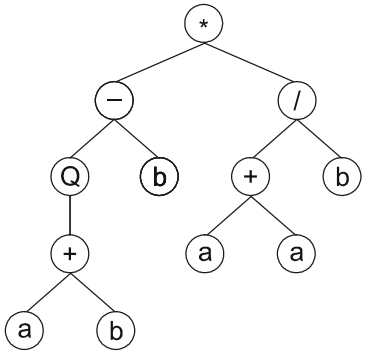
which is the straightforward reading of the ET from left to right and from top to bottom (exactly as we read a page of text). The expression (2.2) is an ORF, starting at '+' (position 0) and terminating at 'e' (position 9). I named these ORFs K-expressions (from Karva notation).

Consider another ORF, the following K-expression:

```
012345678901
*-/Qb+b+aaab
```
(2.3)

Its expression as an ET is also very simple and straightforward. To correctly express the ORF, we must follow the rules governing the spatial distribution of functions and terminals. The start position (position 0) in the ORF corresponds to the root of the ET. Then, below

each function are attached as many branches as there are arguments to that function. The assemblage is complete when a baseline composed only of terminals (the variables or constants used in a problem) is formed. So, for the K-expression (2.3) above, the following ET is formed:



Looking at the structure of GEP ORFs only, it is difficult or even impossible to see the advantages of such a representation, except perhaps for its simplicity and elegance. However, when ORFs are analyzed in the context of a gene, the advantages of this representation become obvious. As I said, GEP chromosomes have fixed length, and they are composed of one or more genes of equal length. Therefore the length of a gene is also fixed. Thus, in GEP, what varies is not the length of genes which is constant, but the length of the ORFs. Indeed, the length of an ORF may be equal or less than the length of the gene. In the first case, the termination point coincides with the end of the gene, and in the last case, the termination point is somewhere upstream of the end of the gene.

So, what is the role of these non-coding regions in GEP genes? They are in fact the essence of GEP and evolvability, for they allow the modification of the genome using any genetic operator without restrictions, producing always syntactically correct programs without the need for a complicated editing process or highly constrained ways of implementing genetic operators. Indeed, this is the paramount difference between GEP and previous GP implementations, with or without linear genomes.

Let's analyze then the structural organization of GEP genes in order to understand how they invariably code for syntactically correct programs and why they allow the unconstrained application of any genetic operator.

2.2. GEP genes

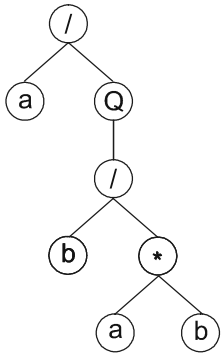
GEP genes are composed of a head and a tail. The head contains symbols that represent both functions and terminals, whereas the tail contains only terminals. For each problem, the length of the head *h* is chosen, whereas the length of the tail *t* is a function of *h* and the number of arguments of the function with more arguments *n*, and is evaluated by the equation:

$$t = h\,(n - 1) + 1 \qquad (2.4)$$

Consider a gene for which the set of functions F = {Q, *, /, -, +} and the set of terminals T = {a, b}. In this case, $n = 2$; and if we chose an $h = 15$, then $t = 16$. Thus, the length of the gene *g* is 15+16=31. One such gene is shown below (the tail is shown in bold):

```
012345678901234567890123456789
/aQ/b*ab/Qa*b*-ababaababbabbbba
```
(2.5)

It codes for the following ET with eight nodes:



In this case, the ORF ends at position 7, whereas the gene ends at position 30.

Suppose now a mutation occurred at position 2, changing the ‘Q’ into ‘+’. Then the following gene is obtained:

```
012345678901234567890123456789
/a+/b*ab/Qa*b*-ababaababbabbbba
```
(2.6)

And its expression gives an ET with 18 nodes. In this case, the termination point shifts 10 positions to the right (position 17).

Obviously the opposite might also happen, and the ORF can be shortened. For example, consider again gene (2.5) above, and suppose a mutation occurred at position 5, changing the ‘*’ into ‘b’:

```
012345678901234567890123456789
/aQ/bbab/Qa*b*-ababaababbabbbba
```
(2.7)

Its expression results in an ET with six nodes. In this case, the ORF ends at position 5, shortening the parental ET in two nodes.

So, despite its fixed length, each gene has the potential to code for ETs of different sizes and shapes, being the simplest composed of only one node (when the first element of a gene is a terminal) and the biggest composed of as many nodes as the length of the gene (when all the elements of the head are functions with maximum arity).

It is evident from the examples above, that any modification made in the genome, no matter how profound, results always in a structurally correct ET. The only thing we must be careful about, is in not disrupting the structural organization of genes, maintaining always the boundaries between head and tail and not allowing symbols representing functions on the tail. We will pursue these matters further in section 3 where the mechanisms and effects of different genetic operators are thoroughly analyzed.

2.3. Multigenic chromosomes

GEP chromosomes are usually composed of more than one gene of equal length. For each problem or run, the number of genes, as well as the length of the head, are *a priori* chosen. Each gene codes for a sub-ET and the sub-ETs interact with one another forming a more complex multi-subunit ET.

Consider, for example, the following chromosome with length 45, composed of three genes (the tails are shown in bold):

```
012345678901234012345678901234012345678901234
Q/*b+Qa babaaaba -abQ/*+bababbab**-*bb/babaaaab
```
(2.8)

It has three ORFs, and each ORF codes for a sub-ET (Figure 1). Position zero marks the start of each gene. The end of each ORF, though, is only evident upon construction of the respective sub-ET. As shown in Figure 1, the first ORF ends at position 8 (sub-ET$_1$); the second ORF ends at position 2 (sub-ET$_2$); and the last ORF ends at position 10 (sub-ET$_3$). Thus, GEP chromosomes contain several ORFs, each ORF coding for a structurally and functionally unique sub-ET. Depending on the problem at hand, these sub-ETs may be selected individually according to their respective fitness (for example, in problems with multiple outputs), or they may form a more complex, multi-subunit ET where individual sub-ETs interact with one another by a particular kind of posttranslational interaction or linking. For instance, algebraic sub-ETs can be linked by addition or multiplication whereas Boolean sub-ETs can be linked by OR, AND or if(x,y,z).

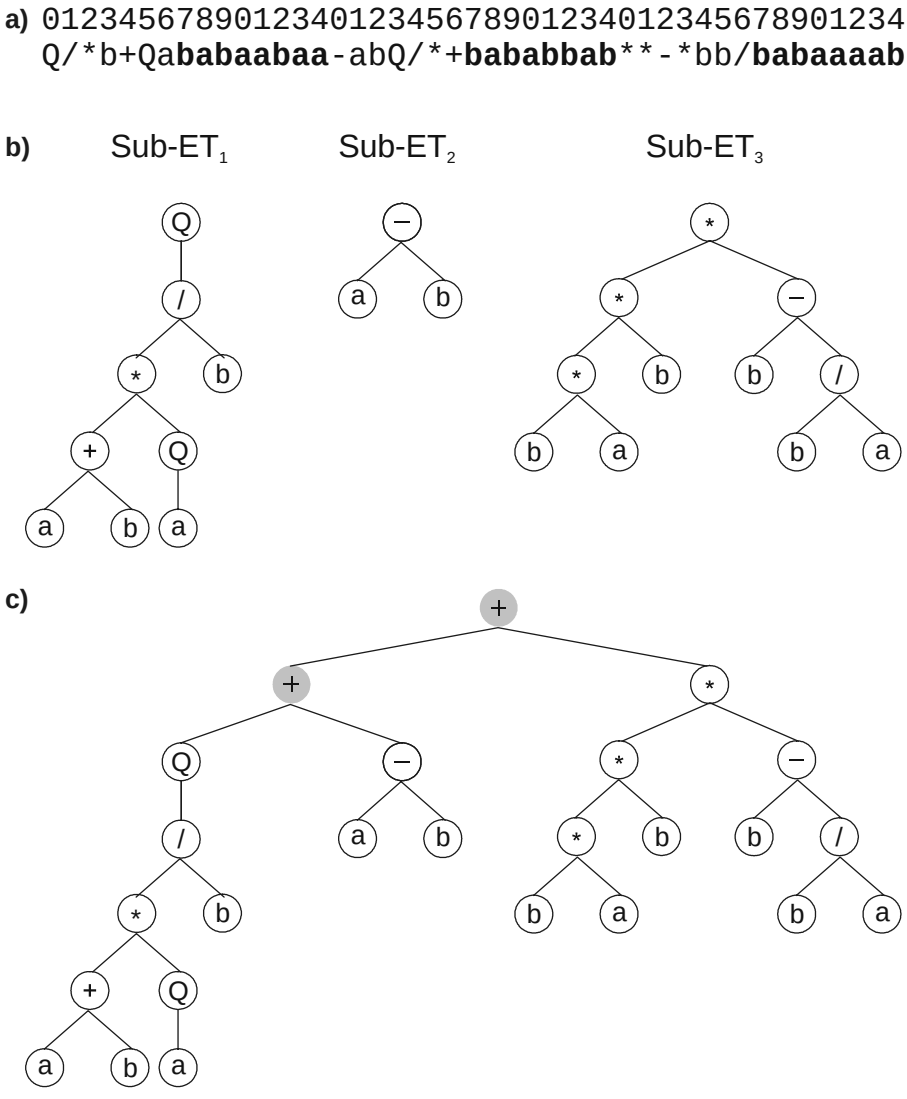


Fig. 1. Expression of GEP genes as sub-ETs. **a)** A three-genic chromosome with the tails shown in bold. Position zero marks the start of each gene. **b)** The sub-ETs codified by each gene. **c)** The result of posttranlational linking with addition. The linking functions are shown in gray.

The linking of three sub-ETs by addition is illustrated in Figure 1, c. Note that the final ET could be linearly encoded as the following K-expression:

```
012345678901234567890123456
++*Q-*-/ab*bb/*bbaba+b+Qaba                    (2.9)
```

However, to evolve solutions to complex problems, it is more effective the use of multigenic chromosomes, for they permit the modular construction of complex, hierarchical structures, where each gene codes for a small building block. These small building blocks are separated from each other, and thus can evolve independently. Furthermore, these multigenic systems are much more efficient than unigenic ones. Indeed, GEP is a highly efficient hierarchical invention system capable of discovering simple blocks and using them to form more complex structures.

So, for each problem, the type of linking function, as well as the number of genes and the length of each gene, are *a priori* chosen. While attempting to solve a problem, we can always start by using a single-gene chromosome and then proceed by increasing the length of the head. If it becomes very large, we can increase the number of genes and obviously choose a function to link the sub-ETs. We can start with addition for algebraic expressions or OR for Boolean expressions, but in some cases another linking function might be more appropriate (like multiplication or IF, for instance). The idea, of course, is to find a good solution, and GEP provides the means of finding one very efficiently.

3. Genetic operators and evolution

Genetic operators are the core of all genetic algorithms, and two of them are common to all evolutionary systems: selection and replication. Indeed, all artificial systems use a scheme to select individuals more or less according to fitness. Some schemes are totally deterministic, whereas others include a touch of unpredictability. For GEP, I chose one of the latter, namely, a fitness proportionate roulette-wheel scheme (Goldberg 1989) coupled with the cloning of the best individual (simple elitism) as it pretty accurately mimics nature and produces very good results.

According to fitness and the luck of the roulette, individuals are selected to be replicated. Although vital, replication is the most uninteresting operator. During replication, chromosomes are dully copied into the next generation. The fitter the individual the higher the probability of leaving more offspring. Thus, during replication, the genomes of the selected individuals are copied as many times as the outcome of the roulette. The roulette is spun as many times as there are individuals in the population, maintaining always the same population size.

Although the center of the storm, selection and replication, by themselves, do nothing in terms of evolution. In fact, they can only cause genetic drift, making populations less and less diverse with time until all the individuals are exactly the same. So, the touch stone of all evolutionary systems is modification, or more specifically, the genetic operators that cause variation. And different algorithms create this modification differently. For instance, GAs normally use mutation and recombination; GP uses almost exclusively GP-specific recombination; and GEP uses mutation, recombination and transposition.

With the exception of GP, which is severely constrained in terms of tools of genetic modification, in GAs and GEP, it is possible to implement easily a vast set of genetic operators capable of causing genetic diversification (from now on, unless otherwise stated, I will use the designation ‘genetic operators’ to refer to those with intrinsic transforming power,

putting selection and replication aside) because the chromosomes of both algorithms allow their easy implementation. In fact, I implemented several genetic operators in GEP in order to shed some light on the dynamics of evolutionary systems, but what is important is to provide for the necessary degree of genetic diversification to allow an efficient evolution. Mutation alone (by far the most important operator) is capable of wonders. However, the interplay of mutation and the other genetic operators not only allows an effective evolution but also allows the duplication of building blocks, their circulation in the genetic pool, the creation of repetitive sequences, etc., making things really interesting.

In the remainder of this section we will see how genetic operators work and how their implementation in GEP is a child's play due to the genotype/phenotype mapping.

3.1. Mutation

Mutations can occur anywhere in the chromosome. However, the structural organization of chromosomes must remain intact. In the heads, any symbol can change into another (function or terminal); in the tails, terminals can only change into terminals. This way, the structural organization of chromosomes is maintained, and all the new individuals produced by mutation are structurally correct programs.

Consider the following three-genic chromosome:

```
012345678900123456789001234567890
Q+bb*bbbaba-**--abbbaaQ*a*Qbbbaab
```

Suppose a mutation changed the '*' at position 4 in gene 1 to '/'; the '-' at position 0 in gene 2 to 'Q'; and the 'a' at position 2 in gene 3 to '+', obtaining:

```
012345678900123456789001234567890
Q+bb/bbbabaQ**--abbbaaQ*+*Qbbbaab
```

Note that if a function is mutated into a terminal or vice versa, or a function of one argument is mutated into a function of two arguments or vice versa, the ET is modified drastically. Note also that the mutation on gene 1 is an example of a neutral mutation, as it occurred in the non-coding region of the gene. It is worth emphasizing that the non-coding regions of GEP chromosomes are ideal places for the accumulation of neutral mutations. In summary, in GEP there are no constraints neither in the kind of mutation nor the number of mutations in a chromosome: in all cases the newly created individuals are syntactically correct programs.

3.2. Transposition and insertion sequence elements

The transposable elements of GEP are fragments of the genome that can be activated and jump to another place in the chromosome. In GEP there are three kinds of transposable elements: i) short fragments with a function or terminal in the first position that transpose to the head of genes except the root (insertion sequence elements or IS elements); ii) short fragments with a function in the first position that transpose to the root of genes (root IS elements or RIS elements); iii) and entire genes that transpose to the beginning of chromosomes.

3.2.1. Transposition of IS elements

Any sequence in the genome might become an IS element, being therefore these elements randomly selected throughout the chromosome. A copy of the transposon is made and inserted at any position in the head of a gene, except the first position. The transposition operator randomly chooses the chromosome, the start of the IS element, the target site, and the length of the transposon.

Consider the following two-genic chromosome:

```
012345678901234560123456789012345 6
-ab+Q-baabaabaabQ*+*+-/aababbaaaa
```

Suppose that the sequence ‘a+Q’ in gene 1 (positions 3-5) was randomly chosen to become an IS element and transpose between positions 2-3 in gene 2, obtaining:

```
012345678901234560123456789012345 6
-ab+Q-baabaabaabQ*+a+Q*+ababbaaaa
```

Note that, on the one hand, the sequence of the transposon becomes duplicated but, on the other, a sequence with as many symbols as the IS element was deleted at the end of the head of the target gene (in this case the sequence ‘-/a’ was deleted. Thus, despite the insertion, the structural organization of chromosomes is maintained, and therefore all the new individuals created by transposition are syntactically correct programs.

3.2.2. Root transposition

All RIS elements start with a function, and thus are chosen among the sequences of the heads. For that, a point is randomly chosen in the head and the gene is scanned downstream until a function is found. This function becomes the start position of the RIS element. If no functions are found, the operator does nothing.

This operator randomly chooses the chromosome, the gene to be modified, the start of the RIS element, and its length. Consider the two-genic chromosome below:

```
012345678901234560123456789012345 6
*-bQ/++/babbabbba//Q*baa+bbbabbbbb
```

Suppose that the sequence ‘Q/+’ in gene 1 was randomly chosen to become an RIS element. Then, a copy of the transposon is made into the root of the gene, obtaining:

```
012345678901234560123456789012345 6
Q/+*-bQ/babbabbba//Q*baa+bbbabbbbb
```

Note that during transposition, the whole head shifts to accommodate the RIS element, losing, at the same time, the last symbols of the head (as many as the transposon length). In this case, the sequence ‘++/’ was deleted, and the transposon became only partially duplicated. As with IS elements, the tail of the gene subjected to transposition and all nearby genes stay unchanged. Note, again, that the newly created programs are syntactically correct because the structural organization of the chromosome is maintained.

3.2.3. Gene transposition

In gene transposition an entire gene functions as a transposon and transposes itself to the beginning of the chromosome. In contrast to the other forms of transposition, in gene transposition, the transposon (the gene) is deleted at the place of origin.

Apparently, gene transposition is only capable of shuffling genes and, for ETs linked by commutative functions, this contributes nothing to adaptation in the short run. However, gene transposition is very important when coupled with recombination (see below), for it allows not only the duplication of genes but also a more generalized recombination of genes or smaller building blocks.

This operator randomly chooses the chromosome to undergo gene transposition, and randomly chooses one of its genes (except the first, obviously) to transpose. Consider the following chromosome composed of 3 genes:

```
012345678901201234567890120123456789012
/+Qa*bbaaabaa*a*/Qbbbbbabb/Q-aabbaaabbb
```

Suppose gene 3 was chosen to undergo gene transposition. Then the following chromosome is obtained:

```
012345678901201234567890120123456789012
/Q-aabbaaabbb/+Qa*bbaaabaa*a*/Qbbbbbabb
```

Note that for numerical applications where the function chosen to link the genes is commutative, the expression evaluated by the chromosome is not modified. But the situation differs in other applications where the linking function is not commutative, for instance, the IF function chosen to link sub-ETs in Boolean problems. Note that, in those cases, gene transposition has a very drastic effect, generating most of the times nonviable individuals.

3.3. Recombination

In GEP there are three kinds of recombination: one-point recombination, two-point recombination and gene recombination. In all types of recombination, two chromosomes are randomly chosen and paired to exchange some material between them, creating two new daughter chromosomes. Usually the daughter chromosomes are as different from each other as they are from their mothers.

3.3.1. One-point recombination

In one-point recombination the chromosomes are paired and split in the same point. The material downstream of the recombination point is afterwards exchanged between the two chromosomes.

Consider the following parent chromosomes:

```
012345678901234560123456789012345 6
+*-b-Qa*aabbbbaaa-Q-//b/*aabbabbab
++//b//-bbbbbbbbb-*-ab/b+bbbaabbaa
```

Suppose bond 6 in gene 1 (between positions 5 and 6) was randomly chosen as the crossover point. Then, the paired chromosomes are cut at this bond, and exchange between them the material downstream of the crossover point, forming the offspring below:

```
012345678901234560123456789012345 6
+*-b-Q/-bbbbbbbbb-*-ab/b+bbbaabbaa
++//b/a*aabbbbaaa-Q-//b/*aabbabbab
```

It is worth noticing that with this kind of recombination, most of the times, the offspring created exhibits different traits from those of the parents.

3.3.2. Two-point recombination

In two-point recombination the chromosomes are paired and two points are randomly chosen as crossover points. The material between the recombination points is afterwards exchanged between the two parent chromosomes, forming two new daughter chromosomes.

Consider the following parent chromosomes:

```
012345678901234560123456789012345 6
*-+Q/Q*Qaaabbbbab QQab*++-aabbabaab
Q/-b-+/abaabbbaab/*-aQa*babbabbabb
```

Suppose bond 5 in gene 1 (between positions 4 and 5) and bond 7 in gene 2 (between positions 6 and 7) were chosen as crossover points. Then, the following chromosomes are created:

```
012345678901234560123456789012345 6
*-+Q/+/abaabbbaab/*-aQa*-aabbabaab
Q/-b-Q*Qaaabbbbab QQab*++babbabbabb
```

It's worth noticing that the non-coding regions of GEP chromosomes are ideal regions where chromosomes can be split to cross over without interfering with the ORFs and, in fact, during search, these regions are most favored by crossover.

3.3.3. Gene recombination

In the third kind of GEP recombination, gene recombination, entire genes are exchanged between two parent chromosomes, forming two daughter chromosomes containing genes form both mothers. The exchanged genes are randomly chosen and occupy the same position in the parent chromosomes. Consider the following parent chromosomes:

```
012345678901201234567890120123456789012
/+/ab-aabbbbb-aa**+aaabaaa-+--babbbbaab
+baQaaaabaaba*-+a-aabbabbb/ab/+bbbabaaa
```

Suppose gene 2 was chosen to be exchanged. In this case the following offspring is formed:

```
012345678901201234567890120123456789012
/+/ab-aabbbbb*-+a-aabbabbb-+--babbbbaab
+baQaaaabaaba-aa**+aaabaaa/ab/+bbbabaaa
```

Note that, with this kind of recombination, similar genes can be exchanged but, most of the times, the exchanged genes are very different and new material is introduced in the population.

It is worth noticing that this operator is unable to create new genes: the individuals created are different arrangements of existing genes. Understandingly, when gene recombi-

nation is used as the unique source of genetic variation, more complex problems can only be solved using very large initial populations in order to provide for the necessary diversity of genes. However, the creative power of GEP is based not only in the shuffling of genes or building blocks, but also in the constant creation of new genetic material.

4. Solving a simple problem with GEP

The aim of this section is to study a successful run in its entirety in order to understand how populations of GEP individuals evolve towards a perfect or good solution.

In symbolic regression or function finding the goal is to find an expression that satisfactorily explains the dependent variable. The input into the system is a set of fitness cases in the form ($a_{(i,0)}$, $a_{(i,1)}$, ..., $a_{(i,n-1)}$, y_i) where $a_{(i,0)}$ - $a_{(i,n-1)}$ are the independent variables and y_i is the dependent variable. The set of fitness cases consists of the adaptation environment where solutions adapt, discovering, in the process, solutions to problems.

In the example of this section, a simple test function was chosen, being therefore the fitness cases computer generated. Thus, in this case, we know exactly which function we are aiming at (remember, however, that in real-world problems the function is obviously unknown). So, suppose we are given a sampling of the numerical values from the curve

$$y = 3a^2 + 2a + 1 \tag{4.1}$$

over 10 randomly chosen points in the real interval [-10, +10] and we wanted to find a function fitting those values within a certain error. In this case, we are given a sample of data in the form of 10 pairs (a_i, y_i), where a_i is the value of the independent variable in the given interval and y_i is the respective value of the dependent variable (a_i values: -4.2605, -2.0437, -9.8317, -8.6491, 0.7328, -3.6101, 2.7429, -1.8999, -4.8852, 7.3998; the corresponding y_i values can be easily evaluated). These 10 pairs are the fitness cases (the input) that will be used as the adaptation environment. The fitness of a particular program will depend on how well it performs in this environment.

There are five major steps in preparing to use gene expression programming, and the first is to choose the fitness function. For this problem we could measure the fitness f_i of an individual program i by the following expression:

$$f_i = \sum_{j=1}^{C_t} \left(M - \left| C_{(i,j)} - T_j \right| \right) \tag{4.2}$$

where M is the range of selection, $C_{(i,j)}$ the value returned by the individual chromosome i for fitness case j (out of C_t fitness cases) and T_j is the target value for fitness case j. If, for all j, $|C_{(i,j)} - T_j|$ (the precision) less or equal to 0.01, then the precision is equal to zero, and $f_i = f_{max} = C_t \cdot M$. For this problem, we will use an M = 100 and, therefore, f_{max} = 1000. The advantage of this kind of fitness function is that the system can find the optimal solution for itself.

The second major step consists in choosing the set of terminals T and the set of functions F to create the chromosomes. In this problem, the terminal set consists obviously of the independent variable, i.e., T = {a}. The choice of the appropriate function set is not so obvious, but a good guess can always be done in order to include all the necessary functions. In this case, to make things simple, we will use the four basic arithmetic operators. Thus, F = {+, -, *, /}.

The third major step is to choose the chromosomal architecture, i.e., the length of the head and the number of genes. In this problem we will use an h = 6 and three genes per chromosome.

The fourth major step in preparing to use gene expression programming is to choose the linking function. In this case we will link the sub-ETs by addition.

And finally, the fifth major step is to choose the set of genetic operators that cause variation and their rates. In this case we will use a combination of all genetic operators (mutation at p_m = 0.051; IS and RIS transposition at rates of 0.1 and three transposons of length 1, 2, and 3; one-point and two-point recombination at rates of 0.3; gene transposition and gene recombination both at rates of 0.1).

To solve this problem, I chose an evolutionary time of 50 generations and a small population of 20 individuals in order to simplify the analysis of the evolutionary process and not fill this text with pages of encoded individuals. However, one of the advantages of GEP is that it is capable of solving relatively complex problems using small population sizes and, thanks to the compact Karva notation, it is possible to fully analyze the evolutionary history of a run.

To show evolution at work, I chose a successful run in which a perfect solution was found in generation 3. The initial population of this run, together with the fitness of each individual, is shown below:

```
Generation N: 0
012345678901201234567890120123456789012
+**/*/aaaaaaa/+a/a*aaaaaaa/a-*a+aaaaaaa-[ 0] = 577.3946
--aa++aaaaaaa+-/a*/aaaaaaa/--a-aaaaaaaa-[ 1] = 0
/***/+aaaaaaa*+/+-aaaaaaaa++aa/aaaaaaaa-[ 2] = 463.6533
-/+/++aaaaaaa+-//+/aaaaaaa+-/a/*aaaaaaa-[ 3] = 546.4241
++a/*aaaaaaaa+-+a*-aaaaaaa-a/-*aaaaaaaa-[ 4] = 460.8625
*+*a-*aaaaaaa*a/aa/aaaaaaa//+*a/aaaaaaa-[ 5] = 353.2168
*/**+aaaaaaaa+a/**+aaaaaaa----+/aaaaaaa-[ 6] = 492.6827
*aa-+-aaaaaaa+a/-+/aaaaaaa***/-*aaaaaaa-[ 7] = 560.9289
+/-*//aaaaaaa*+*//+aaaaaaa-/**+*aaaaaaa-[ 8] = 363.4358
--a+*/aaaaaaa+a++--aaaaaaa+a+aa+aaaaaaa-[ 9] = 386.7576
+-*-**aaaaaaa*/-+**aaaaaaa*+--++aaaaaaa-[10] = 380.6484
/a-**/aaaaaaa/-a/a/aaaaaaa+/a/-*aaaaaaa-[11] = 0
+--+//aaaaaaa+*+/*-aaaaaaa/*-a-+aaaaaaa-[12] = 551.2066
-a/+a/aaaaaaa*/--/aaaaaaaa*-+/a+aaaaaaa-[13] = 308.1296
/+/-+-aaaaaaa+-a/aaaaaaaaa**+-*-aaaaaaa-[14] = 0
//-*+/aaaaaaa//*a+aaaaaaaa/a++a*aaaaaaa-[15] = 489.5392
*a-a*-aaaaaaa+*+-a/aaaaaaa*/*aa*aaaaaaa-[16] = 399.2122
-a++*/aaaaaaa+/aa-*aaaaaaa---/**aaaaaaa-[17] = 317.6631
--a/*aaaaaaaa++*+-aaaaaaaa+-/*+-aaaaaaa-[18] = 597.8777
*+++-/aaaaaaa/--///aaaaaaa+-+aaaaaaaaaa-[19] = 661.5933
```

Note that three of the 20 individuals are nonviable and thus have fitness 0. The best of generation, chromosome 19, has fitness 661.5933. Note that the second gene of this individual returns 0 and, therefore, might be considered a pseudogene. The descendants of the individuals of the initial population are shown below:

```
Generation N: 1
012345678901201234567890120123456789012
*+++-/aaaaaaa/--///aaaaaaa+-+aaaaaaaaaa-[ 0] = 661.5933
-a++*/aaaaaaa+//a--aaaaaaa---/**aaaaaaa-[ 1] = 0
+-*-**aaaaaaa*/-+**aaaaaaa*+--++aaaaaaa-[ 2] = 380.6484
+-*-**aaaaaaa*/-+**aaaaaaa*/*a**aaaaaaa-[ 3] = 356.9471
+-+aaaaaaaaaa*+++-/aaaaaaa/--///aaaaaaa-[ 4] = 661.5933
```

```
*aa-+-aaaaaaa+a/++/aaaaaaa***+-*aaaaaaa-[ 5] = 567.9289
*a-a*-aaaaaaa+/*-a/aaaaaaa*+-*++aaaaaaa-[ 6] = 449.802
*aa-+-aaaaaaa+a/-+/aaaaaaa*+--++aaaaaaa-[ 7] = 961.8512
/***/+aaaaaaa*+/+-aaaaaaaa-a/-*aaaaaaaa-[ 8] = 470.5862
+--+//aaaaaaa+*+/*-aaaaaaa/*-a-+aaaaaaa-[ 9] = 551.2066
*+++-/aaaaaaa-//--/aaaaaaa+-+aaaaaaaaaa-[10] = 0
--+a*-aaaaaaa++a/*aaaaaaaa-a/-*aaaaaaaa-[11] = 487.3099
-a++*/aaaaaaa+/aa-*aaaaaaa---/**aaaaaaa-[12] = 317.6631
++a/*aaaaaaaa+-+a*-aaaaaaa++aa/aaaaaaaa-[13] = 451.464
+--+/-aaaaaaa+a/**+aaaaaaa----+/aaaaaaa-[14] = 493.5336
*/-a++aaaaaaa+/aa-*aaaaaaa---/**aaaaaaa-[15] = 356.4241
+/-*//aaaaaaa*+a//+aaaaaaa-/+*+*aaaaaaa-[16] = 493.9218
*/**+aaaaaaaa+*+/*aaaaaaaa***/-*aaaaaaa-[17] = 448.4805
+-*-**aaaaaaa*/-+**aaaaaaa*+--++aaaaaaa-[18] = 380.6484
++a/*aaaaaaaa+-+a*+aaaaaaa--/-*aaaaaaaa-[19] = 380.8585
```

Note that chromosome 0 is the clone of the best individual of the previous generation. In this generation, a new individual was created, chromosome 7, considerably better than the best individual of the initial population. The descendants of the individuals of this generation are shown below:

```
Generation N: 2
012345678901201234567890120123456789012
*aa-+-aaaaaaa+a/-+/aaaaaaa*+--++aaaaaaa-[ 0] = 961.8512
*/**+aaaaaaaa*/-+**aaaaaaa***/-*aaaaaaa-[ 1] = 446.2061
+-*-**aaaaaaa*+a//-aaaaaaa-/+*+*aaaaaaa-[ 2] = 323.1036
+--+//aaaaaaa+*+/*-aaaaaaa/*-*-+aaaaaaa-[ 3] = 551.2066
*aa-+-aaaaaaa+a/++/aaaaaaa***+-*aaaaaaa-[ 4] = 567.9289
++a/*aaaaaaaa*/-+-*aaaaaaa*+--++aaaaaaa-[ 5] = 0
+-*-**aaaaaaa+*+/*aaaaaaaa*/*a**aaaaaaa-[ 6] = 386.6484
++a/*aaaaaaaa+-+/*-aaaaaaa+aa++aaaaaaaa-[ 7] = 466.1533
+-*-a*aaaaaaa*/-+**aaaaaaa*a*a**aaaaaaa-[ 8] = 194.0452
/***/+aaaaaaa*+/+-aaaaaaaa-a--*aaaaaaaa-[ 9] = 541.4829
+-*-+*aaaaaaa+-+a*-aaaaaaa***/-*aaaaaaa-[10] = 346.2235
--*+*-aaaaaaa*aa-+-aaaaaaaaa/-+/aaaaaaa-[11] = 467.0862
*/-+**aaaaaaa+-*-*+aaaaaaa*/*a**aaaaaaa-[12] = 672.877
*aa+*/aaaaaaa+a/-+/aaaaaaa*+--++aaaaaaa-[13] = 961.8512
*+++/+aaaaaaa*++/+-aaaaaaa-a/-*aaaaaaaa-[14] = 395.858
/***-/aaaaaaa/--///aaaaaaa+-+a-aaaaaaaa-[15] = 467.0862
*aa-+-aaaaaaa+a/++/aaaaaaa***+-*aaaaaaa-[16] = 567.9289
+-+aaaaaaaaaa*+++-/aaaaaaa/--///aaaaaaa-[17] = 661.5933
+/-*//aaaaaaa*/a+**aaaaaaa*+--++aaaaaaa-[18] = 903.8886
*/**+aaaaaaaa+*+/*aaaaaaaa+/aa/aaaaaaaa-[19] = 423.885
```

Note that none of the descendants surpassed the best individual of the previous generation. And finally, in the next generation, an individual with maximum fitness was created:

```
Generation N: 3
012345678901201234567890120123456789012
*aa+*/aaaaaaa+a/-+/aaaaaaa*+--++aaaaaaa-[ 0] = 961.8512
*aa-+-aaaaaaa+a/-+/aaaaaaa/--///aaaaaaa-[ 1] = 560.9289
*aa-+-aaaaaaa-++/+-aaaaaaa-a/-*aaaaaaaa-[ 2] = 558.2066
*+++/+aaaaaaa*+a/-+aaaaaaa++--++aaaaaaa-[ 3] = 569.0469
```

```
/+++/+aaaaaaa*++/+-aaaaaaa-a/-*aaaaaaaa-[ 4] = 699.5153
+-+aa/aaaaaaa++++-/aaaaaaa***+-*aaaaaaa-[ 5] = 466.1533
*aa-+-aaaaaaaaa--**aaaaaaa*+--++aaaaaaa-[ 6] = 957.9443
--++*-aaaaaaa*a+/*-aaaaaaa+aa++aaaaaaaa-[ 7] = 337.7807
*aaa*/aaaaaaa+a+-+/aaaaaaa*+-/++aaaaaaa-[ 8] = 953.9443
/***/-aaaaaaa*+/+-aaaaaaaa-a--*aaaaaaaa-[ 9] = 0
*aa-+-aaaaaaa+a/-+/aaaaaaa*/--++aaaaaaa-[10] = 560.9289
*aa-+-aaaaaaa+a/++/aaaaaaa/--///aaaaaaa-[11] = 567.9289
+-+a-aaaaaaaa/***-/aaaaaaa*+--++aaaaaaa-[12] = 676.0663
+/**//aaaaaaa*/a+**aaaaaaa*+--++aaaaaaa-[13] = 1000
*/-+**aaaaaaa+-*-*+aaaaaaa*/*a**aaaaaaa-[14] = 672.877
/***/+aaaaaaa/+*+/+aaaaaaa-a*/--aaaaaaa-[15] = 498.3734
+/-*//aaaaaaa*/a+-*aaaaaaa*+--++aaaaaaa-[16] = 0
--*+--aaaaaaa*/a-+-aaaaaaa/a/-+/aaaaaaa-[17] = 506.1233
++a/*aaaaaaaa+-a-+-aaaaaaa-a*-+/aaaaaaa-[18] = 815.7772
*+a//-aaaaaaa+a/-+/aaaaaaa-/+*+*aaaaaaa-[19] = 412.5237
```

Note that this chromosome is a descendant, via mutation, of chromosome 18 of the previous generation: their chromosomes differ only in one position (the '-' at position 2 of gene 1 was replaced by '*'). The expression of this chromosome shows that it codes for a perfect solution, the test function (4.1).

5. Summary

Gene expression programming is the most recent development on artificial evolutionary systems and one that brings about a considerable increase in performance due to the crossing of the phenotype threshold. For the first time in artificial evolution, with GEP, the phenotype threshold is fully crossed, allowing the unconstrained exploration of the search space. In GEP, the implementation of high-performing search operators such as point mutation, transposition and recombination, is a child's play as any modification made in the genome always results in valid phenotypes or programs. The structural and functional organization of GEP chromosomes and the new language (Karva language) especially developed to read and express the information encoded in the chromosomes, were thoroughly presented, allowing the easy understanding and implementation of the algorithm.

Furthermore, the workings of the algorithm were analyzed step-by-step with a simple problem of symbolic regression, where entire populations were thoroughly analyzed so that the simple yet wondrous ways of evolution could be completely understood.

References

1. Cramer, N. L., A Representation for the Adaptive Generation of Simple Sequential Programs. In J. J. Grefenstette, ed., *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, Erlbaum, 1985.

2. Dawkins, R., *River out of Eden*, Weidenfeld & Nicolson, 1995.

3. Ferreira, C., 2001. Gene Expression Programming: A New Adaptive Algorithm for Solving Problems. *Complex Systems*, forthcoming.

4. Goldberg, D. E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

5. Holland, J. H., *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, University of Michigan Press, 1975 (second edition: MIT Press, 1992).

6. Koza, J. R., *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Cambridge, MA: MIT Press, 1992.

7. Mitchell, M., *An Introduction to Genetic Algorithms*, MIT Press, 1996.