AUTOMATICALLY DEFINED FUNCTIONS IN GENE EXPRESSION PROGRAMMING

CÂNDIDA FERREIRA

Gepsoft, 73 Elmtree Drive,
Bristol BS13 8NA, UK
candidaf@gepsoft.com
http://www.gene-expression-programming.com/author.asp

In N. Nedjah, L. de M. Mourelle, A. Abraham, eds., Genetic Systems Programming: Theory and Experiences, Studies in Computational Intelligence, Vol. 13, pp. 21-56, Springer-Verlag, 2006.

In this chapter it is shown how Automatically Defined Functions are encoded in the genotype/phenotype system of Gene Expression Programming. As an introduction, the fundamental differences between Gene Expression Programming and its predecessors, Genetic Algorithms and Genetic Programming, are briefly summarized so that the evolutionary advantages of Gene Expression Programming are better understood. The introduction proceeds with a detailed description of the architecture of the main players of Gene Expression Programming (chromosomes and expression trees), focusing mainly on the interactions between them and how the simple, yet revolutionary, structure of the chromosomes allows the efficient, unconstrained exploration of the search space. The work proceeds with an introduction to Automatically Defined Functions and how they are implemented in Gene Expression Programming. Furthermore, the importance of Automatically Defined Functions in Evolutionary Computation is thoroughly analyzed by comparing the performance of sophisticated learning systems with Automatically Defined Functions with much simpler ones on the sextic polynomial problem.

1. Genetic Algorithms: Historical Background

The way nature solves problems and creates complexity has inspired scientists to create artificial systems that learn how to solve a particular problem without human intervention. The first attempts were done in the 1950s by Friedberg (Friedberg 1958; Friedberg *et al.* 1959), but ever since highly sophisticated systems have been developed that apply Darwin's ideas of natural evolution to the artificial world of computers and modeling. Of particular interest to this work are the Genetic Algorithms (GAs) and the Genetic Programming (GP) technique as they are the predecessors of Gene Expression Programming (GEP), an extremely versatile genotype/phenotype system. The way Automatically Defined Functions (ADFs) are implemented in GEP is another example of the great versatility of this algorithm and the versatility of GEP ADFs opens up new grounds for the creation of even more sophisticated artificial learning systems. So let's start by introducing briefly these three techniques in order to appreciate the versatility of the genotype/phenotype system of Gene Expression Programming with and without ADFs.

1.1. Genetic Algorithms

Genetic Algorithms were invented by John Holland in the 1960s and they also apply biological evolution theory to computer systems (Holland 1975). And like all evolutionary computer systems, GAs are an oversimplification of biological evolution. In this case, solutions to a problem are usually encoded in fixed length strings of 0's and 1's (chromosomes), and populations of such strings (individuals or candidate solutions) are manipulated in order to evolve a good solution to a particular problem. From generation to generation individuals are reproduced with modification and selected according to fitness. Modification in the original genetic algorithm was introduced by the search operators of mutation, crossover, and inversion, but more recent applications started favoring mutation and crossover, dropping inversion in the process.

It is worth pointing out that GAs' individuals consist of naked chromosomes or, in other words, GAs' individuals are simple replicators. And like all simple replicators, the chromosomes of GAs work both as genotype and phenotype. This means that they are simultaneously the objects of selection and the guardians of the genetic information that must be replicated and passed on with modification to the next generation. Consequently, the whole structure of the replicator determines the functionality and, consequently, the fitness of the individual. For instance, in such systems it is not possible to use only a particular region of the replicator as a solution to a problem; the whole replicator is always the solution: nothing more, nothing less.

1.2. Genetic Programming

Genetic Programming, invented by Cramer in 1985 (Cramer 1985) and further developed by Koza (1992), finds an alternative to fixed length solutions through the introduction of nonlinear structures (parse trees) with different sizes and shapes. The alphabet used to create these structures is also more varied than the 0's and 1's of GAs' individuals, creating a richer, more versatile system of representation. Notwithstanding, GP individuals also lack a simple, autonomous genome: like the linear chromosomes of GAs, the nonlinear structures of GP are also naked replicators cursed with the dual role of genotype/phenotype.

It is worth noticing that the parse trees of GP resemble protein molecules in their use of a richer alphabet and in their complex and unique hierarchical representation. Indeed, parse trees are capable of exhibiting a great variety of functionalities. The problem with these complex replicators is that their reproduction with modification is highly constrained in evolutionary terms, simply because the modifications must take place on the parse tree itself and, consequently, only a limited range of modification is possible. Indeed, the genetic operators of GP operate at the tree level, modifying or exchanging particular branches between trees.

Although at first sight this might appear advantageous, it greatly limits the GP technique (we all know the limits of grafting and pruning in nature). Consider, for instance, crossover, the most used and often the only search operator used in GP (Figure 1). In this case,

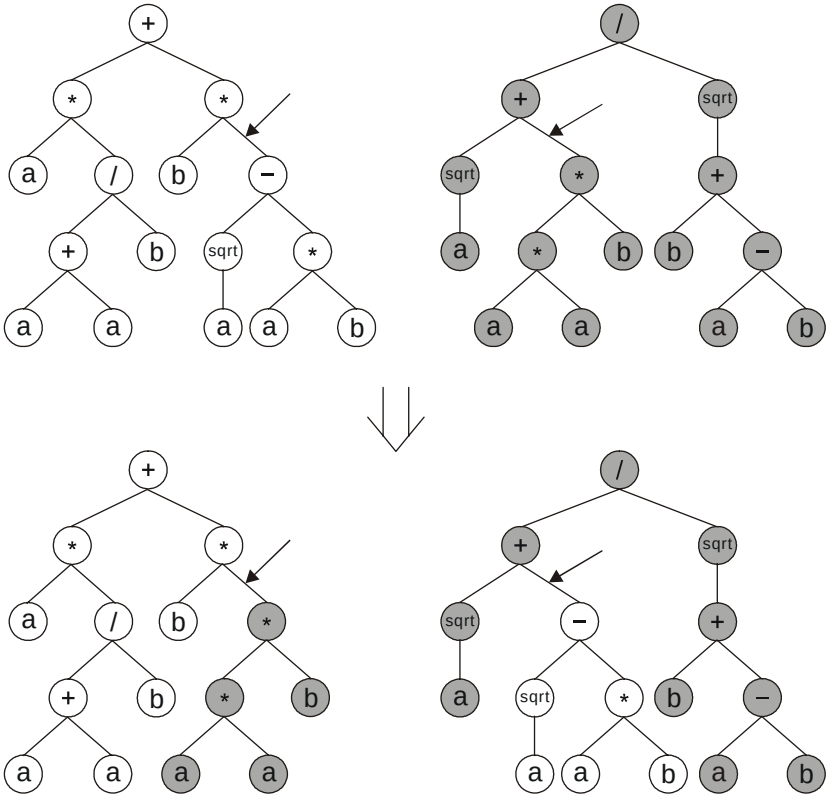


Figure 1. Tree crossover in Genetic Programming. The arrows indicate the crossover points.

selected branches are exchanged between two parent 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 simpler building blocks, guaranteeing, at the same time, the creation of syntactically correct structures.

The mutation operator in GP is also very different from natural point mutation. This operator selects a node in the parse tree and replaces the branch underneath by a new randomly generated branch (Figure 2). Notice that the overall shape of the tree is not greatly changed by this kind of mutation, especially if lower nodes are preferentially chosen as mutation targets.

Permutation is the third operator used in Genetic Programming and the most conservative of the three. During permutation, the arguments of a randomly chosen function are randomly permuted (Figure 3). In this case the overall shape of the tree remains unchanged.

In summary, in Genetic Programming the operators resemble more of a conscious mathematician than the blind way of nature. But in adaptive systems the blind way of nature is

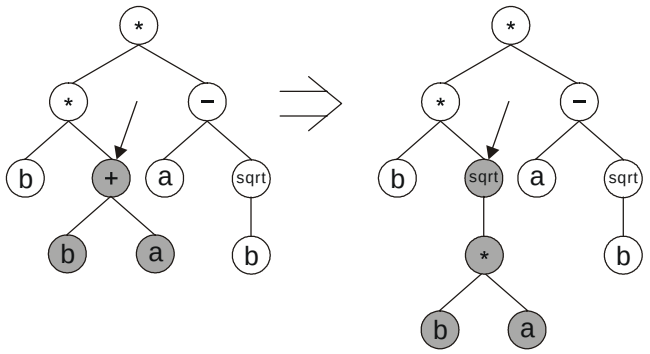


Figure 2. Tree mutation in Genetic Programming. The arrow indicates the mutation point. The new branch randomly generated by the mutation operator in the daughter tree is shown in gray.

much more efficient and systems such as GP are highly limited in evolutionary terms. For instance, the implementation of other operators in GP, such as the simple yet high-performing point mutation (Ferreira 2002c), is unproductive as most mutations would have resulted in syntactically incorrect structures (Figure 4). Obviously, the implementation of other operators such as transposition or inversion raises similar difficulties and the search space in GP remains vastly unexplored.

Although Koza described these three operators as the basic GP operators, crossover is practically the only genetic operator used in most GP applications (Koza 1992, 1994; Koza *et al*. 1999). Consequently, no new genetic material is introduced in the genetic pool of GP populations. Not surprisingly, huge populations of parse trees must be used in order to prime the initial population with all the necessary building blocks so that good solutions could be discovered by just moving these initial building blocks around.

Finally, due to the dual role of the parse trees (genotype and phenotype), Genetic Programming is also incapable of a simple, rudimentary expression; in all cases, the entire parse tree is the solution: nothing more, nothing less.

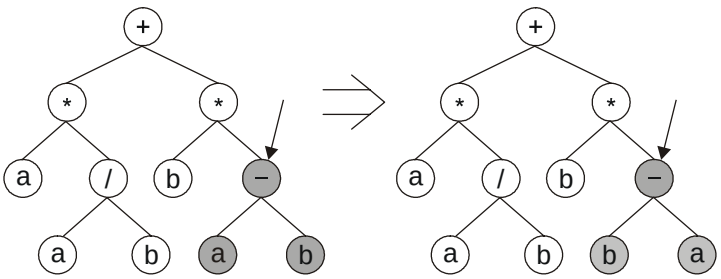


Figure 3. Permutation in Genetic Programming. The arrow indicates the permutation point. Note that the arguments of the permuted function traded places in the daughter tree.

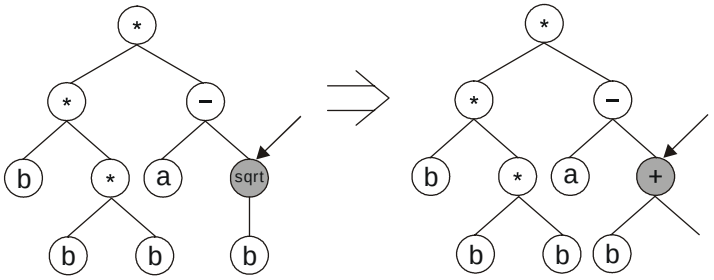


Figure 4. Illustration of a hypothetical event of point mutation in Genetic Programming. The arrow indicates the mutation point. Note that the daughter tree is an invalid structure.

1.3. Gene Expression Programming

Gene Expression Programming was invented by myself in 1999 (Ferreira 2001), and incorporates both the simple, linear chromosomes of fixed length similar to the ones used in Genetic Algorithms and the ramified structures of different sizes and shapes similar to the parse trees of Genetic Programming. And since the ramified structures of different sizes and shapes are totally encoded in the linear chromosomes of fixed length, this is equivalent to say that, in GEP, the genotype and phenotype are finally separated from one another and the system can now benefit from all the evolutionary advantages this brings about.

Thus, the phenotype of GEP consists of the same kind of ramified structure used in GP. But the ramified structures evolved by GEP (called expression trees) are the expression of a totally autonomous genome. Therefore, with GEP, a remarkable thing happened: the second evolutionary threshold – the phenotype threshold – was crossed (Dawkins 1995). And this means that only the genome (slightly modified) is passed on to the next generation. Consequently, one no longer needs to replicate and mutate rather cumbersome structures as all the modifications take place in a simple linear structure which only later will grow into an expression tree.

The fundamental steps of Gene Expression Programming are schematically represented in Figure 5. The process begins with the random generation of the chromosomes of a certain number of individuals (the initial population). Then these chromosomes are expressed and the fitness of each individual is evaluated against a set of fitness cases (also called selection environment). The individuals are then selected according to their fitness (their performance in that particular environment) to reproduce with modification, leaving progeny with new traits. These new individuals are, in their turn, subjected to the same developmental process: expression of the genomes, confrontation of the selection environment, selection according to fitness, and reproduction with modification. The process is repeated for a certain number of generations or until a good solution has been found.

So, the pivotal insight of Gene Expression Programming consisted in the invention of chromosomes capable of representing any parse tree. For that purpose a new language –

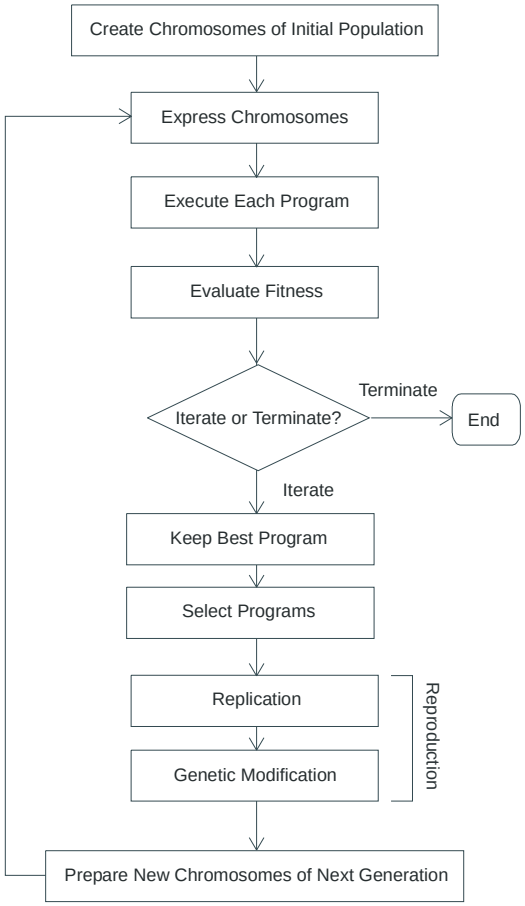


Figure 5. The flowchart of Gene Expression Programming.

Karva language – was created in order to read and express the information encoded in the chromosomes. The details of this new language are given in the next section.

Furthermore, the structure of the chromosomes was designed in order to allow the creation of multiple genes, each coding for a smaller program or sub-expression tree. It is worth emphasizing that Gene Expression Programming is the only genetic algorithm with multiple genes. Indeed, in truly functional genotype/phenotype systems, the creation of more complex individuals composed of multiple genes is a child's play, and illustrates quite well the great versatility of the GEP system. In fact, after their inception, these systems seem to catapult themselves into higher levels of complexity such as the uni- and multicellular systems, where different cells put together different combinations of genes (Ferreira 2002a). We will see later in this chapter how the cellular system of GEP is an extremely elegant

way of implementing Automatically Defined Functions that may be reused by the created programs.

The basis for all this novelty resides on the simple, yet revolutionary structure of GEP genes. This structure not only allows the encoding of any conceivable program but also allows an efficient evolution. This versatile structural organization also allows the implementation of a very powerful set of genetic operators which can then very efficiently search the solution space. As in nature, the search operators of GEP always generate valid structures and therefore are remarkably suited to creating genetic diversity.

2. The Architecture of GEP Individuals

We know already that the main players in Gene Expression Programming are the chromosomes and the expression trees (ETs), and that the latter are the expression of the genetic information encoded in the former. As in nature, the process of information decoding is called translation. And this translation implies obviously a kind of code and a set of rules. The genetic code is very simple: a one-to-one relationship between the symbols of the chromosome and the nodes they represent in the trees. The rules are also very simple: they determine the spatial organization of nodes in the expression trees and the type of interaction between sub-ETs. Therefore, there are two languages in GEP: the language of the genes and the language of expression trees and, thanks to the simple rules that determine the structure of ETs and their interactions, we will see that it is possible to infer immediately the phenotype given the sequence of a gene, and vice versa. This means that we can choose to have a very complex program represented by its compact genome without losing any information. This unequivocal bilingual notation is called *Karva* language. Its details are explained in the remainder of this section.

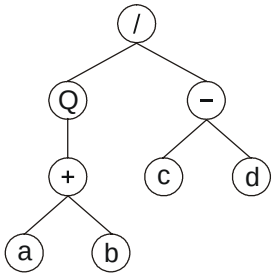
2.1. Open Reading Frames and Genes

The structural organization of GEP genes is better understood in terms of open reading frames (ORFs). In biology, an ORF or coding sequence of a gene begins with the start codon, continues with the amino acid codons, and ends at a termination codon. However, a gene is more than the respective ORF, with sequences upstream of the start codon and sequences downstream of the stop codon. Although in GEP the start site is always the first position of a gene, the termination point does not always coincide with the last position of a gene. Consequently, it is common for GEP genes to have noncoding regions downstream of the termination point. (For now we will not consider these noncoding regions, as they do not interfere with expression.)

Consider, for example, the algebraic expression:

$$\frac{\sqrt{a+b}}{c-d} \tag{1}$$

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. And the genotype can be easily inferred from the phenotype as follows:

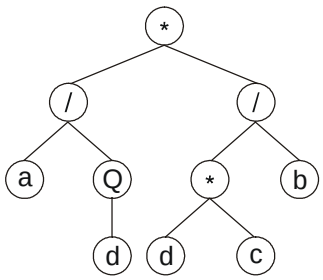
```
01234567
/Q-+cdab                                                    (2)
```

which is the straightforward reading of the ET from left to right and from top to bottom (exactly as one reads a page of text). The expression (2) is an open reading frame, starting at "/" (position 0) and terminating at "b" (position 7). These ORFs were named *K-expressions* from Karva language.

Consider another ORF, the following K-expression:

```
0123456789
*//aQ*bddc                                                  (3)
```

Its expression as an ET is also very simple and straightforward. In order to express the ORF correctly, 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 expression tree, and it occupies the topmost position (or first line) on the tree. Second, in the next line, below each function, are placed as many branch nodes as there are arguments to that function. Third, from left to right, the nodes are filled consecutively with the next elements of the K-expression. Fourth, the process is repeated until a line containing only terminals is formed. In this case, the following expression tree is formed:



Looking at the structure of 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 open reading frames are analyzed in the context of a gene, the advantages of this representation become obvious. As I said before, GEP chromosomes have fixed length, and they are composed of one or more genes of equal length. Consequently, the length of a gene is also fixed. Thus, in GEP, what changes is not the length of genes, but rather the length of the ORF. 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 latter, the termination point is somewhere upstream of the end of the gene. And this obviously means that GEP genes have, most of the time, noncoding regions at their ends.

And what is the function of these noncoding regions at the end of GEP genes? We will see that they are the essence of Gene Expression Programming and evolvability, because they allow the modification of the genome using all kinds of genetic operators without any kind of restriction, always producing syntactically correct programs. Thus, in GEP, the fundamental property of genotype/phenotype systems – syntactic closure – is intrinsic, allowing the totally unconstrained restructuring of the genotype and, consequently, an efficient evolution.

In the next section we are going to analyze the structural organization of GEP genes in order to understand how they invariably code for syntactically correct programs and why they allow the unconstrained application of virtually any genetic operator.

2.2. Structural Organization of Genes

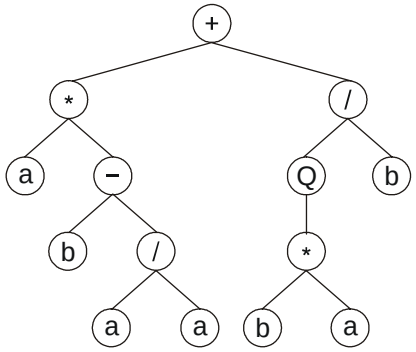
The novelty of GEP genes resides in the fact that they are composed of a head and a tail. The head contains symbols that represent both functions and terminals, whereas the tail contains only terminals. For each problem, the length of the head h is chosen, whereas the length of the tail t is a function of h and the number of arguments n of the function with more arguments (also called maximum arity) and is evaluated by the equation:

$$t = h\,(n\text{-}1) + 1 \qquad (4)$$

Consider a gene for which the set of terminals T = {a, b} and the set of functions F = {Q, *, /, -, +}, thus giving $n = 2$. And if we chose an $h = 10$, then $t = 10\,(2 - 1) + 1 = 11$ and the length of the gene g is 10 + 11 = 21. One such gene is shown below (the tail is shown in bold):

```
012345678901234567890
+*/a-Qbb/*aabaabbabaa                                    (5)
```

It codes for the following expression tree:

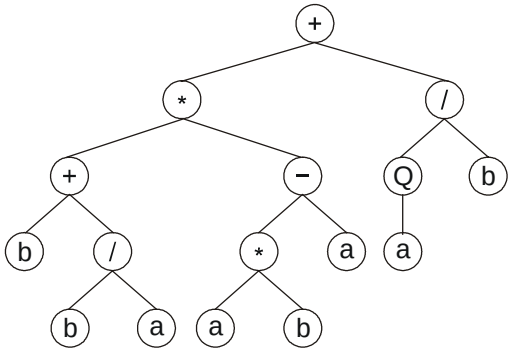


Note that, in this case, the open reading frame ends at position 13, whereas the gene ends at position 20.

Suppose now a mutation occurred at position 3, changing the "a" into "+". Then the following gene is obtained:

```
01234567890123456789
+*/+-Qbb/*aabaabbabaa                                    (6)
```

And its expression gives:

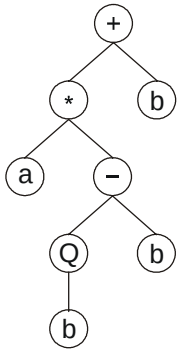


In this case, the termination point shifts two positions to the right (position 15), enlarging and changing significantly the daughter tree.

Obviously the opposite might also happen, and the daughter tree might shrink. For example, consider again gene (5) above, and suppose a mutation occurred at position 2, changing the "/" into "b":

```
01234567890123456789
+*ba-Qbb/*aabaabbabaa                                    (7)
```

And now its expression results in the following ET:



In this case, the ORF ends at position 7, shortening the original ET by six nodes.

So, despite their fixed length, the genes of Gene Expression Programming have the potential to code for expression trees of different sizes and shapes, where the simplest is composed of only one node (when the first element of a gene is a terminal) and the largest is 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 program. Consequently, the implementation of a powerful set of search operators, such as point mutation, inversion, transposition, and recombination, is a child's play, making Gene Expression Programming the ideal playground for the discovery of the perfect solution using an economy of resources (see Ferreira 2001 and 2004a for a detailed description of the mechanisms and effects of the different genetic operators commonly used in Gene Expression Programming).

2.3. Multigenic Chromosomes and Linking Functions

The chromosomes of Gene Expression Programming 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, in problems with just one output, the sub-ETs interact with one another forming a more complex multi-subunit ET; in problems with multiple outputs, though, each sub-ET evolves its respective output.

Consider, for example, the following chromosome with length 39, composed of three genes, each with length 13 (the tails are shown in bold):

```
0123456789012012345678901201234567890 12
*Q-b/abbbaaba/aQb-bbbaabaa*Q-/b*abbbbaa                (8)
```

It has three open reading frames, and each ORF codes for a sub-ET (Figure 6). We know already that the start of each ORF coincides with the first element of the gene and, for the sake of clarity, for each gene it is always indicated by position zero; the end of each ORF, though, is only evident upon construction of the corresponding sub-ET. As you can see in

Figure 6, the first open reading frame ends at position 7; the second ORF ends at position 3; and the last ORF ends at position 9. Thus, GEP chromosomes contain several ORFs of different sizes, 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 outputs, or they may form a more complex, multi-subunit expression tree and be selected as a whole. In these multi-subunit structures, individual sub-ETs interact with one another by a particular kind of posttranslational interaction or linking. For instance, algebraic sub-ETs can be linked by addition or subtraction whereas Boolean sub-ETs can be linked by OR or AND.

a. `012345678901201234567890120123456789012`
`*Q-b/a`**`bbbaaba`**`/aQb-b`**`bbaabaa`**`*Q-/b*`**`abbbbaa`**

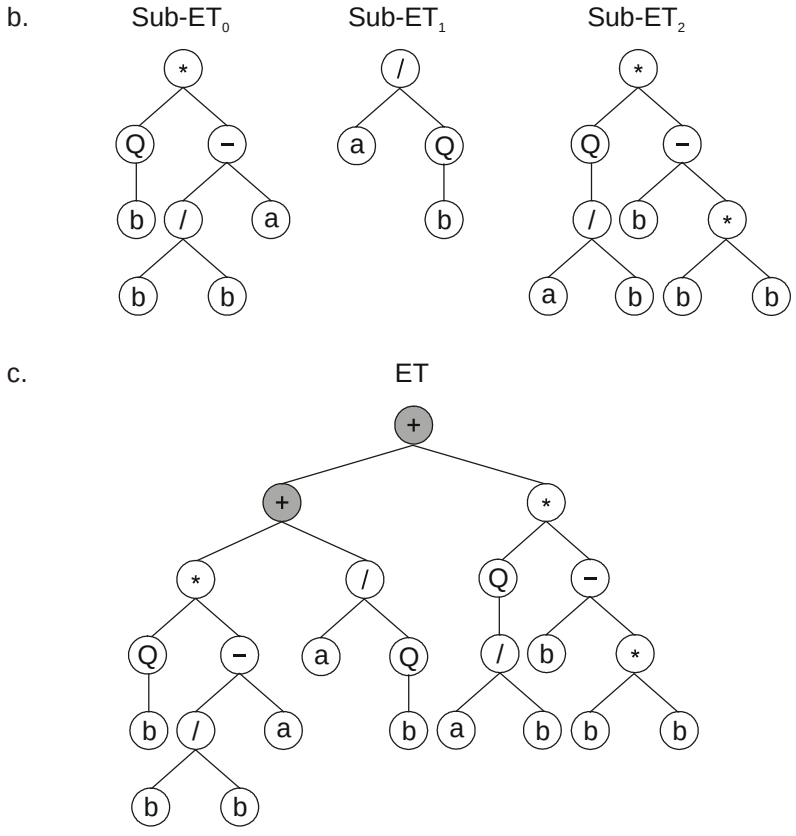


Figure 6. Expression of GEP genes as sub-ETs. **a)** A three-genic chromosome with the tails shown in bold. Position zero marks the start of each gene. **b)** The sub-ETs codified by each gene. **c)** The result of posttranlational linking with addition. The linking functions are shown in gray.

The linking of three sub-ETs by addition is illustrated in Figure 6, c. Note that the final ET could be linearly encoded as the following K-expression:

```
012345678901234567890123
++**/Q-Q-aQ/b*b/ababbbbb                                    (9)
```

However, the use of multigenic chromosomes is more appropriate to evolve good solutions to complex problems, for they permit the modular construction of complex, hierarchical structures, where each gene codes for a smaller and simpler building block. These building blocks are physically separated from one another, and thus can evolve independently. Not surprisingly, these multigenic systems are much more efficient than unigenic ones (Ferreira 2001, 2002a). And, of course, they also open up new grounds to solve problems of multiple outputs, such as parameter optimization or classification problems (Ferreira 2002a).

3. Chromosome Domains and Random Numerical Constants

We have already met two different domains in GEP genes: the head and the tail. And now another one – the Dc domain – will be introduced. This domain was especially designed to handle random numerical constants and consists of an extremely elegant, efficient, and original way of dealing with them.

As an example, Genetic Programming handles random numerical constants 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. Note, however, that with this method no new constants are created during evolution.

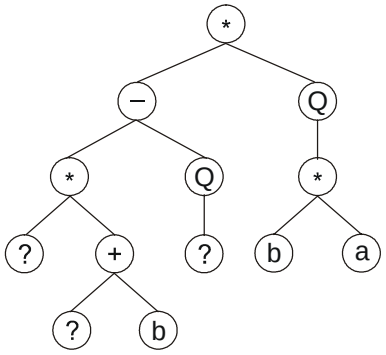
Gene Expression Programming handles random numerical constants differently (Ferreira 2001). GEP uses an extra terminal "?" and an extra domain Dc composed of the symbols chosen to represent the random constants. The values of each random constant, though, are only assigned during gene expression. Thus, for each gene, the initial random constants are generated during the creation of the initial population and kept in an array. However, 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 (mutation, inversion, transposition, and recombination), plus a Dc-specific transposition and a Dc-specific inversion, guarantee the effective circulation of the random constants in the population. Indeed, with this scheme of constants' manipulation, the appropriate diversity of numerical constants is generated at the beginning of a run and maintained easily afterwards by the genetic operators. Let's take then a closer look at the structural organization of this extra domain and how it interacts with the sub-ET encoded in the head/tail structure.

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.

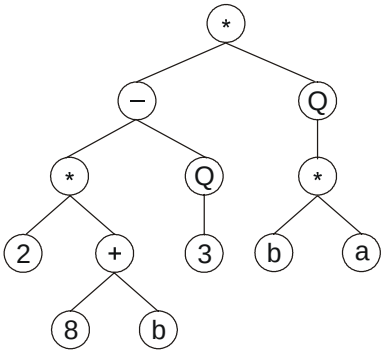
Consider the single-gene chromosome with an $h = 8$ (the Dc is shown in bold):

```
012345678901234567890123 45
*-Q*Q*?+?ba?babba238024198                    (10)
```

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



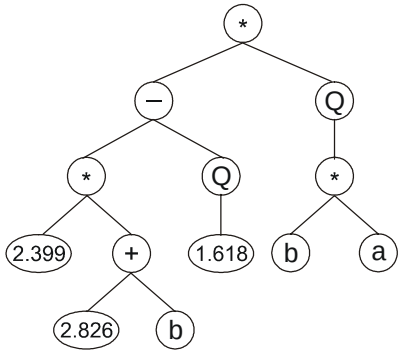
Then the ?'s in the expression tree are replaced from left to right and from top to bottom by the symbols (numerals, for simplicity) in Dc, obtaining:



The random constants corresponding to these symbols are kept in an array and, for simplicity, the number represented by the numeral indicates the order in the array. For instance, for the following array of 10 elements:

A = {1.095, 1.816, 2.399, 1.618, 0.725, 1.997, 0.094, 2.998, 2.826, 2.057}

the chromosome (10) above gives:



This type of domain was used to great advantage not only in symbolic regression but also in parameter optimization and polynomial induction (Ferreira 2002a). But this elegant structure can also be used to evolve the weights and thresholds of evolvable neural networks (Ferreira 2004b) and to encode decision trees with numerical attributes (unpublished material). And we will see here for the first time that this kind of domain can also be used to create Automatically Defined Functions with random numerical constants.

4. Cells and the Creation of Automatically Defined Functions

Automatically Defined Functions were for the first time introduced by Koza as a way of reusing code in Genetic Programming (Koza 1992).

The ADFs of GP obey a rigid syntax in which an S-expression, with a LIST-*n* function on the root, lists *n*-1 function-defining branches and one value-returning branch (Figure 7). The function-defining branches are used to create ADFs that may or may not be called upon by the value-returning branch. Such rigid structure imposes great constraints on the genetic operators as the different branches of the LIST function are not allowed to exchange genetic material amongst themselves. Furthermore, the ADFs of Genetic Programming are further constrained by the number of arguments each takes, as the number of arguments must be a priori defined and cannot be changed during evolution.

In the multigenic system of Gene Expression Programming, the implementation of ADFs can be done elegantly and without any kind of constraints as each gene is used to encode a

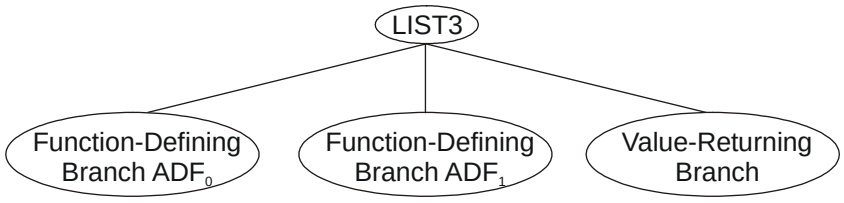


Figure 7. The overall structure of an S-expression with two function-defining branches and the value-returning branch used in Genetic Programming to create Automatically Defined Functions.

different ADF (Ferreira 2002a). The way these ADFs interact with one another and how often they are called upon is encoded in special genes – homeotic genes – thus called because they are the ones controlling the overall development of the individual. And continuing with the biological analogy, the product of expression of such genes is also called a cell. Thus, homeotic genes determine which genes are expressed in which cell and how they interact with one another. Or stated differently, homeotic genes determine which ADFs are called upon in which main program and how they interact with one another. How this is done is explained in the remainder of this section.

4.1. Homeotic Genes and the Cellular System of GEP

Homeotic genes have exactly the same kind of structure as conventional genes and are built using an identical process. They also contain a head and a tail domain, with the heads containing, in this case, linking functions (so called because they are actually used to link different ADFs) and a special class of terminals – genic terminals – representing conventional genes, which, in the cellular system, encode different ADFs; the tails contain obviously only genic terminals.

Consider, for instance, the following chromosome:

```
01234567801234567801234567801234567890
/-b/abbaa*a-/abbab-*+abbbaa**Q2+010102                    (11)
```

It codes for three conventional genes and one homeotic gene (shown in bold). The conventional genes encode, as usual, three different sub-ETs, with the difference that now these sub-ETs will act as ADFs and, therefore, may be invoked multiple times from different places. And the homeotic gene controls the interactions between the different ADFs (Figure 8). As you can see in Figure 8, in this particular case, ADF_0 is used twice in the main program, whereas ADF_1 and ADF_2 are both used just once.

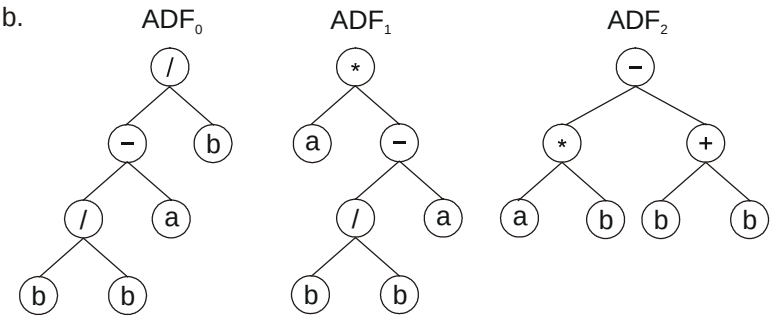
It is worth pointing out that homeotic genes have their specific length and their specific set of functions. And these functions can take any number of arguments (functions with 1, 2, 3, …, *n*, arguments). For instance, in the particular case of chromosome (11), the head length of the homeotic gene h_H is equal to five, whereas for the conventional genes $h = 4$; the function set used in the homeotic gene F_H consists of $F_H = \{+, -, *, /, Q\}$, whereas for the conventional genes the function set consists of $F = \{+, -, *, /\}$. As shown in Figure 8, this cellular system is not only a form of elegantly allowing the evolution of linking functions in multigenic systems but also an extremely elegant way of encoding ADFs that can be called an arbitrary number of times from an arbitrary number of different places.

4.2. Multicellular Systems

The use of more than one homeotic gene results obviously in a multicellular system, in which each homeotic gene puts together a different consortium of genes.

a.
```
012345678012345678012345678012345678901234567890
/-b/abbaa*a-/abbab-*+abbbaa**Q2+010102
```

b.



c.

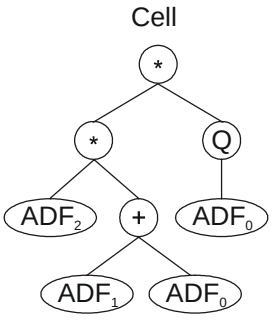


Figure 8. Expression of a unicellular system with three Automatically Defined Functions. **a)** The chromosome composed of three conventional genes and one homeotic gene (shown in bold). **b)** The ADFs codified by each conventional gene. **c)** The main program or cell.

Consider, for instance, the following chromosome:

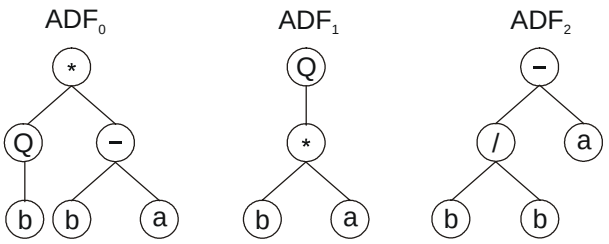
```
0123456012345601234560123456780123456 78
*Q-bbabQ*baabb-/abbab**+21Q1102/*21+1011                (12)
```

It codes for three conventional genes and two homeotic genes (shown in bold). And its expression results in two different cells or programs, each expressing different genes in different ways (Figure 9). As you can see in Figure 9, ADF_1 is used twice in both cells; ADF_2 is used just once in both cells; and ADF_0 is only used in $Cell_1$.

The applications of these multicellular systems are multiple and varied and, like the multigenic systems, they can be used both in problems with just one output and in problems with multiple outputs. In the former case, the best program or cell accounts for the fitness of the individual; in the latter, each cell is responsible for a particular facet in a multiple output task such as a classification task with multiple classes.

a. 012345601234560123456012345678012345678
*Q-bbabQ*baabb-/abbab**+21Q1102/*21+1011**

b.



c.

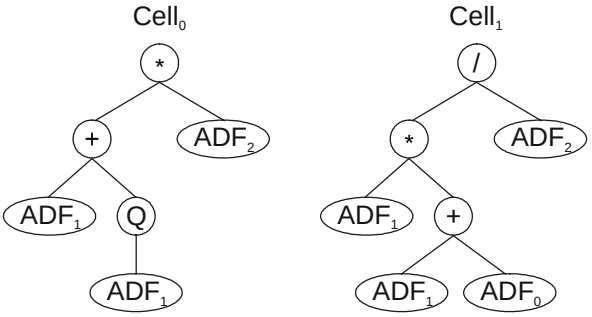


Figure 9. Expression of a multicellular system with three Automatically Defined Functions. **a)** The chromosome composed of three conventional genes and two homeotic genes (shown in bold). **b)** The ADFs codified by each conventional gene. **c)** Two different main programs expressed in two different cells. Note how different cells put together different combinations of ADFs.

It is worth pointing out that the implementation of multiple main programs in Genetic Programming is virtually unthinkable and so far no one has attempted it.

4.3. Incorporating Random Numerical Constants in ADFs

The incorporation of random numerical constants in Automatically Defined Functions is also easy and straightforward. As you probably guessed, the gene structure used to accomplish this includes the special domain Dc for encoding the random numerical constants, which, for the sake of simplicity and efficiency, is only implemented in the genes encoding the ADFs (one can obviously extend this organization to the homeotic genes, but nothing is gained from that except a considerable increase in computational effort). The structure of the homeotic genes remains exactly the same and they continue to control how often each ADF is called upon and how they interact with one another.

Consider, for instance, the chromosome with two conventional genes and their respective arrays of random numerical constants:

```
01234567890012345678900123456780123456 78
**?b?aa4701+/Q?ba?8536*0Q-10010/Q-+01111                (13)
```

A_0 = {0.664, 1.703, 1.958, 1.178, 1.258, 2.903, 2.677, 1.761, 0.923, 0.796}
A_1 = {0.588, 2.114, 0.510, 2.359, 1.355, 0.186, 0.070, 2.620, 2.374, 1.710}

The genes encoding the ADFs are expressed exactly as normal genes with a Dc domain and, therefore, their respective ADFs will, most probably, include random numerical constants (Figure 10). Then these ADFs with random numerical constants are called upon as many times as necessary from any of the main programs encoded in the homeotic genes. As you can see in Figure 10, ADF_0 is invoked twice in $Cell_0$ and once in $Cell_1$, whereas ADF_1 is used just once in $Cell_0$ and called three different times in $Cell_1$.

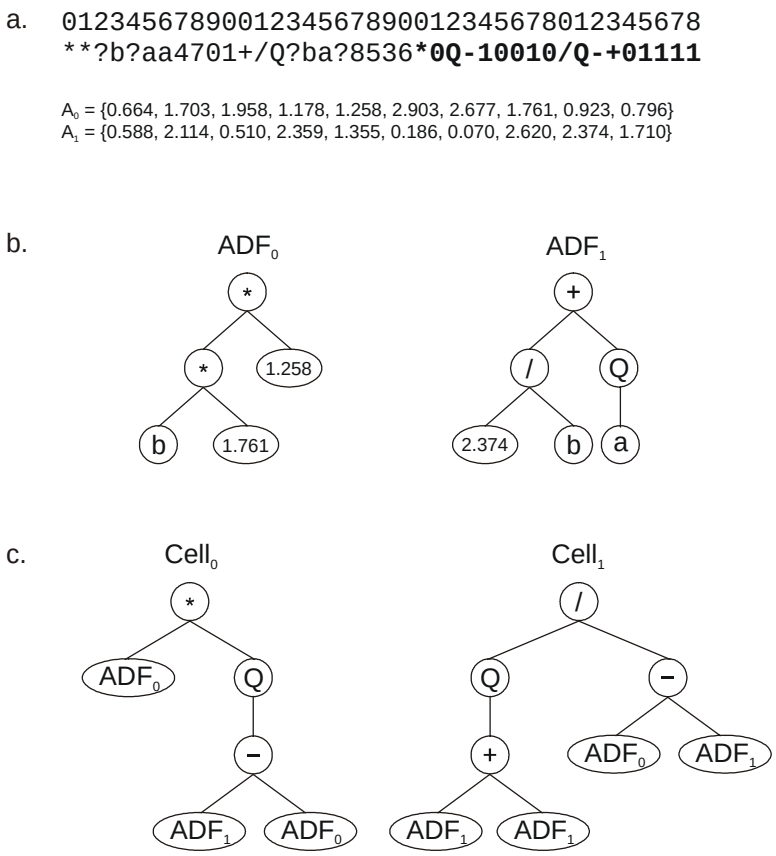


Figure 10. Expression of a multicellular system with Automatically Defined Functions containing random numerical constants. **a)** The chromosome composed of two conventional genes and two homeotic genes (shown in bold). **b)** The ADFs codified by each conventional gene. **c)** Two different programs expressed in two different cells.

5. Analyzing the Importance of ADFs in Automatic Programming

The motivation behind the implementation of Automatically Defined Functions in Genetic Programming, was the belief that ADFs allow the evolution of modular solutions and, consequently, improve the performance of the GP technique (Koza 1992, 1994; Koza *et al*. 1999). Koza proved this by solving a sextic polynomial problem and the even-parity functions, both with and without ADFs (Koza 1994).

In this section, we are going to solve the sextic polynomial problem using not only a cellular system with ADFs but also a multigenic system with static linking and a simple unigenic system. The study of the simple unigenic system is particularly interesting because it has some similarities with the GP system without ADFs.

5.1. General Settings

The sextic polynomial of this section $x^6-2x^4+x^2$ was chosen by Koza (Koza 1994) because of its potentially exploitable regularity and modularity, easily guessed by its factorization:

$$x^6-2x^4+x^2 = x^2(x-1)^2(x+1)^2 \tag{14}$$

For this problem, the basic parameters common to both GEP and GP, irrespective of the presence or absence of ADFs, were kept exactly the same as those used by Koza. Thus, a set of 50 random fitness cases chosen from the interval [-1.0, +1.0] was used; and a very simple function set, composed only of the four arithmetic operators F = {+, -, *, /} was used. As for random numerical constants, we will see that their use in this problem is not crucial for the evolution of perfect solutions. Nonetheless, evolution still goes smoothly if integer constants are used and, therefore, one can illustrate the role random constants play and how they are integrated in Automatically Defined Functions by choosing integer random constants. So, when used, integer random constants are chosen from the interval [0, 3].

The fitness function used to evaluate the performance of each evolved program is based on the relative error and explores the idea of a selection range and a precision. The selection range is used as a limit for selection to operate, above which the performance of a program on a particular fitness case contributes nothing to its fitness. And the precision is the limit for improvement, as it allows the fine-tuning of the evolved solutions as accurately as necessary.

Mathematically, the fitness f_i of an individual program i is expressed by the equation:

$$f_i = \sum_{j=1}^{n}\left(R - \left|\frac{P_{(ij)} - T_j}{T_j}\cdot 100\right|\right) \tag{15}$$

where R is the selection range, $P_{(ij)}$ the value predicted by the individual program i for fit-

ness case j (out of n fitness cases) and T_j is the target value for fitness case j. Note that the absolute value term corresponds to the relative error. This term is what is called the precision and if the error is lower than or equal to the precision then the error becomes zero. Thus, for a perfect fit, the absolute value term is zero and $f_i = f_{max} = nR$. In all the experiments of this work we are going to use a selection range of 100 and a precision of 0.01, thus giving for the set of 50 fitness cases $f_{max} = 5{,}000$. It is worth pointing out that these conditions, especially the precision of 0.01, guarantee that all perfect solutions are indeed perfect and match exactly the target function (14). This is important to keep in mind since the performance of the different systems will be compared in terms of success rate.

5.2. Results without ADFs

The importance of Automatically Defined Functions and the advantages they bring to Evolutionary Computation can only be understood if one analyzes their behavior and how the algorithm copes with their integration. Is evolution still smooth? Are there gains in performance? Is the system still simple or excessively complicated? How does it compare to simpler systems without ADFs? How does one integrate random numerical constants in ADFs? Are these ADFs still manageable or excessively complex? Are there problems that can only be solved with ADFs? These are some of the questions that we will try to address in this work. And for that we are going to start this study by analyzing two simpler systems: the simpler of the two is the unigenic system of GEP that evolves a single parse tree and therefore bares some resemblance to the GP system without ADFs; the other one is the multigenic system of GEP that evolves multiple parse trees linked together by a predefined linking function.

5.2.1. The Unigenic System

For this analysis, we are going to use the basic Gene Expression Algorithm both with and without random constants. In both cases, though, a single tree structure encoded in a single gene will be evolved.

As it is customary, the parameters used per run are summarized in a table (Table 1). Note that, in these experiments, small population sizes of only 50 individuals were used, incomparably smaller than the 4,000 individuals used by Koza to solve this same problem (indeed, this population size of 50 individuals is kept constant throughout this work in order to allow a more fruitful comparison between the different systems). Also worth pointing out is that, throughout this study and whenever possible, a maximum program size around 50 points was used so that comparisons could be made between all the experiments (to be more precise, for the unigenic systems a head length of 24 gives maximum program size 49, whereas for the multigenic systems with four genes with head length six, maximum program size equals 52). So, for the unigenic systems of this study, chromosomes with a head length $h = 24$ were chosen, giving maximum program length of 49 (note, however, that the chromosome length in the system with random numerical constants is larger on account of the Dc domain).

Table 1. Settings for the sextic polynomial problem using a unigenic system with (**ugGEP-RNC**) and without (**ugGEP**) random numerical constants.

	ugGEP	**ugGEP-RNC**
Number of runs	100	100
Number of generations	200	200
Population size	50	50
Chromosome length	49	74
Number of genes	1	1
Head length	24	24
Gene length	49	74
Terminal set	a	a ?
Function set	+ - * /	+ - * /
Mutation rate	0.044	0.044
Inversion rate	0.1	0.1
RIS transposition rate	0.1	0.1
IS transposition rate	0.1	0.1
Two-point recombination rate	0.3	0.3
One-point recombination rate	0.3	0.3
Random constants per gene	--	5
Random constants data type	--	Integer
Random constants range	--	0-3
Dc-specific mutation rate	--	0.044
Dc-specific inversion rate	--	0.1
Dc-specific IS transposition rate	--	0.1
Random constants mutation rate	--	0.01
Number of fitness cases	50	50
Selection range	100	100
Precision	0.01	0.01
Success rate	26%	4%

As shown in Table 1, the probability of success for this problem using the unigenic system without random numerical constants is considerably higher than the success rate obtained with the ugGEP-RNC algorithm (26% as opposed to just 4%), which, again, shows that, for this kind of simple, modular problem, evolutionary algorithms fare far better if the numerical constants are created from scratch by the algorithm itself through the evolution of simple mathematical expressions. This is not to say that, for complex real-world problems of just one variable or really complex problems of higher dimensions, random numerical constants are not important; indeed, most of the time, they play a crucial role in the discovery of good solutions.

Let's take a look at the structure of the first perfect solution found using the unigenic system without the facility for the manipulation of random numerical constants:

```
012345678901234567890123456789012345678901234567890123456789012345678
*++a/****+-a--*/aa*/*---aaaaaaaaaaaaaaaaaaaaaaaaa                    (16)
```

As its expression shows, it contains a relatively big neutral region involving a total of nine nodes and two smaller ones involving just three nodes each. It is also worth pointing out the creation of the numerical constant 1 through the simple arithmetic operation *a*/*a*.

It is also interesting to take a look at the structure of the first perfect solution found using the unigenic system with the facility for the manipulation of random numerical constants:

```
0123456789012345678901234567890123456789...
*a+*aa+*a*/aa-a*++-?aa*a?aaa???aaaaaa?a?...

             ...0123456789012345678901234567890123
             ...a?a???a?a0210121021334303442030040

A = {1, 1, 1, 1, 3}                                         (17)
```

As its expression reveals, it is a fairly compact solution with no obvious neutral regions that makes good use of the random numerical constant 1 to evolve a perfect solution.

5.2.2. The Multigenic System with Static Linking

For this analysis we are going to use again both the basic Gene Expression Algorithm without random constants and GEP with random numerical constants. The parameters and the performance of both experiments are shown in Table 2.

It's worth pointing out that maximum program length in these experiments is similar to the one used in the unigenic systems of the previous section. Here, head lengths $h = 6$ and four genes per chromosome were used, giving maximum program length of 52 points (again note that the chromosome length in the systems with random numerical constants is larger on account of the Dc domain, but maximum program length remains the same).

As you can see by comparing Tables 1 and 2, the use of multiple genes resulted in a considerable increase in performance for both systems. In the systems without random constants, by partitioning the genome into four autonomous genes, the performance increased from 26% to 93%, whereas in the systems with random numerical constants, the performance increased from 4% to 49%. Note also that, in this analysis, the already familiar pattern is observed when random numerical constants are introduced: the success rate decreases considerably from 93% to 49% (in the unigenic systems it decreased from 26% to 4%).

Let's also take a look at the structure of the first perfect solution found using the multigenic system without the facility for the manipulation of random numerical constants (the sub-ETs are linked by multiplication):

```
0123456789012012345678901201234567890120123456789012
+/aaa/aaaaaaa+//a/aaaaaaaa-**a/aaaaaaaa-**a/aaaaaaaa   (18)
```

As its expression shows, it contains three small neutral regions involving a total of nine

Table 2. Settings for the sextic polynomial problem using a multigenic system with (**mgGEP-RNC**) and without random numerical constants (**mgGEP**).

	mgGEP	mgGEP-RNC
Number of runs	100	100
Number of generations	200	200
Population size	50	50
Chromosome length	52	80
Number of genes	4	4
Head length	6	6
Gene length	13	20
Linking function	*	*
Terminal set	a	a ?
Function set	+ - * /	+ - * /
Mutation rate	0.044	0.044
Inversion rate	0.1	0.1
RIS transposition rate	0.1	0.1
IS transposition rate	0.1	0.1
Two-point recombination rate	0.3	0.3
One-point recombination rate	0.3	0.3
Gene recombination rate	0.3	0.3
Gene transposition rate	0.1	0.1
Random constants per gene	--	5
Random constants data type	--	Integer
Random constants range	--	0-3
Dc-specific mutation rate	--	0.044
Dc-specific inversion rate	--	0.1
Dc-specific IS transposition rate	--	0.1
Random constants mutation rate	--	0.01
Number of fitness cases	50	50
Selection range	100	100
Precision	0.01	0.01
Success rate	93%	49%

nodes, all encoding the numerical constant 1. Note also that, in two occasions (in sub-ETs 0 and 1), the numerical constant 1 plays an important role in the overall making of the perfect solution. Also interesting about this perfect solution, is that genes 2 and 3 are exactly the same, suggesting a major event of gene duplication (it's worth pointing out that the duplication of genes can only be achieved by the concerting action of gene recombination and gene transposition, as a gene duplication operator is not part of the genetic modification arsenal of Gene Expression Programming).

It is also interesting to take a look at the structure of the first perfect solution found using the multigenic system with the facility for the manipulation of random numerical constants (the sub-ETs are linked by multiplication):

```
01234567890123456789
+--+*aa??aa?a0444212
+--+*aa??aa?a0244422
a?a??a?aaa?a?2212021
aa-a*/?aa????3202123

A0 = {0, 3, 1, 2, 1}
A1 = {0, 3, 1, 2, 1}
A2 = {0, 3, 1, 2, 1}
A3 = {3, 3, 2, 0, 2}                                    (19)
```

As its expression reveals, it is a fairly compact solution with two small neutral motifs plus a couple of neutral nodes, all representing the numerical constant zero. Note that genes 0 and 1 are almost exact copies of one another (there is only variation at positions 17 and 18, but they are of no consequence as they are part of a noncoding region of the gene), suggesting a recent event of gene duplication. Note also that although genes 2 and 3 encode exactly the same sub-ET (a simple sub-ET with just one node), they most certainly followed different evolutionary paths as the homology between their sequences suggests.

5.3. Results with ADFs

In this section, we are going to conduct a series of four studies, each with four different experiments. In the first study, we are going to use a unicellular system encoding 1, 2, 3, and 4 ADFs. In the second study, we are going to use again a unicellular system, encoding also 1, 2, 3, and 4 ADFs, but, in this case, the ADFs will also incorporate random numerical constants. The third and fourth studies are respectively similar to the first and second one, with the difference that we are going to use a system with multiple cells (three, to be precise) instead of just the one.

5.3.1. The Unicellular System

For the unicellular system without random numerical constants, both the parameters and performance are shown in Table 3. It's worth pointing out, that, in these experiments, we are dealing with Automatically Defined Functions that can be reused again and again, and therefore it makes little sense to talk about maximum program length. However, in these series of experiments the same head length of six was used to encode all the ADFs and a system with four ADFs was also analyzed, thus allowing the comparison of these cellular systems with the simpler acellular ones (recall that we used four genes with $h = 6$ in the multigenic system and one gene with $h = 24$ in the unigenic system).

As you can see in Table 3, these cellular systems with Automatically Defined Functions perform extremely well, especially if we compare them with the unigenic system (see Table 1), which, as you recall, is the closer we can get to the Genetic Programming system. So, we can also conclude that the use of ADFs can bring considerable benefits to systems

Table 3. Settings and performance for the sextic polynomial problem using a unicellular system encoding 1, 2, 3, and 4 ADFs.

	1 ADF	2 ADFs	3 ADFs	4 ADFs
Number of runs	100	100	100	100
Number of generations	200	200	200	200
Population size	50	50	50	50
Chromosome length	22	35	48	61
Number of genes/ADFs	1	2	3	4
Head length	6	6	6	6
Gene length	13	13	13	13
Function set of ADFs	+ - * /	+ - * /	+ - * /	+ - * /
Terminal set	a	a	a	a
Number of homeotic genes/cells	1	1	1	1
Head length of homeotic genes	4	4	4	4
Length of homeotic genes	9	9	9	9
Function set of homeotic genes	+ - * /	+ - * /	+ - * /	+ - * /
Terminal set of homeotic genes	ADF 0	ADFs 0-1	ADFs 0-2	ADFs 0-3
Mutation rate	0.044	0.044	0.044	0.044
Inversion rate	0.1	0.1	0.1	0.1
RIS transposition rate	0.1	0.1	0.1	0.1
IS transposition rate	0.1	0.1	0.1	0.1
Two-point recombination rate	0.3	0.3	0.3	0.3
One-point recombination rate	0.3	0.3	0.3	0.3
Gene recombination rate	0.3	0.3	0.3	0.3
Gene transposition rate	--	0.1	0.1	0.1
Mutation rate in homeotic genes	0.044	0.044	0.044	0.044
Inversion rate in homeotic genes	0.1	0.1	0.1	0.1
RIS transposition rate in homeotic genes	0.1	0.1	0.1	0.1
IS transposition rate in homeotic genes	0.1	0.1	0.1	0.1
Number of fitness cases	50	50	50	50
Selection range	100	100	100	100
Precision	0.01	0.01	0.01	0.01
Success rate	82%	78%	69%	63%

limited to just one gene or parse tree, especially if there is some modularity in the problem at hand. Note, however, that the unicellular system slightly pales in comparison to the multigenic system with static linking (see Table 2), showing that the multigenic system of GEP is already a highly efficient system that can be used to solve virtually any kind of problem.

It is worth pointing out that, in his analysis of the role of ADFs to solve this problem, Koza uses just one ADF, which, as you can see in Table 3, is the most successful organization to solve this problem, with a success rate of 82%. And, curiously enough, this same pattern appears in all the experiments conducted here, in which the highest success rates were obtained when just one ADF was used (see Tables 3, 4, 5, and 6).

Let's take a look at the structure of the first perfect solution found using the unicellular system encoding just one ADF:

```
012345678901201234567 8
-a**aaaaaaaaa*/0*00000                                    (20)
```

As its expression reveals, the main program is far from parsimonious and could be simplified to $(ADF)^2$. But, nevertheless, it illustrates perfectly how a useful building block created by the evolutionary process itself can be reused several times by the main program encoded in the homeotic gene.

Let's also analyze the structure of a program with more than one ADF, the individual below with four ADFs (genes are shown separately):

```
0123456789012
*-a--+aaaaaaa
/-a---aaaaaaa
-a*+*-aaaaaaa
a+*-+aaaaaaaa
*22121133                                                 (21)
```

As its expression shows, the main program is fairly compact and invokes only ADF_2. All the remaining ADFs are not used at all, and, therefore, are free to evolve without much pressure. We know already that neutral regions play an important role both in natural evolution (Kimura 1983) and in GEP (Ferreira 2002b), and that their use in good measure is responsible for a considerable increase in performance. And the same phenomenon is observed in these cellular systems, where the simplest one with only one ADF and a single cell seems to have the right amount of neutral regions as it evolves very efficiently with a success rate of 82%. So, in this particular problem, by increasing the number of ADFs, we are basically increasing the number of neutral regions, and the performance for this simple modular problem decreases proportionately, dropping down to 63% in the system with four ADFs (see Table 3).

Let's now analyze the behavior of the unicellular system when random numerical constants are also incorporated in the Automatically Defined Functions.

For that purpose a similar set of experiments were done, using also 1, 2, 3, and 4 ADFs (Table 4). And as expected, a considerable decrease in performance was observed comparatively to the performance observed in the unicellular system without random numerical constants (see Table 3).

Let's take a look at the structure of the first perfect solution found using the unicellular system encoding just one ADF with random numerical constants:

```
012345678901234567890 12345678
---*a-?aa?a??0412201*+0*00000

A = {1, 0, 3, 1, 2}                                       (22)
```

Table 4. Settings and performance for the sextic polynomial problem using a unicellular system encoding 1, 2, 3, and 4 ADFs with random numerical constants.

	1 ADF	2 ADFs	3 ADFs	4 ADFs
Number of runs	100	100	100	100
Number of generations	200	200	200	200
Population size	50	50	50	50
Chromosome length	29	49	69	89
Number of genes/ADFs	1	2	3	4
Head length	6	6	6	6
Gene length	20	20	20	20
Function set of ADFs	+ - * /	+ - * /	+ - * /	+ - * /
Terminal set	a ?	a ?	a ?	a ?
Number of homeotic genes/cells	1	1	1	1
Head length of homeotic genes	4	4	4	4
Length of homeotic genes	9	9	9	9
Function set of homeotic genes	+ - * /	+ - * /	+ - * /	+ - * /
Terminal set of homeotic genes	ADF 0	ADFs 0-1	ADFs 0-2	ADFs 0-3
Mutation rate	0.044	0.044	0.044	0.044
Inversion rate	0.1	0.1	0.1	0.1
RIS transposition rate	0.1	0.1	0.1	0.1
IS transposition rate	0.1	0.1	0.1	0.1
Two-point recombination rate	0.3	0.3	0.3	0.3
One-point recombination rate	0.3	0.3	0.3	0.3
Gene recombination rate	0.3	0.3	0.3	0.3
Gene transposition rate	--	0.1	0.1	0.1
Random constants per gene	5	5	5	5
Random constants data type	Integer	Integer	Integer	Integer
Random constants range	0-3	0-3	0-3	0-3
Dc-specific mutation rate	0.044	0.044	0.044	0.044
Dc-specific inversion rate	0.1	0.1	0.1	0.1
Dc-specific IS transposition rate	0.1	0.1	0.1	0.1
Random constants mutation rate	0.01	0.01	0.01	0.01
Mutation rate in homeotic genes	0.044	0.044	0.044	0.044
Inversion rate in homeotic genes	0.1	0.1	0.1	0.1
RIS transposition rate in homeotic genes	0.1	0.1	0.1	0.1
IS transposition rate in homeotic genes	0.1	0.1	0.1	0.1
Number of fitness cases	50	50	50	50
Selection range	100	100	100	100
Precision	0.01	0.01	0.01	0.01
Success rate	57%	42%	33%	26%

As its expression shows, the simple module discovered in the structure encoding the ADF (a^2-1) is called four times in the main program, creating a perfect solution with just one kind of building block.

Let's now analyze the structure of a program with more than one ADF, the individual below with four ADFs with random numerical constants (the genes are shown separately):

```
01234567890123456789
*-+*a*?aaa?a?3324010
-aa?*-aa?a???3123440
*a/a-a?aa?a??2234201
--*+*+??aa?aa0141122
*00003233

A0 = {1, 1, 2, 1, 1}
A1 = {2, 3, 0, 2, 0}
A2 = {3, 1, 0, 0, 3}
A3 = {2, 0, 3, 2, 2}                                        (23)
```

As its expression reveals, the main program is fairly compact and invokes only ADF_0. Indeed, for this simple modular problem, most perfect solutions involve just one ADF, suggesting that this problem is better solved using just one kind of building block that can then be used as many times as necessary by the main program. And the fact that the system evolves more efficiently with just one ADF is just another indication of this (57% success rate in the system with just one ADF versus 26% in the system with four ADFs).

5.3.2. The Multicellular System

For the multicellular system without random numerical constants, both the parameters and performance are shown in Table 5.

As you can see, these multicellular systems with Automatically Defined Functions perform extremely well, even better than the multigenic system with static linking (see Table 2). And they are very interesting because they can also be used to solve problems with multiple outputs, where each cell is engaged in the identification of one class or output. Here, however, we are using a multicellular system to solve a problem with just one output, which means that all the cells are trying to find the same kind of solution and, therefore, for each individual, the fitness is determined by the best cell. Obviously, the greater the number of cells the higher the probability of evolving the perfect solution or cell. But there is one caveat though: one cannot go on increasing the number of cells indefinitely because it takes time and resources to express all of them. The use of three cells per individual seems a good compromise, and we are going to use just that in all the experiments of this section.

Let's take a look at the structure of the first perfect solution found using the multicellular system encoding just one ADF (which cell is best is indicated after the coma):

```
0123456789012012345678012345678012345678
-*a*aaaaaaaaa-0-+000000***00000-**+00000,2                   (24)
```

As its expression shows, the best main program invokes the ADF encoded in the conventional gene five times. Note, however, that this perfect solution is far from parsimonious and could indeed be simplified to $(ADF)^2$.

Table 5. Settings and performance for the sextic polynomial problem using a multicellular system encoding 1, 2, 3, and 4 ADFs.

	1 ADF	2 ADFs	3 ADFs	4 ADFs
Number of runs	100	100	100	100
Number of generations	200	200	200	200
Population size	50	50	50	50
Chromosome length	40	53	66	79
Number of genes/ADFs	1	2	3	4
Head length	6	6	6	6
Gene length	13	13	13	13
Function set of ADFs	+ - * /	+ - * /	+ - * /	+ - * /
Terminal set	a	a	a	a
Number of homeotic genes/cells	3	3	3	3
Head length of homeotic genes	4	4	4	4
Length of homeotic genes	9	9	9	9
Function set of homeotic genes	+ - * /	+ - * /	+ - * /	+ - * /
Terminal set of homeotic genes	ADF 0	ADFs 0-1	ADFs 0-2	ADFs 0-3
Mutation rate	0.044	0.044	0.044	0.044
Inversion rate	0.1	0.1	0.1	0.1
RIS transposition rate	0.1	0.1	0.1	0.1
IS transposition rate	0.1	0.1	0.1	0.1
Two-point recombination rate	0.3	0.3	0.3	0.3
One-point recombination rate	0.3	0.3	0.3	0.3
Gene recombination rate	0.3	0.3	0.3	0.3
Gene transposition rate	--	0.1	0.1	0.1
Mutation rate in homeotic genes	0.044	0.044	0.044	0.044
Inversion rate in homeotic genes	0.1	0.1	0.1	0.1
RIS transposition rate in homeotic genes	0.1	0.1	0.1	0.1
IS transposition rate in homeotic genes	0.1	0.1	0.1	0.1
Number of fitness cases	50	50	50	50
Selection range	100	100	100	100
Precision	0.01	0.01	0.01	0.01
Success rate	98%	96%	95%	90%

It is also interesting to take a look at what the other cells are doing. For instance, $Cell_0$ encodes zero and $Cell_1$ encodes $(ADF)^4$, both a far cry from the perfect solution.

Let's also analyze the structure of a program with more than one ADF, the individual below with four ADFs and three cells (the best cell is indicated after the coma):

```
0123456789012012345678901201234567890120123456789012...
**-/aaaaaaaaa/aaaa/aaaaaaa*-a*/aaaaaaaa/-***/aaaaaaa...

                ...012345678012345678012345678
                ...*-*222301*+2021323-+0020321,1          (25)
```

As its expression shows, the best main program invokes two different ADFs (ADF_0 and ADF_2), but since ADF_0 encodes zero, the best cell could be simplified to $(ADF_2)^2$, which is

again a perfect solution built with just one kind of building block (a^3-a). It is also worth noticing that two of the ADFs (ADF_0 and ADF_3) and one of the cells ($Cell_0$) encode zero, and the numerical constant 1 is also encoded by ADF_1; they are all good examples of the kind of neutral region that permeates all these solutions.

Let's now analyze the behavior of the multicellular system when random numerical constants are also incorporated in the Automatically Defined Functions.

For that purpose a similar set of experiments were done, using also 1, 2, 3, and 4 ADFs (Table 6). And as expected, a considerable decrease in performance was observed comparatively to the performance obtained in the multicellular system without random numerical constants (see Table 5). Notwithstanding, ADFs with random numerical constants perform quite well despite the additional complexity, and they may prove valuable in problems where random numerical constants are crucial to the discovery of good solutions.

Let's take a look at the structure of a perfect solution found using the multicellular system encoding just one ADF with random constants (the best cell is indicated after the coma):

```
01234567890123456789012345678012345678012345678
-***a?aaa???a4000424+/0+00000*0*/00000-+*-00000,1

A = {3, 1, 1, 1, 1}                                    (26)
```

As its expression shows, the main program encoded in $Cell_1$ is far from parsimonious, but it encodes nonetheless a perfect solution to the sextic polynomial (14). The only available ADF is called four times from the main program, but in essence it could have been called just twice as it can be simplified to $(ADF)^2$.

Let's now analyze the structure of a program with more than one ADF, the individual below with four ADFs and three cells (the best cell is indicated after the coma):

```
01234567890123456789
-a*-*a?aa????2322013
?-*//aaa?aa?a2412442
*a+a*aa?aaaaa4024010
*-a?a?aaa????3224232

A0 = {0, 0, 0, 1, 0}
A1 = {2, 0, 0, 2, 2}
A2 = {2, 1, 3, 0, 0}
A3 = {2, 1, 3, 0, 0}

01234567801234567801234567
*2*/00213/-+*03022233201102,0                          (27)
```

As its expression shows, the best main program invokes two different ADFs (ADF_0 and ADF_2), but the calls to ADF_2 cancel themselves out, and the main program is reduced to $(ADF_0)^2$, which, of course is a perfect solution to the problem at hand.

Table 6. Settings and performance for the sextic polynomial problem using a multicellular system encoding 1, 2, 3, and 4 ADFs with random numerical constants.

	1 ADF	2 ADFs	3 ADFs	4 ADFs
Number of runs	100	100	100	100
Number of generations	200	200	200	200
Population size	50	50	50	50
Chromosome length	29	67	87	107
Number of genes/ADFs	1	2	3	4
Head length	6	6	6	6
Gene length	20	20	20	20
Function set of ADFs	+ - * /	+ - * /	+ - * /	+ - * /
Terminal set	a ?	a ?	a ?	a ?
Number of homeotic genes/cells	1	3	3	3
Head length of homeotic genes	4	4	4	4
Length of homeotic genes	9	9	9	9
Function set of homeotic genes	+ - * /	+ - * /	+ - * /	+ - * /
Terminal set of homeotic genes	ADF 0	ADFs 0-1	ADFs 0-2	ADFs 0-3
Mutation rate	0.044	0.044	0.044	0.044
Inversion rate	0.1	0.1	0.1	0.1
RIS transposition rate	0.1	0.1	0.1	0.1
IS transposition rate	0.1	0.1	0.1	0.1
Two-point recombination rate	0.3	0.3	0.3	0.3
One-point recombination rate	0.3	0.3	0.3	0.3
Gene recombination rate	0.3	0.3	0.3	0.3
Gene transposition rate	--	0.1	0.1	0.1
Random constants per gene	5	5	5	5
Random constants data type	Integer	Integer	Integer	Integer
Random constants range	0-3	0-3	0-3	0-3
Dc-specific mutation rate	0.044	0.044	0.044	0.044
Dc-specific inversion rate	0.1	0.1	0.1	0.1
Dc-specific IS transposition rate	0.1	0.1	0.1	0.1
Random constants mutation rate	0.01	0.01	0.01	0.01
Mutation rate in homeotic genes	0.044	0.044	0.044	0.044
Inversion rate in homeotic genes	0.1	0.1	0.1	0.1
RIS transposition rate in homeotic genes	0.1	0.1	0.1	0.1
IS transposition rate in homeotic genes	0.1	0.1	0.1	0.1
Number of fitness cases	50	50	50	50
Selection range	100	100	100	100
Precision	0.01	0.01	0.01	0.01
Success rate	79%	60%	58%	50%

Conclusions

Comparatively to Genetic Programming, the implementation of Automatically Defined Functions in Gene Expression Programming is very simple because it stands on the shoulders of the multigenic system with static linking and, therefore, requires just a small addition to make it work. And because the cellular system of GEP with ADFs, like all GEP systems, continues to be totally encoded in a simple linear genome, it poses no constraints whatso-

ever to the action of the genetic operators and, therefore, these systems can also evolve efficiently (indeed, all the genetic operators of GEP were easily extended to the homeotic genes). As a comparison, the implementation of ADFs in GP adds additional constraints to the already constrained genetic operators in order to ensure the integrity of the different structural branches of the parse tree. Furthermore, due to its mammothness, the implementation of multiple main programs in Genetic Programming is prohibitive, whereas in Gene Expression Programming the creation of a multicellular system encoding multiple main programs is a child's play.

Indeed, another advantage of the cellular system of GEP, is that it can easily grow into a multicellular one, encoding not just one but multiple cells or main programs, each using a different set of ADFs. These multicellular systems have multiple applications, some of which were already illustrated in this work, but their real potential resides in solving problems with multiple outputs where each cell encodes a program involved in the identification of a certain class or pattern. Indeed, the high performance exhibited by the multicellular system in this work gives hope that this system can be fruitfully explored to solve much more complex problems. In fact, in this work, not only the multicellular but also the unicellular and the multigenic system with static linking, were all far from stretched to their limits as the small population sizes of just 50 individuals used in all the experiments of this work indicate. As a comparison, to solve this same problem, the GP system with ADFs uses already populations of 4,000 individuals.

And yet another advantage of the ADFs of Gene Expression Programming, is that they are free to become functions of one or several arguments, being this totally decided by evolution itself. Again, in GP, the number of arguments each ADF takes must be a priori decided and cannot be changed during the course of evolution lest invalid structures are created.

And finally, the cellular system (and multicellular also) encoding ADFs with random numerical constants was for the first time described in this work. Although their performance was also compared to other systems, the main goal was to show that ADFs with random numerical constants can also evolve efficiently, extending not only their appeal but also the range of their potential applications.

Bibliography

Cramer, N. L. (1985). 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.

Dawkins, R. (1995). *River out of Eden*, Weidenfeld and Nicolson.

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

Ferreira, C. (2002a). *Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence*, Angra do Heroísmo, Portugal.

Ferreira, C. (2002b). Genetic Representation and Genetic Neutrality in Gene Expression Programming. *Advances in Complex Systems*, 5 (4): 389-408.

Ferreira, C. (2002c). Mutation, Transposition, and Recombination: An Analysis of the Evolutionary Dynamics. In H. J. Caulfield, S.-H. Chen, H.-D. Cheng, R. Duro, V. Honavar, E. E. Kerre, M. Lu, M. G. Romay, T. K. Shih, D. Ventura, P. P. Wang, Y. Yang, eds., *Proceedings of the 6th Joint Conference on Information Sciences, 4th International Workshop on Frontiers in Evolutionary Algorithms*, 614-617, Research Triangle Park, North Carolina, USA.

Ferreira, C. (2004a). Gene Expression Programming and the Evolution of Computer Programs. In Leandro N. de Castro and Fernando J. Von Zuben, eds., *Recent Developments in Biologically Inspired Computing*, pages 82-103, Idea Group Publishing.

Ferreira, C. (2004b). Designing Neural Networks Using Gene Expression Programming. *9th Online World Conference on Soft Computing in Industrial Applications*, September 20 - October 8, 2004.

Friedberg, R. M. (1958). A Learning Machine: Part I. *IBM Journal*, 2 (1): 2-13.

Friedberg, R. M., B. Dunham, and J. H. North (1959). A Learning Machine: Part II. *IBM Journal*, 3 (7): 282-287.

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

Kimura, M. (1983). *The Neutral Theory of Molecular Evolution*, Cambridge University Press, Cambridge, UK.

Koza, J. R., F. H. Bennett III, D. Andre, and M. A. Keane (1999). *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco: Morgan Kaufmann Publishers.

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

Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*, Cambridge, MA: MIT Press.