Function Finding and the Creation of Numerical Constants in Gene Expression Programming

Cândida Ferreira
Gepsoft, 37 The Ridings,
Bristol BS13 8NU, UK
candidaf@gepsoft.com
www.gene-expression-programming.com/author.asp

7th Online World Conference on Soft Computing in Industrial Applications, September 23 - October 4, 2002

Gene expression programming is a genotype/phenotype system that evolves computer programs of different sizes and shapes (the phenotype) encoded in linear chromosomes of fixed length (the genotype). The chromosomes are composed of multiple genes, each gene encoding a smaller sub-program. Furthermore, the structural and functional organization of the linear chromosomes allows the unconstrained operation of important genetic operators such as mutation, transposition, and recombination. In this work, three function finding problems, including a high dimensional time series prediction task, are analyzed in an attempt to discuss the question of constant creation in evolutionary computation by comparing two different approaches to the problem of constant creation. The first algorithm involves a facility to manipulate random numerical constants, whereas the second finds the numerical constants on its own or invents new ways of representing them. The results presented here show that evolutionary algorithms perform considerably worse if numerical constants are explicitly used.

1. Introduction

Genetic programming (GP) evolves computer programs by genetically modifying nonlinear entities with different sizes and shapes (Koza 1992). These nonlinear entities can be represented as diagrams or trees. Gene expression programming (GEP) is an extension to GP that also evolves computer programs of different sizes and shapes, but the programs are encoded in a linear chromosome of fixed length (Ferreira 2001). One strength of the GEP approach is that the creation of genetic diversity is extremely simplified as genetic operators work at the chromosome level. Indeed, due to the structural organization of GEP chromosomes, the implementation of high-performing search operators is extremely simplified, as any modification made in the genome always results in valid programs. Another strength of GEP consists of its unique, multigenic nature which allows the evolution of complex programs composed of several simpler sub-programs.

It is assumed that the creation of floating-point constants is necessary to do symbolic regression in general (see, e.g., Banzhaf 1994 and Koza 1992). Genetic programming solved the problem of constant creation by using a special terminal named “ephemeral random constant” (Koza 1992). For each ephemeral random constant used in the trees of the initial population, a random number of a special data type in a specified range is generated. Then these random constants are moved around from tree to tree by the crossover operator.

Gene expression programming solves the problem of constant creation differently (Ferreira 2001). GEP uses an extra terminal “?” and an extra domain Dc composed of the symbols chosen to represent the random constants. For each gene, the random constants are generated during the inception of the initial population and kept in an array. The values of each random constant are only assigned during gene expression. Furthermore, a special operator is used to introduce genetic variation in the available pool of random constants by mutating the random constants directly. In addition, the usual operators of GEP plus a Dc specific transposition guarantee the effective circulation of the numerical constants in the population. Indeed, with this scheme of constants manipulation, the appropriate diversity of numerical constants can be generated at the beginning of a run and maintained easily afterwards by the genetic operators.

Notwithstanding, in this work it is shown that evolutionary algorithms do symbolic regression more efficiently if the problem of constant creation is handled by the algorithm itself. In other words, the special facilities for manipulating random constants are indeed unnecessary to solve problems of symbolic regression.

2. Genetic algorithms with tree representations

All genetic algorithms use populations of individuals, select individuals according to fitness, and introduce genetic variation using one or more genetic operators (see, e.g., Mitchell 1996). In recent years different systems have been developed so that this powerful algorithm inspired in natural evolution could be applied to a wide spectrum of problem domains (see, e.g., Mitchell 1996 for a review of recent work on genetic algorithms and Banzhaf *et al*. 1998 for a review of recent work on GP).

Structurally, genetic algorithms can be subdivided in three fundamental groups: i) Genetic algorithms with individuals consisting of linear chromosomes of fixed length devoid of complex expression. In these systems, replicators (chromosomes) survive by virtue of their own properties. The algorithm invented by Holland (1975) belongs to this group and is known as genetic algorithm or GA; ii) Genetic algorithms with individuals consisting of ramified structures of different sizes and shapes and, therefore, capable of assuming a richer number of functionalities. In these systems, replicators (ramified structures) also survive by virtue of their own properties. The algorithm invented by Cramer (1985) and later developed by Koza (1992) belongs to this group and is known as genetic programming or GP; iii) Genetic algorithms with individuals encoded in linear chromosomes of fixed length which are afterwards expressed as ramified structures of different sizes and shapes. In these systems, replicators (chromosomes) survive by virtue of causal effects on the phenotype (ramified structures). The algorithm invented by myself (Ferreira 2001) belongs to this group and is known as gene expression programming or GEP.

GEP shares with GP the same kind of ramified structure and, therefore, can be applied to the same problem domains. However, the logistics of both systems differ significantly and the existence of a real genotype in GEP allows the unprecedented manipulation and exploration of more complex systems. Below are briefly highlighted some of the differences between GEP and GP.

2.1. Genetic programming

As simple replicators, the ramified structures of GP are tied up in their own complexity: on the one hand, bigger, more complex structures are more difficult to handle and, on the other, the introduction of genetic variation can only be done at the tree level and, therefore, must be done carefully so that valid structures are created. A special kind of tree crossover is practically the only source of genetic variation used in GP for it allows the exchange of sub-trees and, therefore, always produces valid structures. Indeed, the implementation of high-performing operators, like the equivalent of natural point mutation, is unproductive as most mutations would result in syntactically invalid structures. Understandingly, the other genetic operators described by Koza (1992) – mutation and permutation – also operate at the tree level.

2.2. Gene expression programming

The phenotype of GEP individuals consists of the same kind of diagram representation used by GP. However, these complex phenotypes are encoded in simpler, linear structures of fixed length – the chromosomes. Thus, the main players in GEP are the chromosomes and the ramified structures or expression trees (ETs), the latter being the expression of the genetic information encoded in the former. The decoding of GEP genes 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 multigenic systems.

In GEP there are therefore two languages: the language of the genes and the language of ETs. However, 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 the genotype, and *vice versa*. This bilingual and unequivocal system is called *Karva* language. The details of this new language are given below.

2.2.1. Open reading frames and genes

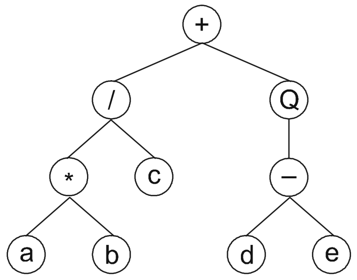
In GEP, the genome or chromosome consists of a linear, symbolic string of fixed length composed of one or more genes. Despite their fixed length, GEP chromosomes code for ETs with different sizes and shapes, as will next be shown.

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 does not always coincide with the last position of a gene. It is common for GEP genes to have noncoding regions downstream of the termination point. (For now these noncoding regions will not be considered because they do not 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 or ET:



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:

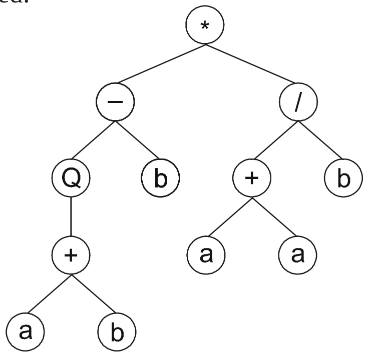
```
0123456789
+/Q*c-abde                                    (2.2)
```

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). These ORFs are called 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 express correctly the ORF, the rules governing the spatial distribution of functions and terminals must be followed. 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 stated previously, 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 but the length of the ORFs. Indeed, the length of an ORF may be equal to 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.

As will next be shown, the junk sequences of GEP genes are extremely important for they allow the modification of the genome using any genetic operator without restrictions, always producing syntactically correct programs. The section proceeds with the study of the structural organization of GEP genes in order to show how these genes invariably code for syntactically correct programs and why they allow an unconstrained application of any genetic operator.

2.2.2. Structural organization of genes

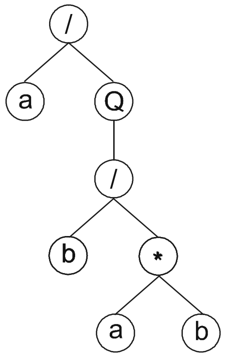
GEP genes are composed of a head and a tail. The head contains symbols that represent both functions and terminals, whereas the tail contains only terminals. For each problem, the length of the head h is chosen, whereas the length of the tail t is a function of h and maximum arity n, and is evaluated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

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

Consider a gene for which the set of functions consists of 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 with only eight nodes:



Note that 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)
```

In this case, its expression gives a new ET with 18 nodes. Note that 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”, obtaining:

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

Its expression results in a new ET with six nodes. Note that the ORF ends at position 5, shortening the parental ET in two nodes.

So, despite their fixed length, GEP genes have 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, always results in a structurally correct ET as long as the structural organization of genes is maintained. Indeed, the implementation of high-performing genetic operators in GEP is a child's play, and Ferreira (2001) describes seven: point mutation, RIS and IS transposition, two-point and one-point recombination, gene transposition and gene recombination.

2.2.3. 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 (genes are shown separately):

```
012345678901234
Q/*b+Qababaabaa
-abQ/*+bababbab
**-*bb/babaaaab                 (2.8)
```

It has three ORFs, and each ORF codes for a sub-ET. Position 0 marks the start of each gene. The end of each ORF, though, is only evident upon construction of the respective sub-ET. In this case, the first ORF ends at position 8; the second ORF ends at position 2; and the last ORF ends at position 10. Thus, GEP chromosomes are composed of one or more ORFs, each ORF coding for a structurally and functionally unique sub-ET. Depending on the problem at hand, the sub-ETs encoded by each gene 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 are usually linked by addition or multiplication whereas Boolean sub-ETs are usually linked by OR, AND, or IF.

3. Function finding and the creation of numerical constants

In this section, after a brief presentation of the facility for the explicit handling of random numerical constants in GEP, the problem of constant creation is discussed by comparing the performance of two different algorithms. The first manipulates explicitly the numerical constants and the second solves the problem of constant creation in symbolic regression by creating constants from scratch or by inventing new ways of representing them.

3.1. Manipulating numerical constants in gene expression programming

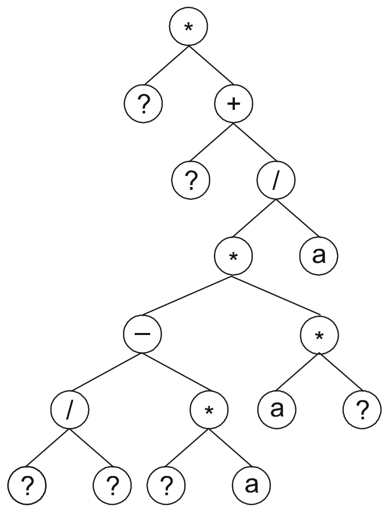
Numerical constants can be easily implemented in GEP (Ferreira 2001). For that an additional domain Dc was created in the gene. Structurally, the Dc comes after the tail, has a length equal to *t*, and is composed of the symbols used to represent the random constants. Therefore, another region with defined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

For each gene the constants are randomly generated at the beginning of a run, but their circulation is guaranteed by the usual genetic operators of mutation, transposition, and recombination. Besides, a special mutation operator allows the permanent introduction of variation in the set of random constants and a domain specific IS transposition guarantees a more generalized shuffling of constants.

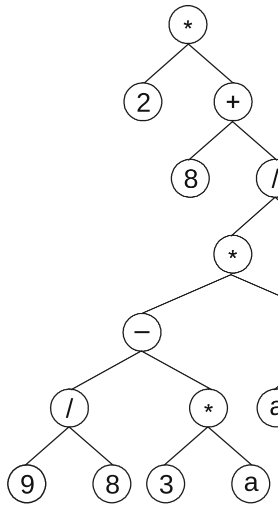
Consider the single-gene chromosome with an $h = 11$ (the Dc is shown in bold):

```
0123456789012345678901234567890123 4
*?+?/*a-*/*a????a??a??a281983874486    (3.1)
```

where "?" represents the random constants. The expression of this kind of chromosome is done exactly as before, giving:



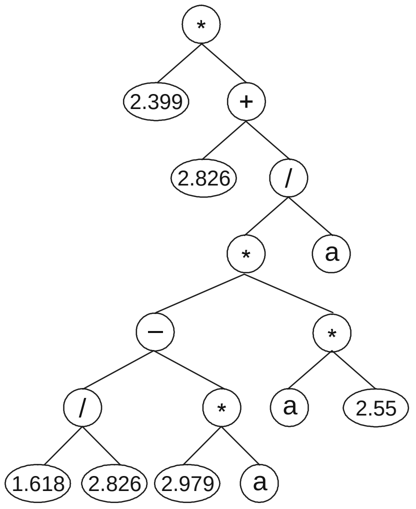
Then the ?'s in the ET are replaced from left to right and from top to bottom by the symbols in Dc, obtaining:



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:



3.2. Two approaches to the problem of constant creation

The comparison between the two approaches (with and without the facility to manipulate random constants) was made on three different problems. The first is a problem of sequence induction requiring integer constants. The *n*th term *N* of the chosen sequence is given by the formula:

$$N = 5a_n^4 + 4a_n^3 + 3a_n^2 + 2a_n + 1 \tag{3.2}$$

where a_n consists of the nonnegative integers. This sequence was chosen because it can be exactly solved and therefore can provide an accurate measure of performance in terms of success rate.

The second is a problem of function finding requiring floating-point constants. In this case, the following "V" shaped function was chosen:

$$y = 4.251a^2 + \ln(a^2) + 7.243e^a \tag{3.3}$$

where *a* is the independent variable and *e* is the irrational number 2.71828183. Problems of this kind cannot be exactly solved by evolutionary algorithms and, therefore, the performance of both approaches is compared in terms of average best-of-run fitness and average best-of-run R-square.

The third is the well-studied benchmark problem of predicting sunspots (Weigend *et al*. 1992). In this case, 100 observations of the Wolfer sunspots series were used (Table 1) with an embedding dimension of 10 and a delay time of one. Again, the performance of both approaches is compared in terms of average best-of-run fitness and R-square.

Table 1
Wolfer sunspots series (read by rows).

101	82	66	35	31	7	20	92
154	125	85	68	38	23	10	24
83	132	131	118	90	67	60	47
41	21	16	6	4	7	14	34
45	43	48	42	28	10	8	2
0	1	5	12	14	35	46	41
30	24	16	7	4	2	8	17
36	50	62	67	71	48	28	8
13	57	122	138	103	86	63	37
24	11	15	40	62	98	124	96
66	64	54	39	21	7	4	23
55	94	96	77	59	44	47	30
16	7	37	74				

3.2.1. Setting the system

For the sequence induction problem, the first 10 positive integers a_n and their corresponding term *N* were used as fitness cases. The fitness function was based on the relative error with a selection range of 20% and maximum precision (0% error), giving maximum fitness f_{max}= 200 (Ferreira 2001).

For the "V" shaped function problem, a set of 20 random fitness cases chosen from the interval [-1, 1] was used. The fitness function used was also based on the relative error but in this case a selection range of 100% was used, giving f_{max}= 2000.

For the time series prediction problem, using an embedding dimension of 10 and a delay time of one, the sunspots series presented in Table 1 result in 90 fitness cases. In this case, a wider selection range of 1000% was chosen, giving f_{max}= 90,000.

In all the experiments, the selection was made by roulette-wheel sampling coupled with simple elitism and the performance was evaluated over 100 independent runs. The six experiments are summarized in Table 2.

3.2.2. First approach: Direct manipulation of numerical constants

To solve the sequence induction problem using random constants, F = {+, -, *}, T = {a, ?}, the set of integer random constants R = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, and “?” ranged over the integers 0, 1, 2, and 3. The parameters used per run are shown in the first column of Table 2. In this experiment, the first perfect solution was found in generation 45 of run 9 (the sub-ETs are linked by addition):

Gene 0:
```
*-aa+-a?aaa??1742174
```
A_0 = {0, 0, 2, 3, 0, 2, 1, 1, 1, 3}

Gene 1:
```
++*/+-?aaa???4460170
```
A_1 = {3, 0, 2, 2, 1, 3, 1, 0, 0, 1}

Gene 2:
```
*a**++aa?aa??4101213
```
A_2 = {1, 2, 3, 3, 2, 2, 0, 1, 1, 2}

Gene 3:
```
**+--+?aaa???2637797
```
A_3 = {0, 0, 2, 3, 3, 3, 0, 0, 1, 0}

Gene 4:
```
+?*++?aaaa?a?2890192
```
A_4 = {1, 1, 0, 1, 1, 3, 1, 0, 0, 2}

Gene 5:
```
-+-/*-?aa?a?a8147432
```
A_5 = {0, 0, 0, 2, 0, 2, 2, 0, 0, 0}

Gene 6:
```
**aa**?aa?a??2314518
```
A_6 = {0, 2, 3, 2, 3, 1, 3, 2, 3, 0}

which corresponds to the target sequence (3.2).

As shown in the first column of Table 2, the probability of success for this problem is 16%, considerably lower than the 81% of the second approach (see Table 2, column 2). It is worth emphasizing that only the prior knowledge of the solution enabled us, in this case, to choose correctly the type and the range of the random constants.

To find the “V” shaped function using random constants 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) and T = {a, ?}. The set of rational random constants R = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, and “?” ranged over the interval [-1, 1]. The parameters used

Table 2
General settings used in the sequence induction (SI), the “V” function, and sunspots (SS) problems. The “*” indicates the explicit use of random constants.

	SI*	SI	V*	V	SS*	SS
Number of runs	100	100	100	100	100	100
Number of generations	100	100	5000	5000	5000	5000
Population size	100	100	100	100	100	100
Number of fitness cases	10	10	20	20	90	90
Function set	+ - * /	+ - * /	+ - * / L E K ~ S C	+ - * / L E K ~ S C	4 (+ - * /)	4 (+ - * /)
Terminal set	a, ?	a	a, ?	a	a - j, ?	a - j
Random constants array length	10	--	10	--	10	--
Random constants range	{0, 1, 2, 3}	--	[-1,1]	--	[-1,1]	--
Head length	6	6	6	6	8	8
Number of genes	7	7	5	5	3	3
Linking function	+	+	+	+	+	+
Chromosome length	140	91	100	65	78	51
Mutation rate	0.044	0.044	0.044	0.044	0.044	0.044
One-point recombination rate	0.3	0.3	0.3	0.3	0.3	0.3
Two-point recombination rate	0.3	0.3	0.3	0.3	0.3	0.3
Gene recombination rate	0.1	0.1	0.1	0.1	0.1	0.1
IS transposition rate	0.1	0.1	0.1	0.1	0.1	0.1
IS elements length	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3
RIS transposition rate	0.1	0.1	0.1	0.1	0.1	0.1
RIS elements length	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3
Gene transposition rate	0.1	0.1	0.1	0.1	0.1	0.1
Random constants mutation rate	0.01	--	0.01	--	0.01	--
Dc specific transposition rate	0.1	--	0.1	--	0.1	--
Dc specific IS elements length	1,2,3	--	1,2,3	--	1,2,3	--
Selection range	20%	20%	100%	100%	1000%	1000%
Precision	0%	0%	0%	0%	0%	0%
Average best-of-run fitness	179.827	197.232	1914.8	1931.84	86215.27	89033.29
Average best-of-run R-square	0.977612	0.999345	0.957255	0.995340	0.713365	0.811863
Success rate	16%	81%	--	--	--	--

per run are shown in the third column of Table 2. The best solution, found in run 50 after 4584 generations, is shown below (the sub-ETs are linked by addition):

Gene 0:
```
L*L*ECaa??a??8534167
```
A_0 = {0.189, 0.13, 0.753, 0.548, 0.277,
0.257, 0.743, 0.46, 0.066, 0.801}

Gene 1:
```
~S/aC-??aa?aa5477773
```
A_1 = {0.337, 0.99, 0.536, 0.406, 0.283,
0.95, 0.968, 0.108, 0.672, 0.644}

Gene 2:
```
~*/a*aa???a?a1437777
```
A_2 = {0.247, 0.929, 0.779, 0.89, 0.926,
0.24, 0.667, 0.254, 0.518, 0.927}

Gene 3:
```
-C*?/*a?aaa??4725239
```
A_3 = {0.792, 0.019, 0.472, 0.005, 0.682,
0.605, 0.094, 0.357, 0.074, 0.713}

Gene 4:
```
+E+*EE?a?a???4233680
```
A_4 = {0.883, 0.768, 0.899, 0.311, 0.981,
0.845, 0.428, 0.308, 0.519, 0.381} (3.4)

It has a fitness of 1989.566 and an R-square of 0.9997001 evaluated over the set of 20 fitness cases and an R-square of 0.9997185 evaluated against a test set of 100 random points also chosen from the interval [-1, 1]. Mathematically, it corresponds to the following function (the contribution of each sub-ET is indicated in square brackets):

$$y = \left[\ln\left(0.99782a^2\right)\right] + \left[10^{\sin(1.27278a)}\right] + \left[10^{0.929a}\right] + \left[0.77631 - 2.80112a^3\right] + \left[2.45714 + e^{0.981a} + e^a\right]$$

which is a very good approximation to the target function (3.3) as the high value of R-square indicates.

It is worth noticing that the algorithm does in fact integrate constants in the evolved solutions, but the constants are very different from the expected ones. Indeed, GEP (and I believe, all genetic algorithms with tree representations) 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 it is possible to guess pretty accurately the function set, otherwise a very creative solution would be found.

To predict sunspots using random numerical constants F = {+, -, *, /}$_4$ and T = {a, b, c, d, e, f, g, h, i, j, ?}. The set of rational random constants R = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, and "?" ranged over the interval [-1, 1]. The parameters used per run are shown in the fifth column of Table 2. The best solution, found in run 92 after 4759 generations, is shown below (the sub-ETs are linked by addition):

Gene 0:
```
/*++j+hjjijg?cfda894833994
```
A_0 = {0.977, 0.421, 0.226, 0.325, 0.933,
0.204, 0.594, 0.8, 0.212, 0.395}

Gene 1:
```
/++b+*+ag?c?eiejb795620470
```
A_1 = {0.72, 0.447, 0.266, 0.511, 0.304,
0.247, 0.159, 0.847, 0.204, 0.995}

Gene 2:
```
/*++jj*+jii??f?ig454696802
```
A_2 = {0.52, 0.595, 0.714, 0.982, 0.987,
0.916, 0.153, 0.779, 0.987, 0.672} (3.5)

It has a fitness of 86603.2 and an R-square of 0.833714 evaluated over the set of 90 fitness cases. Mathematically, it corresponds to the following function:

$$y = \frac{2j^2}{h+i+j} + \frac{a+b+g}{0.995+0.847c+e} + \frac{1.903j+j^2}{i^2+j}$$

3.2.3. Second approach: Creation of numerical constants from scratch

To solve the sequence induction problem without the facility to manipulate numerical constants, the function set was exactly the same as in the experiment with random constants. The terminal set consisted obviously of the independent variable alone.

As shown in the second column of Table 2, the probability of success using this approach is 81%, considerably higher than the 16% obtained using the facility to manipulate random constants. In this experiment, the first perfect solution was found in generation 44 of run 0 (the sub-ETs are linked by addition):

```
0123456789012
+aa+a-aaaaaaa
*+/*/*aaaaaaa
*+++**aaaaaaa
*+***+aaaaaaa
+//++-aaaaaaa
+*---*aaaaaaa
*a-aa-aaaaaaa
```

which corresponds to the target sequence (3.2). Note that the algorithm creates all necessary constants from scratch by performing simple mathematical operations.

To find the "V" shaped function without using random constants, the function set is exactly the same as in the first approach. With this collection of functions, most of which extraneous, the algorithm is equipped with different tools for evolving highly accurate models without using numerical constants. The parameters used per run are shown in the fourth column of Table 2. In this experiment of 100 identical runs, the best solution was found in generation 4679 of run 10:

```
0123456789012
+L~*S+aaaaaaa
++a+*Saaaaaaa
+CEC*+aaaaaaa
ESaaSaaaaaaaaa
++EE/*aaaaaaa                                  (3.6)
```

It has a fitness of 1990.023 and an R-square of 0.9999313 evaluated over the set of 20 fitness cases and an R-square of 0.9998606 evaluated against the same test set used in the first approach, and thus is better than the model (3.4) evolved with the facility for the manipulation of random constants. More formally, the model (3.6) is expressed by the equation (the contribution of each gene is shown in square brackets):

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

To predict sunspots without using random numerical constants, the function set is exactly the same as in the first approach. The parameters used per run are shown in the sixth column of Table 2. In this experiment of 100 identical runs, the best solution was found in generation 2273 of run 57:

```
01234567890123456
j+a/+a+*gaafchdci
/++-+be+ijdjjaiid
/++*ci+-jiabiddhf                              (3.7)
```

It has a fitness of 89176.61 and an R-square of 0.882831 evaluated over the set of 90 fitness cases, and thus is better than the model (3.5) evolved with the facility for the manipulation of random constants. More formally, the model (3.7) is expressed by the equation:

$$y = j + \frac{d - i + 3j}{b + e} + \frac{d + bj - ij}{a + 2i}$$

It is instructive to compare the results obtained in both approaches. In all the experiments the explicit use of random constants resulted in a worse performance. In the sequence induction problem, success rates of 81% against 16% were obtained; in the "V" function problem average best-of-run fitnesses of 1931.84 versus 1914.80 and average best-of-run R-squares of 0.995340 versus 0.957255 were obtained; and in the sunspots prediction problem average best-of-run fitnesses of 89033.29 versus 86215.27 and average best-of-run R-squares of 0.811863 versus 0.713365 were obtained (see Table 2). Thus, in real-world applications where complex realities are modeled, of which nothing is known concerning neither the type nor the range of the numerical constants, and where most of the times it is impossible 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. Not only the results will be better but also the complexity of the system will be much smaller.

4. Conclusions

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. In practical terms, the crossing of the phenotype threshold allows the unconstrained exploration of the search space because all modifications are made on the genome and because all modifications always result in valid phenotypes or programs. In addition, the genotype/phenotype representation of GEP not only simplifies but also invites the creation of more complexity. The elegant mechanism developed to deal with random constants is a good example of this.

In this work, the question of constant creation in symbolic regression was discussed comparing two different approaches to solve this problem: one with the explicit use of numerical constants, and another without them. The results presented here suggest that the latter is more efficient, not only in terms of the accuracy of the best evolved models and overall performance, but also because the search space is much smaller, reducing greatly the complexity of the system and, consequently, the precious CPU time.

Finally, the results presented in this work also suggest that, apparently, the term "constant" is just another word for mathematical expression and that evolutionary algorithms are particularly good at finding these expressions because the search is totally unbiased.

Bibliography

Banzhaf, W., Genotype-Phenotype-Mapping and Neutral Variation – A Case Study in Genetic Programming. In Y. Davidor, H.-P. Schwefel, and R. Männer, eds., *Parallel Problem Solving from Nature III*, *Lecture Notes in Computer Science*, 866: 322-332, Springer-Verlag, 1994.

Banzhaf, W., P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, 1998.

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.

Ferreira, C., 2001. Gene Expression Programming: A New Adaptive Algorithm for Solving Problems. *Complex Systems*, 13 (2): 87-129.

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).

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

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

Weigend, A.S., B. A. Huberman, and D. E. Rumelhart, Predicting Sunspots and Exchange Rates with Connectionist Networks. In S. Eubank and M. Casdagli, eds., *Nonlinear Modeling and Forecasting*, pages 395-432, Redwood City, CA, Addison-Wesley, 1992.