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

WSC6 tutorial, 2001

In this work, the recently invented learning algorithm, gene expression programming, will be introduced focusing mainly on problem solving. Besides a simple introductory example, I chose two relatively complex test problems of symbolic regression. One of these problems was chosen in an attempt to shed some light on the question of constant creation in models discovered with learning algorithms and to provide a palpable measure of the accuracy of the evolved models and the efficiency of the algorithms. The chosen problems also show how gene expression programming is capable of modeling complex realities with great accuracy, allowing, at the same time, the extraction of knowledge from the evolved models.

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. Consequentely, 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: two structurally equivalent nodes (two terminals or two functions with the same number of arguments) are chosen and their positions are 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 produced by GEP (expression trees) are the expression of a genome. Therefore, with GEP, the second evolutionary threshold - the Phenotype Threshold - was crossed, providing new and efficient solutions to evolutionary computation.

So, the great insight of GEP consisted in the invention of chromosomes capable of representing any expression tree. For that I created a new language (Karva) to read and express the information of GEP chromosomes. Furthermore, the structure of chromosomes was designed to allow the creation of multiple genes, each encoding a sub-expression tree. The genes are structurally organized in a head and a tail, and it is this structural and functional organization of GEP genes that always guarantees the production of valid programs, no matter how much or how profoundly we modify the chromosomes.

In the next section I describe 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. Gene expression programming: an introduction

In contrast to its analogous cellular gene expression, 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 chromosomes. 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 the 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.

2.1. The genome

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.

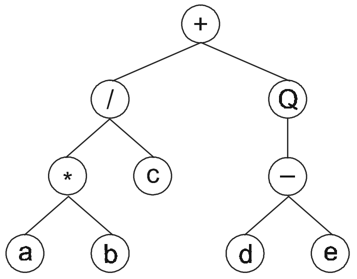
2.1.1. Open reading frames and genes

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 the start codon and sequences downstream 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 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 representations is in fact the phenotype of GEP chromosomes, being the genotype easily inferred from the phenotype as follows:

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

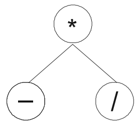
which is the straightforward reading of the ET from left to right and from top to bottom (exactly as we read a page of text). The expression 2.2 is an ORF, strarting at ‘+’ (position 0) and terminating at ‘e’ (position 9). I named these ORFs K-expressions (from Karva notation).

Consider another ORF, the following K-expression:

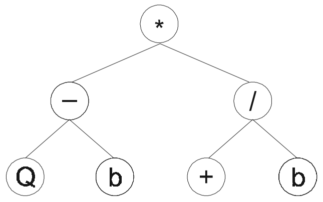
```
012345678901
*-/Qb+b+aaab                                 (2.3)
```

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. First, the start of a gene corresponds to the root of the ET, forming this node the first line. Second, depending on the number of arguments to each element (functions may have a different number of arguments, whereas terminals have an arity of zero), in the next line are placed as many nodes as there are arguments to the functions in the previous line. Third, from left to right, the nodes are filled, in the same order, with the elements of the gene. Fourth, the process is repeated until a line containing only terminals is formed. So, for the K-expression 2.3 above, the root of the ET is the symbol at position 0, obtaining:

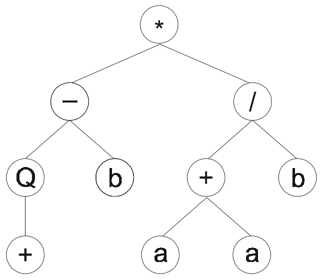
The multiplication function has two arguments, so the next line will have two nodes, in this case, the symbols at position 1 and 2:



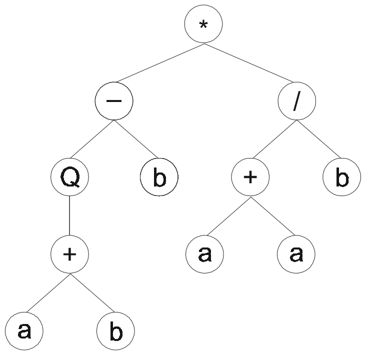
The subtraction and division are functions of two arguments, and therefore in the next line are placed four more nodes. In this case, the symbols at positions 3, 4, 5, and 6:



Now we have two different functions in the third line: one is a function of one argument (Q), and another a function of two arguments (+). Therefore three more nodes are required in the next line. In this case, they are filled with the elements at positions 7, 8, and 9:



In this new line, although there are three nodes, only one of them is a function (+). Again, the required nodes are placed below that function and filled with the next elements in the ORF (positions 10 and 11), obtaining:



In this case, with this step the ET was completely formed as the last line contains only nodes with terminals. We will see that, thanks to the structural organization of GEP genes, the last line of all ETs contains exclusively terminals. This is equivalent to say that all GEP ETs are syntactically correct.

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 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 an unconstrained application of any genetic operator.

2.1.2. GEP genes

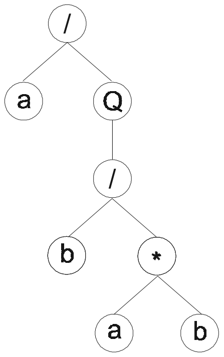
GEP genes are composed of a head and a tail. The head contains symbols that represent both functions and terminals, whereas the tail contains only terminals. For each problem, the length of the head h is chosen, whereas the length of the tail t is a function of h and the number of arguments of the function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

```
0123456789012345678901234567890
/aQ/b*ab/Qa*b*-ababaababbabbbba
```
(2.5)

It codes for the following ET:

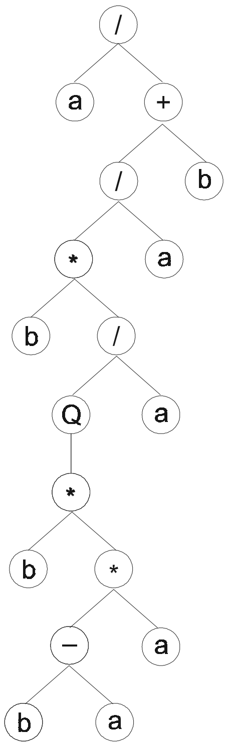


In this case, the ORF ends at position 7, whereas the gene ends at position 30.

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

```
0123456789012345678901234567890
/a+/b*ab/Qa*b*-ababaababbabbbba
```
(2.6)

And its expression gives:

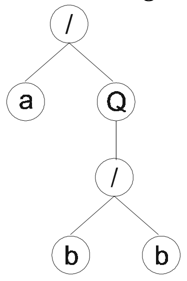


In this case, the termination point shifts 10 positions to the right (position 17).

Obviously the opposite might also happen, and the ORF is shortened. For example, consider again gene 2.5 above, and suppose a mutation occurred at position 5, changing the '*' into 'b':

```
0123456789012345678901234567890
/aQ/bbab/Qa*b*-ababaababbabbbba    (2.7)
```

Its expression results in the following ET:



In this case, the ORF ends at position 5, shortening the parental ET in 2 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 the maximum number of arguments).

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 2.3 where the mechanisms and effects of different genetic operators are thoroughly analyzed.

2.1.3. 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 multisubunit ET. The details of such interactions will be fully explained in section 2.2.

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

```
012345678901234
Q/*b+Qababaabaa
-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 and be selected according to the fitness of the whole, multi-subunit ET. The patterns of expression and the details of selection will be presented below. However, keep in mind that each sub-ET is both a separate entity and a part of a more complex, hierarchical structure, and, as in all complex systems, the whole is more than the sum of its parts.

a)
```
012345678901234012345678901234012345678901234
Q/*b+Qababaabaa-abQ/*+bababbab**-*bb/babaaaab
```

b)

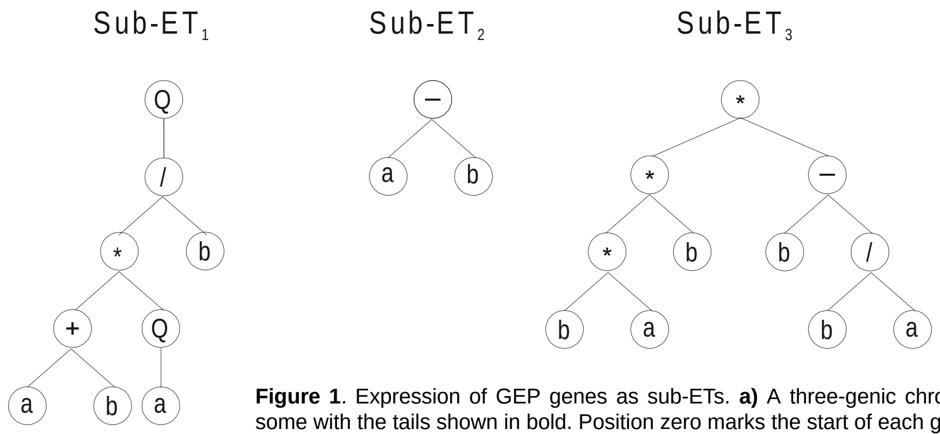


Figure 1. Expression of GEP genes as sub-ETs. **a)** A three-genic chromosome with the tails shown in bold. Position zero marks the start of each gene. **b)** The sub-ETs codified by each gene.

2.2. Posttranslational interactions and linking functions

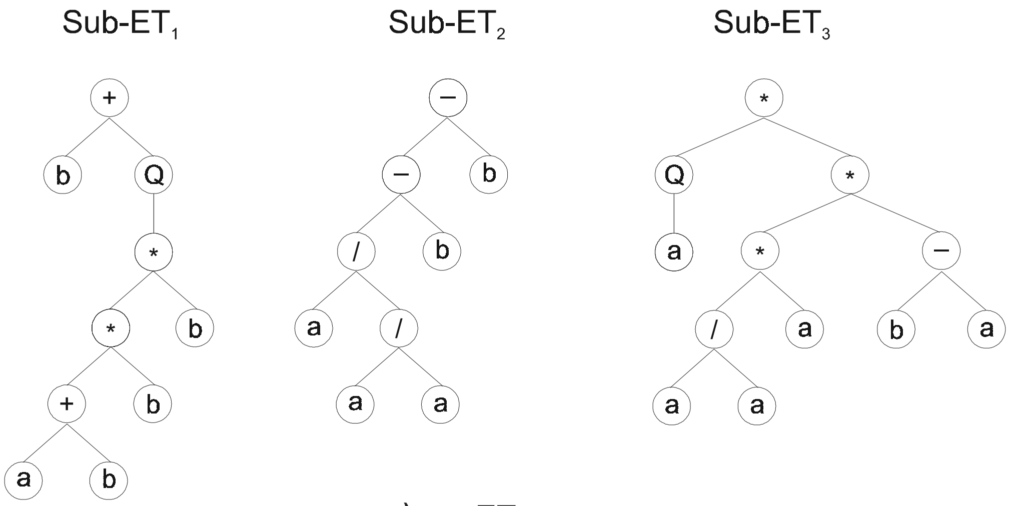
In GEP, from the simplest individual to the most complex, the expression of the genetic information starts with translation, the transfer of information from a gene into an ET. We have already seen that translation results in the formation of sub-ETs with different sizes and shapes but, in most cases, the complete expression of the genetic information requires the interaction of these sub-ETs with one another. One of the most simple interactions is the linking of sub-ETs by a particular function. This process is similar to the assemblage of different protein subunits in a multi-subunit protein.

When the sub-ETs are algebraic or Boolean expressions, any algebraic or Boolean function with more than one argument can be used to link the sub-ETs in a final, multi-subunit ET. The functions most chosen are addition or multiplication for algebraic sub-ETs, and OR or IF for Boolean sub-ETs.

Figure 2 illustrates the linking of three sub-ETs by addition. Note that the final ET could be linearly encoded as the following K-expression:

a)
```
012345678901234012345678901234012345678901234
+bQ**b+bababbbb--b/ba/aaababab*Q*a*-/abaaaaab
```

b)



c) ET

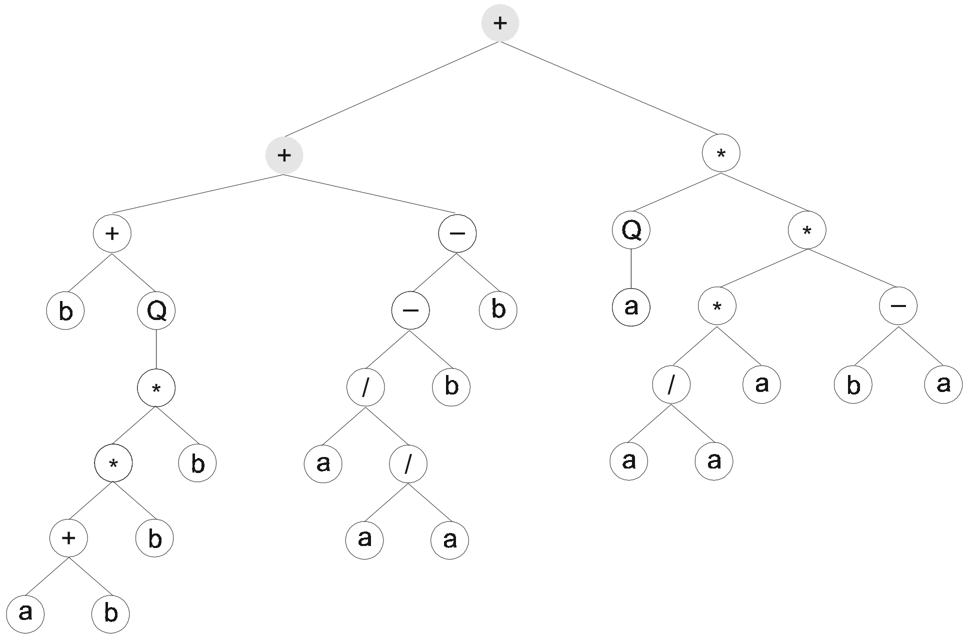


Figure 2. Expression of algebraic multigenic chromosomes as multi-subunit expression trees. **a**) A three-genic chromosome with the tails shown in bold. **b**) The sub-ETs codified by each gene. **c**) The result of posttranslational linking with addition. The linking functions are shown in gray.

```
012345678901234567890123456789012
++*+-Q*bQ-ba*-*/b/aba*ba/aa+baaab   (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 (Ferreira 2001). 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 effectively a hierarchical invention system capable of discovering simple blocks and using them to form more complex structures.

Figure 3 shows another example of posttranslational interaction, where three Boolean sub-ETs are linked by the function IF(x, y, z) (if $x = 1$, then return y; otherwise return z). Again, the multi-subunit ET could be linearized into the following K-expression:

```
012345678901234567890123
IOIAcIANAcbbAcIbbaaaaaab   (2.10)
```

where N, A, O, and I represent respectively the Boolean functions NOT, AND, OR and IF, taking ‘N’ one argument, ‘A’ and ‘O’ two, and ‘I’ three arguments.

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 for each problem. 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.

a)
012345678901234501234567890123450123456789012345
OcIbA**caabcbccaaa**IANAI**bbaaaabaaab**AcbcI**caaaacaccaa**

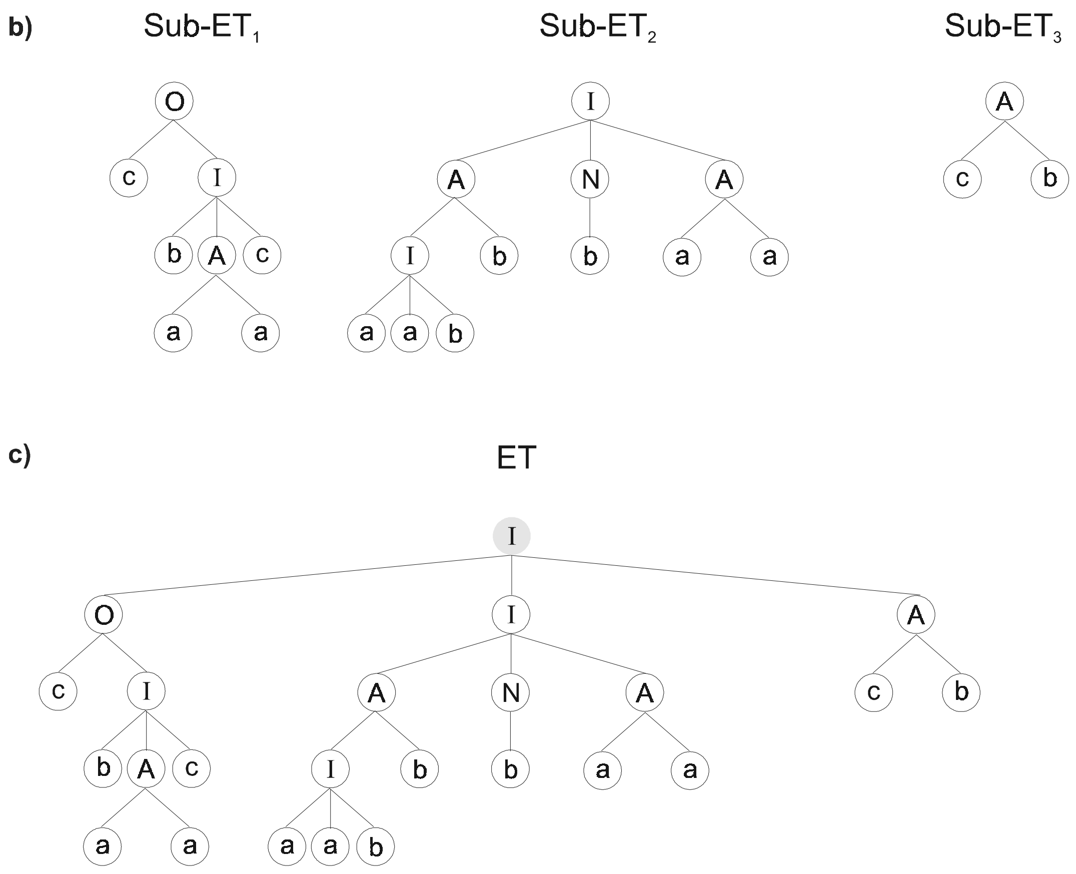


Figure 3. Expression of Boolean multigenic chromosomes as multi-subunit expression trees. **a)** A three-genic chromosome with the tails shown in bold (‘N’ is a function of one argument and represents NOT; ‘A’ and ‘O’ are functions of two arguments and represent respectively AND and OR; ‘I’ is a function of three arguments and represents IF; the remaining symbols are terminals). **b)** The sub-ETs codified by each gene. **c)** The result of posttranslational linking with IF. The linking function is shown in gray.

2.3. 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. Although the center of the storm, these operators, 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 (see Figures 4 and 5 below). 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 (Ferreira 2001), but what is important is to provide for the necessary degree of genetic diversification to allow 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 (including selection and replication) work and how they can be easily implemented in GEP.

2.3.1. Selection and replication

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.

Figure 4 shows how selected individuals are replicated (the other operators and elitism were switched off in order to better understand replication and roulette-wheel selection). For instance, chromosome 3, the best individual of generation 0, left only one daughter (chromosome 4 of generation 1); chromosome 1, the second best of generation 0, left two descendants (chromosomes 1 and 9 of generation 1); chromosome 0, a medium individual, died without leaving offspring; and, although one of the most unfit of generation 0, chromosome 6, didn't reproduce, a mediocre individual, chromosome 9, left one of the biggest progeny (chromosomes 5 and 6 of generation 1). The outcome of such an 'evolutionary' process is shown in Figure 5, where we can see that by generation 8 all the individuals are descendants of only one individual: in this case, chromosome 1 of the initial popula-

```
Generation N: 0
01234567890120123456789012
*+-/a*aaaaaaa//+*aaaaaaaaa-[0] = 10.64033
/-/a//aaaaaaa+*+a/+aaaaaaa-[1] = 16.2117
*+a-+aaaaaaaa---///aaaaaaa-[2] = 13.81953
+a*/-aaaaaaaa**+a*aaaaaaaa-[3] = 18.32701
*-+a/-aaaaaaa/aa+a/aaaaaaa-[4] = 11.13926
+*//a/aaaaaaa---aa-aaaaaaa-[5] = 13.88255
*-*-*aaaaaaaa/-a///aaaaaaa-[6] = 7.777691
/++a-*aaaaaaa/+a*+-aaaaaaa-[7] = 13.14786
//+*aaaaaaaaa*+-/--aaaaaaa-[8] = 7.713599
-**+-/aaaaaaa*//aa/aaaaaaa-[9] = 8.73985

Generation N: 1
01234567890120123456789012
*+a-+aaaaaaaa---///aaaaaaa-[0] = 13.81953
/-/a//aaaaaaa+*+a/+aaaaaaa-[1] = 16.2117
*-+a/-aaaaaaa/aa+a/aaaaaaa-[2] = 11.13926
+*//a/aaaaaaa---aa-aaaaaaa-[3] = 13.88255
+a*/-aaaaaaaa**+a*aaaaaaaa-[4] = 18.32701
-**+-/aaaaaaa*//aa/aaaaaaa-[5] = 8.73985
-**+-/aaaaaaa*//aa/aaaaaaa-[6] = 8.73985
//+*aaaaaaaaa*+-/--aaaaaaa-[7] = 7.713599
/++a-*aaaaaaa/+a*+-aaaaaaa-[8] = 13.14786
/-/a//aaaaaaa+*+a/+aaaaaaa-[9] = 16.2117
```

Figure 4. An initial population (generation 0) and their immediate descendants (generation 1). The value after each chromosome indicates the fitness.

```
Generation N: 8
01234567890120123456789012
/-/a//aaaaaaa+*+a/+aaaaaaa-[0] = 16.2117
/-/a//aaaaaaa+*+a/+aaaaaaa-[1] = 16.2117
/-/a//aaaaaaa+*+a/+aaaaaaa-[2] = 16.2117
/-/a//aaaaaaa+*+a/+aaaaaaa-[3] = 16.2117
/-/a//aaaaaaa+*+a/+aaaaaaa-[4] = 16.2117
/-/a//aaaaaaa+*+a/+aaaaaaa-[5] = 16.2117
/-/a//aaaaaaa+*+a/+aaaaaaa-[6] = 16.2117
/-/a//aaaaaaa+*+a/+aaaaaaa-[7] = 16.2117
/-/a//aaaaaaa+*+a/+aaaaaaa-[8] = 16.2117
/-/a//aaaaaaa+*+a/+aaaaaaa-[9] = 16.2117
```

Figure 5. Illustration of genetic drift. After 8 generations, the population loses all diversity, and all its members are descendants of chromosome 1 of the initial population (see Figure 4).

tion (see Figure 4). Indeed, replication and selection alone are only capable of causing genetic drift.

2.3.2. 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.

Typically, I use a mutation rate (p_m) equivalent to two point mutations per chromosome. 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.

2.3.3. 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.

2.3.3.1. 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. Typically, a transposition rate (p_{is}) of 0.1 and a set of three IS elements of different lengths is used. 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
-aba+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:

```
0123456789012345601234567890123456
-aba+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.

2.3.3.2. 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.

Typically, I use a root transposition rate (p_{ris}) of 0.1 and a set of three RIS elements of different sizes. 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:

```
0123456789012345601234567890123456
*-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:

```
0123456789012345601234567890123456
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.

2.3.3.3. 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 other operators (all kinds of GEP recombination; see below), for it allows not only the duplication of genes but also a more generalized recombination of genes or smaller building blocks.

The chromosome to undergo gene transposition is randomly chosen, and one of its genes (except the first, obviously) is randomly chosen 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 some sub-ETs in Boolean problems (see Figure 3). Note that, in this case, gene transposition has a very drastic effect, generating most of the times nonviable individuals.

2.3.4. 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 parents.

2.3.4.1. One-point recombination

In 1-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:

```
0123456789012345601234567890123456
+*-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 the crossover point, forming the offspring below:

```
0123456789012345601234567890123456
+*-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. Like the above presented operators, one-point recombination is a very important source of genetic variation, being, after mutation, one of the operators most chosen in gene expression programming. Depending on the rates of the remaining types of recombination, I use a one-point recombination rate (p_{1r}) between 0.3 and 0.7. A good rule of thumb is to use a global crossover rate of 0.7 (the sum of the rates of the three kinds of recombination).

2.3.4.2. Two-point recombination

In 2-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:

```
0123456789012345601234567890123456
*-+Q/Q*QaaabbbbabQQab*++-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 the crossover points. Then, the following chromosomes are created:

```
0123456789012345601234567890123456
*-+Q/+/abaabbbaab/*-aQa*-aabbabaab
Q/-b-Q*QaaabbbbabQQab*++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.

One-point or two-point recombination are, after mutation, the operators most used in GEP. Indeed, the interplay between mutation and one-point and two-point recombination is an excellent source of genetic diversity and is more than sufficient to evolve solutions to virtually all problems.

2.3.4.3. 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 parents. 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
```

The daughter chromosomes contain entire genes from both parents. 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 recombination 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.

2.4. 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

$$y = 3a^2 + 2a + 1 \tag{2.11}$$

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 (Table 1). 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:

$$f_i = \sum_{j=1}^{C_t} \left(M - \left| C_{(i,j)} - T_j \right| \right) \tag{2.12}$$

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 $|C_{(i,j)} - T_j|$ (the precision) less or equal to 0.01, then the precision is equal to zero, and $f_i = f_{max} = C_t M$. 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 (Ferreira 2001).

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 = {+, -, *, /}.

Table 1
Set of 10 random fitness cases used in the simple problem of symbolic regression.

a	f(a)
-4.2605	46.9346
-2.0437	9.44273
-9.8317	271.324
2.7429	29.0563
0.7328	4.07659
-8.6491	208.123
-3.6101	32.8783
-1.8999	8.02906
-4.8852	62.8251
7.3998	180.071

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, the three kinds of transposition, and the three kinds of recombination) (see Table 2).

The parameters used per run are summarized in Table 2. I chose a small population of 20 individuals for this problem 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.

Figure 6 shows the progression of average fitness and the fitness of the best individual of a successful run. In this run, a perfect solution was found in generation 3.

The initial population of this run, together with the fitness of each individual, is shown in Figure 7. Note that three of the 20 individuals are nonviable and thus have fitness zero. The best of generation individual, chromosome 19, has fitness 661.5933. Its expression and the correspondent mathematical equation are shown in Figure 8. Note that gene 2 returns zero and, therefore, might be considered a pseudogene. Note also how the algorithm created constants in all sub-ETs on its own.

Table 2
Parameters for the simple symbolic regression problem.

Number of generations	50
Population size	20
Number of fitness cases	10 (Table 1)
Function set	+ - * /
Gene length	13
Number of genes	3
Linking function	+
Chromosome length	39
Mutation rate	0.051
1-Point recombination rate	0.3
2-Point recombination rate	0.3
Gene recombination rate	0.1
IS transposition rate	0,1
IS elements length	1,2,3
RIS transposition rate	0.1
RIS elements length	1,2,3
Gene transposition rate	0.1
Selection range	100
Precision	0.01

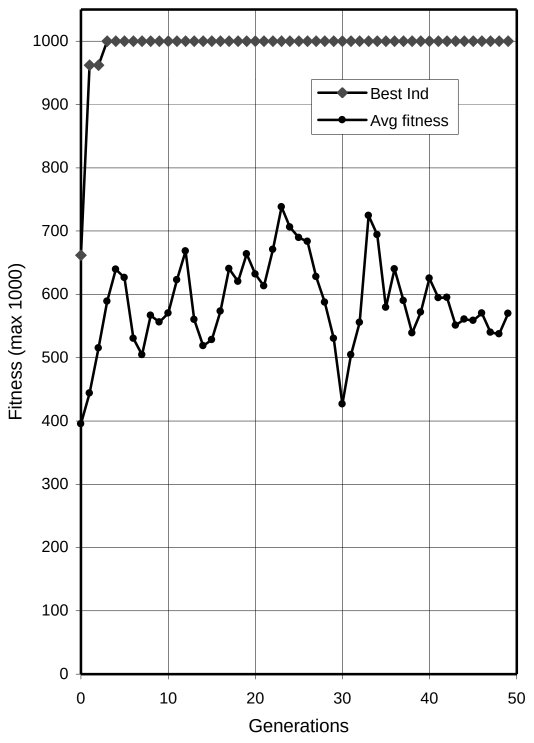


Figure 6. Progression of average fitness of the population and the fitness of the best individual for a successful run of the experiment summarized in Table 2.

The descendants of the individuals of the initial population are shown in Figure 9. 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. This chromosome has a fitness of 961.8512 and its expression is shown in Figure 10.

The descendants of the individuals of this generation are shown in Figure 11 (generation 2). Note that despite the global improvement in fitness (compare the average fitness of both populations in Figure 6), none of the descendants surpassed the best individual of the previous generation.

And finally, in the next generation (generation 3 of Figure 11), an individual with maximum fitness was created. 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 (Figure 12).

3. Function finding

We have already seen how GEP can be used to do symbolic regression in the simple example of section 2.4. However, despite the fact that the target function contained simple numerical constants (3, 2, and 1), there was no explicit facility to generate them: the algorithm created them on its own.

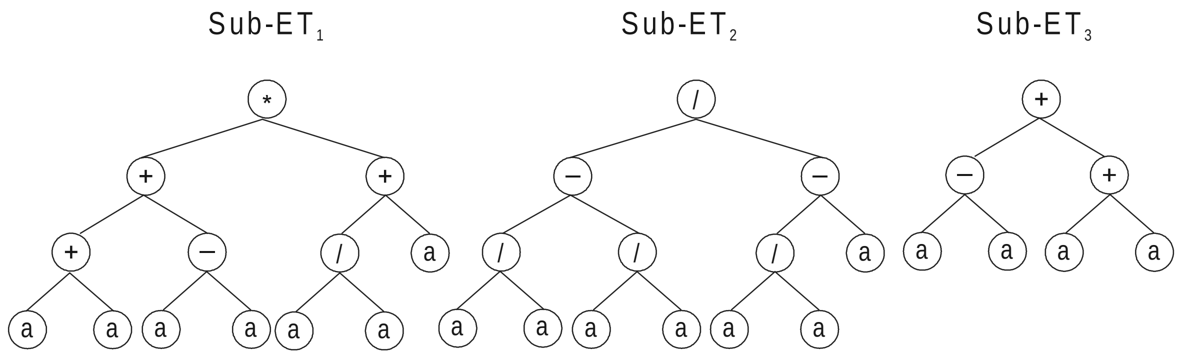
In this section I will show how GEP solves the problem of explicit constant creation to do symbolic regression. Fur-

```
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
```

Figure 7. Initial population (generation 0) for the simple problem of symbolic regression. For each problem, such an initial, totally random population is generated. The value after each chromosome indicates the fitness for the set of fitness cases shown in Table 1.

a)
```
012345678901201234567890120123456789012
*+++-/aaaaaaa/--///aaaaaaa+-+aaaaaaaaaa
```

b)



c)

$$y = \left(2a^2 + 2a\right) + \left(0\right) + \left(2a\right)$$

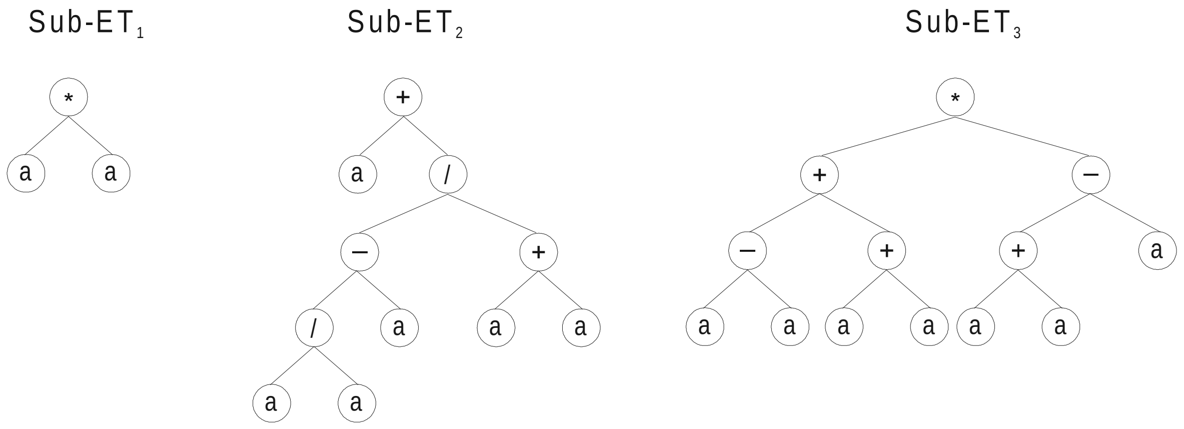
Figure 8. Best individual of generation 0 (chromosome 19 of Figure 7). It has a fitness of 661.5933. **a**) The chromosome of the individual. **b**) The sub-ETs codified by each gene. **c**) The correspondent mathematical expression after linking with addition (the contribution of each sub-ET is shown in brackets).

```
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
```

Figure 9. The descendants of the individuals of the initial population of Figure 7. The value after each chromosome indicates the fitness for the set of fitness cases shown in Table 1. Note that chromosome 0 is the clone of the best individual of the previous generation. In fact, this position is always occupied by the clone of the best individual of the previous generation.

a)
```
012345678901201234567890120123456789012
*aa-+-aaaaaaa+a/-+/aaaaaaa*+--++aaaaaaa
```

b)



c) $$y = (a^2) + \left(a + \frac{1-a}{2a}\right) + (2a^2)$$

Figure 10. Best individual of generation 1 (chromosome 7 of Figure 9) with a fitness of 961.8512. **a**) Its chromosome. **b**) The sub-ETs codified by each gene. **c**) The correspondent mathematical expression after linking with addition (the contribution of each sub-ET is shown in brackets).

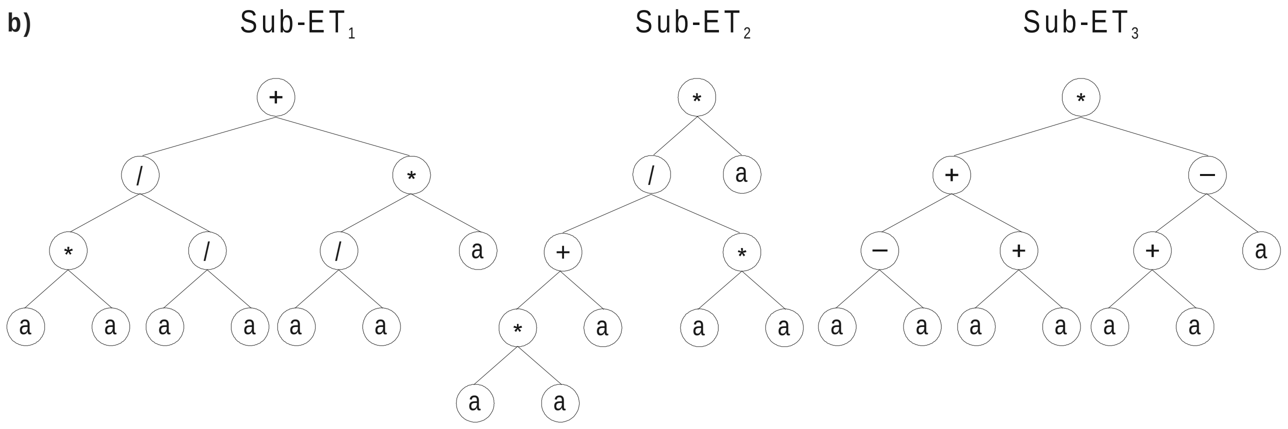
```
Generation N: 2
012345678901201234567890120123456789012
*aa-+-aaaaaaa+a/-+/aaaaaaa*+--++aaaaaaa-[ 0] = 961.8512
*/**+aaaaaaaa*/-+**aaaaaaa***/-*aaaaaaa-[ 1] = 446.2061
+-*-**aaaaaaa*+a//-aaaaaaa-/+*+*aaaaaaa-[ 2] = 323.1036
+--+//aaaaaaa+*+/*-aaaaaaa/*-*-+aaaaaaa-[ 3] = 551.2066
*aa-+-aaaaaaa+a/++/aaaaaaa***+-*aaaaaaa-[ 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


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
```

Figure 11. The chromosomes of two populations for the simple problem of symbolic regression. The value after each chromosome indicates the fitness for the set of fitness cases shown in Table 1. In **generation 2**, none of the individuals surpassed the best of the previous generation. In **generation 3**, a perfect solution with maximum fitness was found (chromosome 13).

a)
```
012345678901201234567890120123456789012
+/**//aaaaaaa*/a+**aaaaaaa*+--++aaaaaaa
```

b)



c) $y = (a^2 + a) + (a + 1) + (2a^2) = 3a^2 + 2a + 1$

Figure 12. Perfect solution found in generation 3 (chromosome 13 of Figure 11). It has the maximum value 1000 of fitness. **a**) The chromosome of this individual. **b**) The sub-ETs codified by each gene. **c**) The correspondent mathematical expression after linking with addition (the contribution of each sub-ET is shown in brackets).

thermore, I will also show that the explicit use of constants is, in fact, better avoided, for GEP produces better results when the numerical constants are not explicitly used and the system is left to invent new ways of representing them.

3.1. Function finding and the creation of numerical constants

Here I show two different approaches to the problem of constant creation: one without using ephemeral random constants, and another using ephemeral random constants. In the first approach, a special facility to handle numerical constants was implemented. In the second approach, the system creates them or finds alternative ways of representing them on its own.

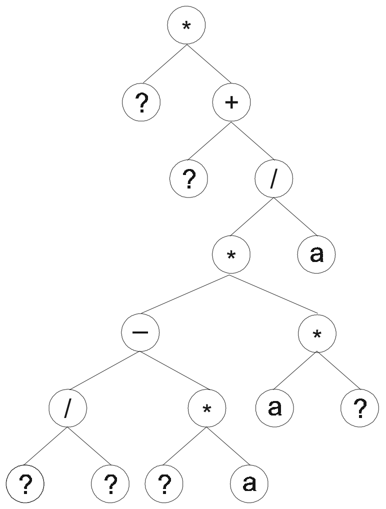
Numerical constants can be easily implemented in GEP (Ferreira 2001). For that an additional domain Dc was created. Structurally, the Dc comes after the tail, has a length equal to t, and consists of the symbols used to represent the ephemeral random constants. Therefore, another region with its boundaries and its own alphabet was created in the chromosome.

For each gene the constants are randomly generated at the beginning of a run, but their circulation is guaranteed by the genetic operators. Besides, a special mutation operator was created that allows the permanent introduction of variation in the set of random constants. A domain specific IS transposition was also created in order to guarantee the effective shuffling of constants. Note that the basic genetic operators are not affected by the Dc: it is only necessary to keep the boundaries of each region and not mix different alphabets.

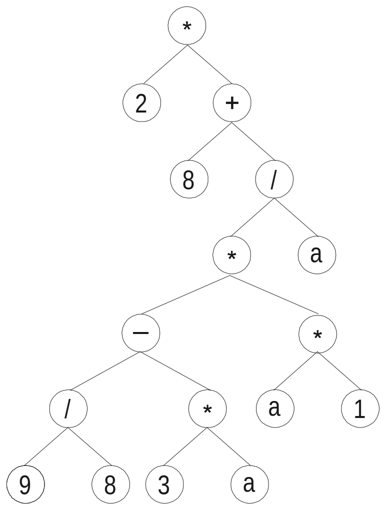
Consider the single-gene chromosome with an h = 11 (the Dc is shown in bold):

```
01234567890123456789012345678901234
*?+?/*a-*/*a????a??a??a281983874486   (3.1)
```

where '?' represents the ephemeral random constants. The expression of this kind of chromosomes is done exactly as before, obtaining:



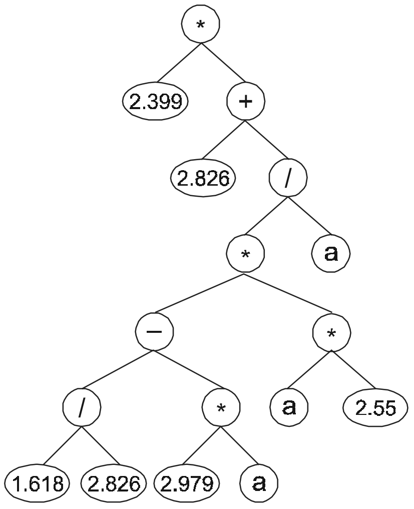
Then the ?'s in the ET are replaced from left to right and from top to bottom by the symbols in Dc, obtaining:



The values corresponding to these symbols are kept in an array. For simplicity, the number represented by the numeral indicates the order in the array. For instance, for the 10 element array,

A = {-2.829, 2.55, 2.399, 2.979, 2.442, 0.662, 1.797, -1.272, 2.826, 1.618},

the chromosome 3.1 above gives:



So, in this section, we will compare the two approaches searching a solution to a relatively complex problem. The test function chosen is the following 'V' shaped function:

$$y = 4.251a^2 + \ln(a^2) + 7.243e^a \tag{3.2}$$

where a is the independent variable and e is the irrational number 2.71828183. For both approaches, we will compare the results obtained for 100 independent runs of 5000 generations each (see Table 4 below).

3.1.1. First approach: direct manipulation of rational constants

For the first approach, the function set contained, besides the expected functions, several extraneous functions, being in this case F = {+, -, *, /, L, E, K, ~, S, C} ('L' represents the natural logarithm, 'E' represents e^x, 'K' represents the logarithm of base 10, '~' represents 10^x, 'S' represents the sine function, and 'C' represents the cosine), T = {a, ?}, the set of random constants R = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, and the ephemeral random constant '?' ranged over the interval [-1, 1]. The set of 20 random fitness cases chosen from the interval [-1,1] are shown in Table 3 and the fitness was evaluated by a variant of equation 2.12 (Ferreira 2001):

$$f_i = \sum_{j=1}^{C_t}\left(M - \left| \frac{C_{(i,j)} - T_j}{T_j} \cdot 100 \right| \right) \tag{3.3}$$

If $\left| \frac{C_{(i,j)} - T_j}{T_j} \cdot 100 \right|$ (the precision) less or equal to 0.01%, then the precision is equal to zero and $f_{(i,j)} = M$. For this problem, M = 100% and C_t = 20; therefore, f_{max} = 2000.

Table 3
Set of 20 random fitness cases used in the finding of the 'V' shaped function.

a	f(a)
-0.2639725157548009	3.194980662652764
0.0578905532656938	1.990520017259985
0.3340252901096346	8.396637039972868
-0.2363345775644623	3.070889769728257
-0.8557443825668047	5.879467636957033
-0.0194437136332785	-0.7753263223284588
-0.1921343881833043	2.834702257744086
0.5293079101246271	12.21547266421373
-0.007889741187284598	-2.498039834186359
0.4389698049506311	10.40717348588088
-0.1075592926980396	2.094136356459081
-0.2745569943771633	3.239272780108398
-0.05953332196045281	1.197012847673475
0.3844929939583523	9.355807691898551
-0.8749230207363339	6.006424530013026
-0.236546636250546	3.071897290438372
-0.1678759417045577	2.674400531309863
0.9506821818220914	22.48196398441491
0.9469791595773622	22.37501611873555
0.6393399100595915	14.5701285332337

In this experiment, 100 identical runs were made. The parameters used per run are shown in the first column of Table 4. The best solution was found in run 79 after 3619 generations. Figure 13 shows the progression of average fitness of the population and the fitness of the best individual for run 79 of this experiment. The best of run solution in terms of R-square is shown below (the sub-ETs are linked by addition):

```
Gene 0: L*~*+/aa?a??a2132990
```
A_0 = {0.565, 0.203, 0.613, 0.219, 0.28, 0.25, 0.48, 0.427, 0.821, 0.127}

```
Gene 1: E-+-*?aaaaaaa7332660
```
A_1 = {0.031, 0.046, 0.696, 0.643, 0.528, 0.417, 0.978, 0.811, 0.637, 0.988}

```
Gene 2: ~Saaa+??aa??a9109969
```
A_2 = {0.515, 0.466, 0.254, 0.219, 0.425, 0.942, 0.306, 0.619, 0.821, 0.262}

```
Gene 3: ~SSaES?????aa5420661
```
A_3 = {0.595, 0.547, 0.525, 0.219, 0.297, 0.387, 0.508, 0.695, 0.728, 0.415} (3.4)

It has a fitness of 1975.264 and an R-square of 0.9999439 evaluated over the set of 20 fitness cases and an R-square of 0.9999075 evaluated against a test set of 100 random points. Its expression is shown in Figure 14. This model is a very good approximation to the target function as both the R-square and the comparison of the plots for the target function and the model show (Figure 15).

It is worth noticing that, despite integrating constants in the evolved solutions, the constants are very different from the expected ones. Indeed, GEP (and I believe, all genetic algorithms) can find the expected constants with a precision to the third or fourth decimal place when the target functions are simple polynomial functions with rational coefficients and/or when we could guess pretty accurately the function set, otherwise a very creative solution would be found. I don't think this should be seen as a weakness of evolutionary algorithms for "constants" is apparently just another word for mathematical expression.

3.1.2. Second approach: creation of rational constants from scratch

For the second approach, the evolution of a model without directly using random constants, the set of fitness cases

Table 4
General settings used in the finding of the 'V' shaped function with and without random constants.

	With Random Constants	Without Random Constants
Number of runs	100	100
Number of generations	5000	5000
Population size	100	100
Number of fitness cases	20 (Table 3)	20 (Table 3)
Function set	+ - * / L E K ~ S C	+ - * / L E K ~ S C
Head length	6	6
Number of genes	4	5
Linking function	+	+
Chromosome length	80	65
Mutation rate	0.044	0.044
1-Point recombination rate	0.3	0.3
2-Point recombination rate	0.3	0.3
Gene recombination rate	0.1	0.1
IS transposition rate	0.1	0.1
IS elements length	1,2,3	1,2,3
RIS transposition rate	0.1	0.1
RIS elements length	1,2,3	1,2,3
Gene transposition rate	0.1	0.1
Rand. const. mut. rate	0.01	--
Dc specific IS transp. rate	0.1	--
Dc specific IS elements length	1,2,3	--
Selection range	100%	100%
Precision	0.01%	0.01%
Average best-of-run fitness	1850.476	1934.619

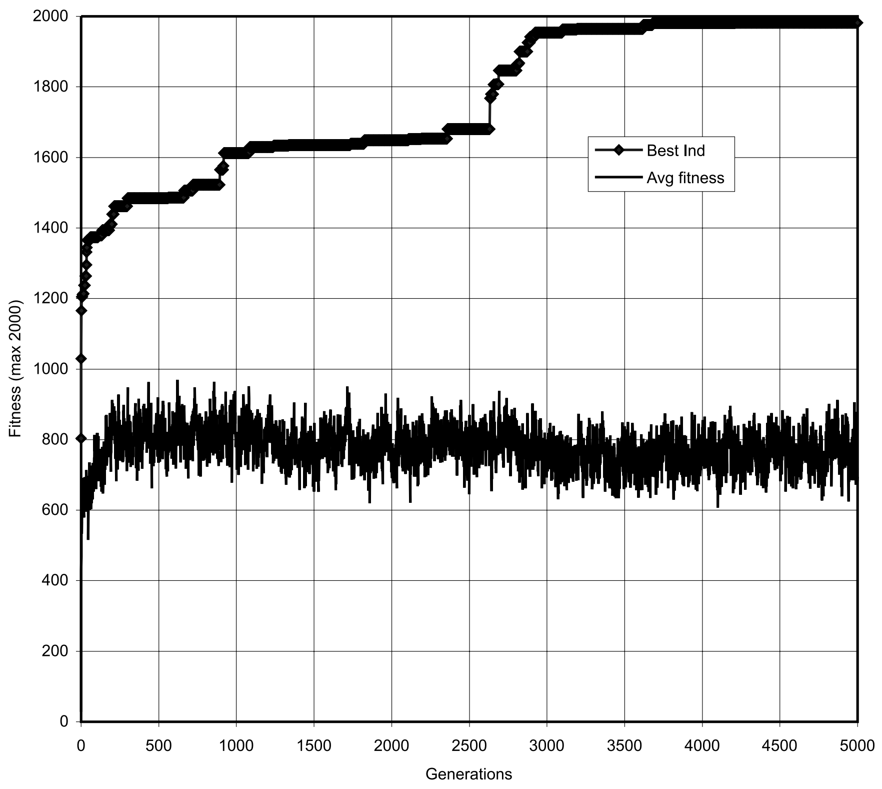
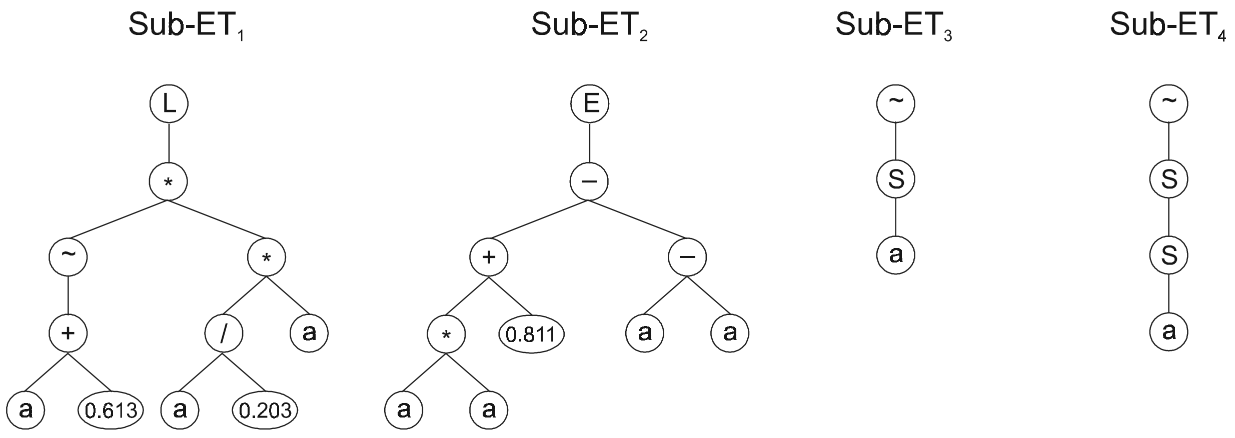


Figure 13. Progression of average fitness of the population and the fitness of the best individual of run 79 of the experiment summarized in Table 4, column 1 (function finding with random constants).

a)



b)

$$y = \left[\ln^{\left(10^{(a+0.613)} \cdot \frac{a^2}{0.203} \right)} \right] + \left[e^{\left(a^2+0.811\right)} \right] + \left[10^{\sin(a)} \right] + \left[10^{\sin(\sin(a))} \right]$$

Figure 14. Model 3.4 evolved by GEP using the facility for manipulation of random constants. **a**) The sub-ETs codified by each gene. **b**) The correspondent mathematical expression after linking with addition (the contribution of each sub-ET is shown in square brackets).

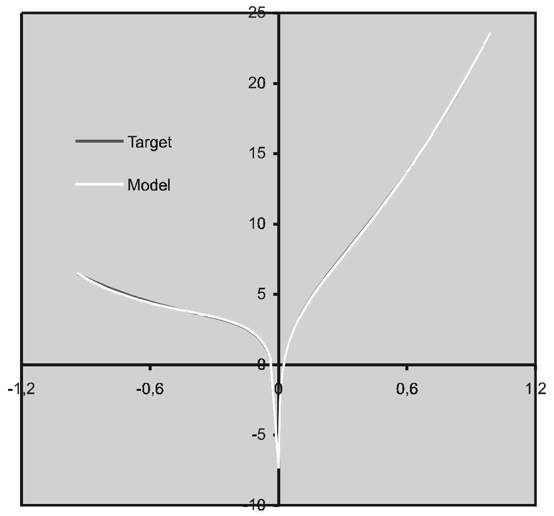


Figure 15. Comparison of the target function with the model 3.4 evolved by GEP using random constants (Figure 14). The R-square was evaluated over the test set of 100 random points and is equal to 0.9999075.

and the function set were as in section 3.1.2, and T = {a}. The parameters used per run are shown in the second column of Table 4. In this experiment of 100 identical runs, the best solution was found in generation 1210 of run 63:

```
0123456789012
+EL-*/aaaaaaa
~a+E/Laaaaaaa
+C+C+Eaaaaaaa
*C~+aSaaaaaaa
~a-L~+aaaaaaa                        (3.5)
```

It has a fitness of 1982.488 and an R-square of 0.99996922 evaluated over the set of 20 fitness cases and an R-square of 0.9999460 evaluated against the same test set of section 3.1.1, and thus is better than the model 3.4 evolved with the facility for creation of random constants. This model is also an almost perfect match for the target function. Its expression is shown in Figure 16.

Once again, the plots of the target function and the model evolved by GEP are compared in Figure 17.

It is instructive to compare the results obtained in both approaches. Not only the model 3.5 evolved without random

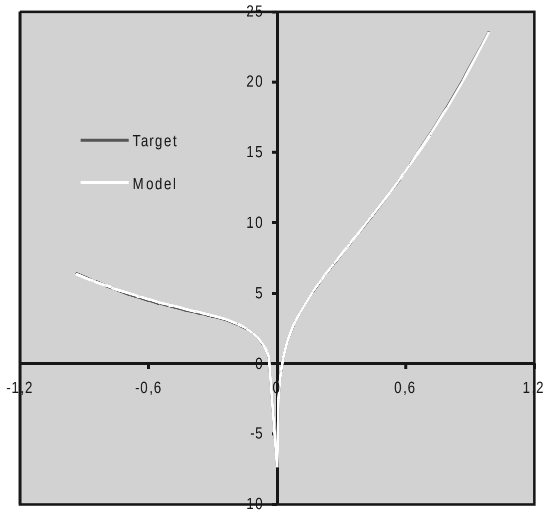
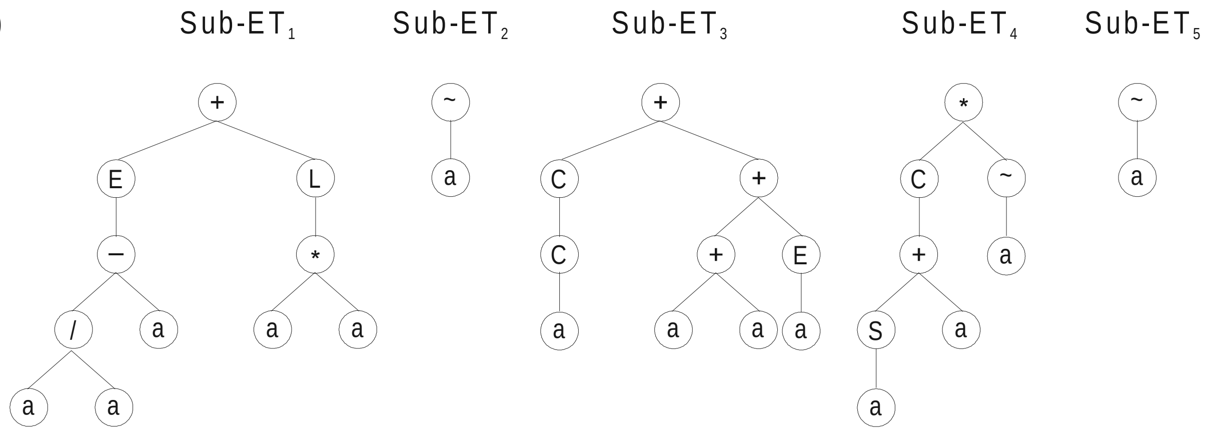


Figure 17. Comparison of the target function with the model 3.5 evolved by GEP without explicitly using random constants (Figure 16). The R-square was evaluated over the test set of 100 random points and is equal to 0.9999460.

a)
```
01234567890120123456789012012345678901201234567890120123456789012
+EL-*/aaaaaaa~a+E/Laaaaaaa+C+C+Eaaaaaaa*C~+aSaaaaaaa~a-L~+aaaaaaa
```

b)



c)

$$y=\left[e^{(1-a)}+\ln\left(a^{2}\right)\right]+\left[10^{a}\right]+\left[\cos(\cos(a))+2a+e^{a}\right]+\left[\cos(\sin(a)+a)\cdot10^{a}\right]+\left[10^{a}\right]$$

Figure 16. Model evolved by GEP without explicitly using random constants. **a**) The model in Karva notation. **b**) The sub-ETs codified by each gene. **c**) The correspondent mathematical expression after linking with addition (the contribution of each sub-ET is shown in square brackets).

constants was better than the model evolved with random constants, but also the average best-of-run fitness was superior in the second approach: 1934.619 compared to 1850.476 (see Table 4). Thus, in real-world applications where complex realities are modeled, of which it is impossible to infer neither the type nor the range of the numerical constants, and where most of the times we are unable to guess the exact function set, it is more appropriate to let the system model the reality on its own without explicitly using random constants. Perhaps the models would be unconventional comparatively to human-created ones, but they allow, nonetheless, the extraction of knowledge because the programs evolved by GEP are simple and accessible. It is worth noticing that some learning algorithms like neural networks do not allow knowledge extraction from their models whereas others produce so complicated models that their analysis is considerably limited.

3.2. Function finding on a five-dimensional parameter space

The objective of this section is to show how GEP can be used to model complex realities with high accuracy. The test function chosen is the following five parameter function:

$$y = \frac{\sin(a)\cdot\cos(b)}{\sqrt{10^c}} + \tan(d-e) \tag{3.6}$$

where *a*, *b*, *c*, *d*, and *e* are the independent variables.

Consider we are given a sampling of the numerical values from this function over 100 random points in the interval [-1,1] and we wanted to find a function fitting those values within 0.01% of the correct value. The fitness was evaluated by equation 3.3, being M = 100%. Thus, for C_t = 100, f_{max} = 10000.

The domain of this problem suggests, besides the arithmetical functions, the use of sqrt(x), log(x), 10^x, sin(x), cos(x) and tan(x) in the function set, which corresponds respectively to Q, K, ~, S, C, and G. Thus, for this problem, F = {+, -, *, /, Q, K, ~, S, C, G} and T consisted obviously of the independent variables {a, b, c, d, e}.

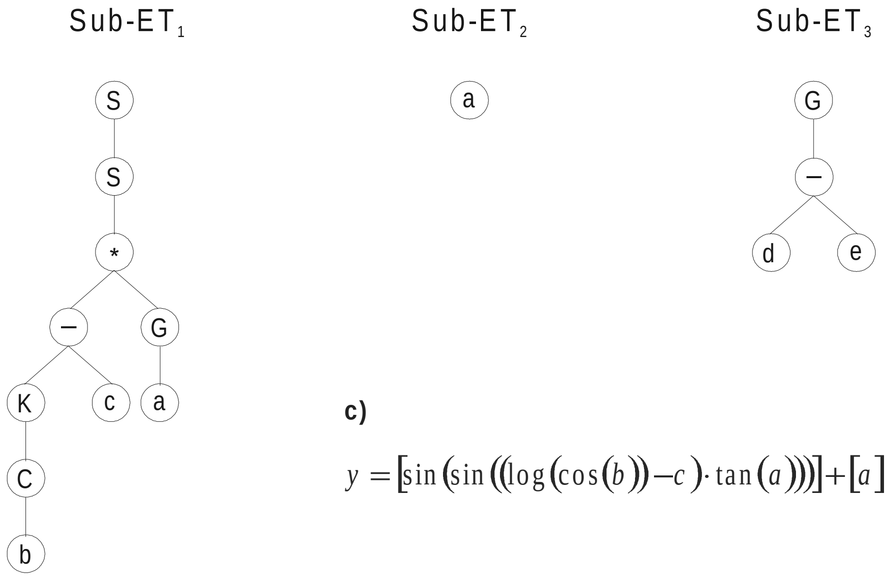
For this problem, I chose 3-genic chromosomes encoding sub-ETs with a maximum of 19 nodes. The sub-ETs were posttranslationally linked by addition. The parameters used per run are summarized in Table 5. I used the software Automatic Problem Solver (APS) to model this function because it allows the easy optimization of intermediate solutions and the easy testing of the evolved models against a test set. In one run a very good solution, with an R-square of 0.9999913 evaluated over a test set of 200 random points, was found:

```
0123456789012345678
SS*-GKcaCbbccbeabdb
aC--SKaeGceadddabad
G-de*add+adedabdeaa
```
(3.7)

Its expression is shown in Figure 18. This model is a very good approximation to the target function 3.6 as the high

a)
```
012345678901234567801234567890123456780123456789012345678
SS*-GKcaCbbccbeabdbaC--SKaeGceadddabadG-de*add+adedabdeaa
```

b)



c)

$$y = \left[\sin\left(\sin\left(\left(\log\left(\cos\left(b\right)\right)-c\right)\cdot\tan\left(a\right)\right)\right)\right]+\left[a\right]+\left[\tan(d-e)\right]$$

Figure 18. Model evolved by GEP to the 5-parameter function 3.6. **a**) The model in Karva notation. **b**) The sub-ETs codified by each gene. **c**) The correspondent mathematical expression after linking with addition (the contribution of each sub-ET is shown in square brackets).

Table 5
Parameters for the problem of function finding on a five-dimensional parameter space.

Number of generations	1000
Population size	100
Number of fitness cases	100
Function set	+ - * / Q K ~ S C G
Gene length	19
Number of genes	3
Linking function	+
Chromosome length	57
Mutation rate	0.044
1-Point recombination rate	0.3
2-Point recombination rate	0.3
Gene recombination rate	0.1
IS transposition rate	0,1
IS elements length	1,2,3
RIS transposition rate	0.1
RIS elements length	1,2,3
Gene transposition rate	0.1
Selection range	100%
Precision	0%

value for the R-square (almost 1) indicates. With APS we can further convert the evolved Karva programs into a more conventional computer program. For instance, the model 3.7 above can be automatically translated into the following C++ function:

```
double APSCfunction(double d[ ])
{
    double dblTemp = 0;
    dblTemp+=sin(sin(((log10(cos(d[1]))-d[2])*tan(d[0]))));
    dblTemp += d[0];
    dblTemp += tan((d[3]-d[4]));
    return dblTemp;
}
```

Note that the term encoded in the last gene matches exactly the second term of the target function. However, a very unconventional and non-parsimonious alternative was found to express the first term of the target function. But the model evolved by GEP is, nonetheless, extremely accurate as the high value for the R-square indicates.

4. Summary

On the one hand, the implementation details of the learning algorithm, gene expression programming, were thoroughly presented, allowing its easy understanding and implementation. On the other hand, the workings of the algorithm were analyzed step-by-step with a simple problem of symbolic regression. Furthermore, the question of constant creation in symbolic regression was discussed comparing two different approaches to solve this problem: one with the explicit use of rational constants, and another without them. The results presented suggest that the latter is best, not only in terms of the accuracy of the evolved models and overall performance evaluated in terms of average best-of-run fitness, but also because the search space is much smaller, reducing greatly the complexity of the system. Moreover, we also saw how GEP efficientely searched for a solution to a complex problem on a five-dimensional parameter space with several extraneous functions, finding an almost perfect solution with an R-square of 0.9999913.

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.