International Journal on Artificial Intelligence Tools

GENETIC PROGRAMMING WITH LINEAR REPRESENTATION A SURVEY

MIHAI OLTEAN, CRINA GROŞAN, LAURA DIOŞAN, CRISTINA MIHĂILĂ*
{moltean,cgrosan,lauras,aneta}@cs.ubbcluj.ro

Received (Day Month Year)
Revised (Day Month Year)
Accepted (25 June 2008)

Genetic Programming (GP) is an automated method for creating computer programs starting from a high-level description of the problem to be solved. Many variants of GP have been proposed in the recent years. In this paper we are reviewing the main GP variants with linear representation. Namely, Linear Genetic Programming, Gene Expression Programming, Multi Expression Programming, Grammatical Evolution, Cartesian Genetic Programming and Stack-Based Genetic Programming. A complete description is provided for each method. The set of applications where the methods have been applied and several Internet sites with more information about them are also given.

Keywords: Genetic Programming, Linear Genetic Programming, Gene Expression Programming, Multi Expression Programming, Grammatical Evolution, Cartesian Genetic Programming, Stack-Based Genetic Programming

1. Introduction

Genetic Programming is widely known as the technique which writes computer programs. Since the term of GP was actually coined many variants of the standard GP have been proposed. Their aims were various: simpler implementation, higher speed, smaller memory requirements, the capability of working with particular hardware architectures etc. Another motivation is given by the problems where some representations work better than the others[1,2,3].

Among these variants a special place is taken by those techniques which have a linear representation of solutions. This basically means that we manipulate the chromosomes, encoding computer programs, in the same manner as we manipulate string-based chromosomes, even if these computer programs have a tree or graph-like execution flow on a normal computer.

Linear encoding of computer programs means that we usually work with arrays of fixed or variable lengths. Specifically:

(1) we generate arrays of instructions, having a particular meaning,

*Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Kogălniceanu, 1, Cluj-Napoca, 400 084, România

(2) we recombine them by using string-based crossover operators such as those from binary encoding (such as one-cutting point, two cutting points, uniform recombination etc.)
(3) we mutate them using operators inspired from the binary encoding or from other representations

This paper performs a review of the most significant GP variants which encode computer programs in linear form. The techniques described in this paper are: Linear Genetic Programming[4], Gene Expression Programming[5], Multi Expression Programming[6], Grammatical Evolution[7], Cartesian Genetic Programming[8] and Stack-Based Genetic Programming[9].

The following elements are present in the description of each technique:

(1) representation - the way in which a computer program is encoded into a chromosome,
(2) initialization - the way in which a chromosome is (sometime randomly) generated,
(3) genetic operators - the way in which variation is introduced in the population,
(4) main algorithm - the strategy which guides the search process,
(5) strengths and weaknesses - the benefits and the difficulties that a researcher or practitioners will meet when using a particular technique. This is a crucial section intended to help the reader choose one technique instead of another depending on the task being solved,
(6) applications - a set of problems where a particular technique has been applied,
(7) on-line resources - several web sites where the reader can find more information (possibly the source code) about the considered techniques.

The motivation for writing this survey was raised by the lack of a unitary presentation of all GP variants in the literature. Some authors used long descriptions for presenting their method in a new light, totally different from what was proposed before. These kinds of presentations can confuse the reader, making him ignorant in the vast field of Genetic Programming. Our second motivation, beside a unitary presentation, is to use this survey as a starting point for assessing the true performance of each GP variant. Each method has some weaknesses (as we shown later in this paper) and these aspects make them vulnerable when solving some problems. By making a complete comparison - non-numerical and numerical - the true potential of each method can be revealed, making much easier the task of selecting them when solving problems. Here, we focus on a theoretical comparison between methods. Numerical experiments are planned for the future, due to numerous problems encountered in that direction (as stated in our last section of this paper).

The paper is organized as follows: Section 2 contains five preparatory steps in order to solve a given problem by using a GP algorithm. Sections 3 - 8 describe the best known GP variants and their applications (see Section 9). Conclusions and further work directions are suggested in Section 10.

2. Prerequisites

Five major preparatory steps[10] must be specified in order to apply a GP technique to a particular problem:

(1) the set T of terminals (e.g. the independent variables of the problem, zero-argument functions and random constants)
(2) the set F of primitive functions,
(3) the fitness measure (for explicitly or implicitly measuring the quality of individuals in the population),
(4) certain parameters for controlling the run
(5) the termination criterion and the method for designating the result of the run.

These preparatory steps are problem-dependent so they must be specified for each particular problem by a human user. In this paper we will use regression and classification problems in order to illustrate the way in which the described GP techniques can be applied. In both cases the problem consists in finding a mathematical expression.

The quality of a GP individual, the fitness measure, is usually computed by using a set of fitness cases[11],[12].

We consider a problem with n inputs: x_1, x_2, $\ldots x_n$ and one output f. Each fitness case is given as a one-dimensional array of $(n+1)$ values:

$$v_1^k, v_2^k, ..., v_n^k, f^k$$

where v_j^k is the value of the j^{th} attribute, x_j, with $j = 1, m$, in the k^{th} fitness case and f^k is the output for the k^{th} fitness case $(k = 1, m)$.

3. Linear Genetic Programming

Linear Genetic Programming (LGP)[4,13,14,1,2] uses a linear representation of computer programs. LGP evolves programs written in an imperative language (like C), rather than the tree-based expressions as in the case of standard GP[11].

LGP is a variant of Automatic Induction of Machine Code - Genetic Programming (AIM-GP)[15,16] - method that was originally referred to as Compiling Genetic Programming System (CGPS)[17]. Note that the use of linear bit sequences in GP goes back to Cramer and his JB language[18] and other works.

In AIMGP individuals are manipulated as binary machine code (which is the main difference to the LGP approach where programs are represented in an imperative language) in memory and are executed directly without passing an interpreter during the fitness calculation. This results in a significant speedup compared to interpreting systems.

3.1. *LGP model*

3.1.1. *Representation*

An LGP individual is represented by a variable-length sequence of simple C language instructions. Instructions operate on one or two indexed variables v (also called registers) or on constants c from predefined sets. The result is assigned to a destination register. Single operations may be skipped by preceding conditional branches, e.g., *if* $(r_j > r_k)$. Some examples of LGP instructions are given below:

- $v_i = v_j * v_k$ // instruction operating on two registers
- $v_i = v_j * c$ // instruction operating on one register and one constant
- $v_i = sin(v_j)$ // instruction operating on one register

Example. An example of the LGP program is the following:

```
void LGP_program (double v[11])
  {
      ...
      v[8] = v[0] − 10;
      v[6] = v[2] ∗ v[0];
      v[5] = v[8] ∗ 7;
      v[4] = v[2] − v[0];
      v[10] = v[1]/v[4];
      v[3] = sin(v[1]);
      v[1] = v[8] − v[6];
      v[7] = v[10] ∗ v[3];
      v[9] = v[0] + v[7];
      v[2] = v[7] + 3;
      ...
  }
```

The initial values for the variables $v[0]$ ... $v[10]$ are set to the value of inputs or to some numerical constants. After executing the program encoded into an LGP chromosome the output will be stored into a destination register. This register is usually chosen at the beginning of the search process and is kept unchanged until the end.

It can be seen that not all the variables (registers) are effective (contribute to the final result). The usefulness of each register depends on the register chosen to provide the output of the program.

Suppose that in the previously described chromosome the output is provided by the register $v[9]$. The last effective instruction (that modifies this register) is $v[9] = v[0] * v[3]$. Now we have to search for the previous instructions that have changed the value of registers $v[0]$ and $v[7]$. $v[0]$ has not been changed, so we look for $v[7]$. This instruction is $v[7] = v[10] * v[3]$. The process continues until we have

found the effective code for this program:

```
void LGP_effective_program (double v[11])
  {
      ...
      v[4] = v[2] − v[0];
      v[10] = v[1]/v[4];
      v[3] = sin(v[1]);
      v[7] = v[10] ∗ v[3];
      v[9] = v[0] + v[7];
      ...
  }
```

The size of the effective code varies between 0 and the number of instructions in the LGP chromosome.

A chromosome is a string of instructions. Each instruction is encoded as

$$(op\_index, out\_register, in\_register_1, in\_register_2)$$

where *op_index* is an integer that represent an index from the set of functions, while *out_register*, $in\_register_1$ and $in\_register_2$ represent indexes from the set of registers (in our example a number between 0 an 10). The corresponding chromosome for our example program is:

$$C = ((1, 4, 2, 0), (3, 10, 1, 4), (4, 3, 1), (2, 7, 10, 3), (0, 9, 0, 7))$$

or

$$C = ((-, v[4], v[2], v[0]), (/, v[10], v[1], v[4]), (sin, v[3], v[1]),$$

$$(*, v[7], v[10], v[3]), (+, v[9], v[0], v[7]))$$

If we consider that the first three registers contain the values a, b and c, then the chromosome C encodes the expression $a + \frac{b}{c-a} * sin(b)$.

3.1.2. *Initialization*

The initial population of a LGP run is generated randomly. An upper bound for the initial program length has to be defined. The lower bound may be equal to the absolute minimum length of a program (that is one instruction). A program is created so that its length is chosen randomly from this predefined range with a uniform probability. Each symbol in the program is randomly chosen from the corresponding set.

There is a trade-off to be addressed when choosing upper and lower bounds of program length: On the one hand, it is not recommended to initialize exceedingly long programs. This may reduce their variability significantly in the course of the

evolutionary process. Besides, the smaller the initial programs are, the more thorough an exploration of the search space can be performed at the beginning of a run. On the other hand, the average initial length of programs should not be too small, because a sufficient diversity of the initial genetic material is necessary, especially in smaller populations or if crossover dominates variation[2].

3.1.3. *Genetic operators*

Variation operators used in conjunction with LGP are crossover and mutation. Both crossover and mutation must handle variable length chromosomes.

Crossover. By crossover, continuous sequences of instructions are selected and exchanged between parents. LGP uses two-point string crossover[4]. A segment of random starting position and random length is selected in both parents and exchanged between them. If one of the resulting children exceeds the maximum length, crossover is abandoned and restarted by exchanging equally sized segments.

Mutation. Two types of mutation are used: micro mutation and macro mutation. By micro mutation, an operand or an operator of an instruction is changed. Macro mutation inserts or deletes a random instruction. As an effect of macro mutation, the size of the LGP chromosome is modified.

3.1.4. *Algorithm*

LGP uses a modified steady-state algorithm.

Algorithm 1 LGP Algorithm

@ Generate the initial population.
while not stop_condition **do**
  @ Four individuals are randomly selected from the current population
  @ The best two of them are considered the winners of the tournament and will act as parents
  @ The parents are recombined
  @ The offspring are mutated and then they replace the losers of the tournament
end while
@ Output S as the best solution (individual) found

3.2. *LGP strengths and weaknesses*

3.2.1. *Strengths*

Evolving programs in a low-level language allows us to run these programs directly on the computer processor, thus avoiding the need of an interpreter. Computer programs can be evolved very quickly in this way.

LGP can be easily used to solve problems with multiple outputs (by choosing multiple registers to provide the solution).

3.2.2. *Weaknesses*

An important LGP parameter is the number of registers (or variables) used by a chromosome. This number is usually equal to the number of attributes of the problem. If the problem has only one attribute, it is impossible to obtain a complex expression such as the quartic polynomial[11]. In that case, we have to use several supplementary registers (variables). The number of supplementary registers (variables) depends on the complexity of the expression being discovered. An inappropriate choice of the number of registers(variables) may lead to poor results.

3.3. *LGP on-line resources*

More information about LGP can be found on the following web pages:

- Register Machine Learning Technologies `http://www.aimlearning.com` – last access June 6, 2008
- Peter Nordin's home page `http://fy.chalmers.se/~pnordin` – last access June 6, 2008
- Wolfgang Banzhaf's home page `http://www.cs.mun.ca/~banzhaf` – last access June 6, 2008
- Markus Brameier's home page `http://www.daimi.au.dk/~brameier` – last access June 6, 2008

4. Gene Expression Programming

Gene Expression Programming (GEP)[5] is a GP variant, relying on linear chromosomes. A GEP chromosome is composed of genes containing terminal and function symbols. Several dedicated operators such as crossover, mutation and transposition modify GEP chromosomes.

4.1. *GEP model*

4.1.1. *Representation*

GEP individuals[19,5] are encoded as linear chromosomes which are expressed or translated into expression trees (branched entities). For a better understanding of the method we will start with an example showing breadth-first translation of a tree.

Example. Consider the tree depicted in Figure 1. By breadth-first parsing of the tree depicted in Figure 1 we obtain the string $+a*/Sb-bca$, where the S symbol stands for the *sin* operator. By decoding GEP chromosomes one actually has to perform a breadth-first parsing.

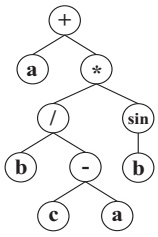

Fig. 1. A tree encoding the expression $a + b/(c - a) * sin(b)$.

Functionally speaking, GEP genes are composed of a *head* and a *tail*. The head contains both functions and terminal symbols. The tail contains only terminal symbols. The head length, h, is chosen by the user for each problem. The tail length, t, is calculated using the formula: $t = (n - 1) * h + 1$, where n is the maximum arity of functions.

Let us consider a gene made up of symbols in the sets F and T, where $F = \{+, -, *, /, S\}$ and $T = \{a, b, c\}$.

In this case maximum arity is $n = 2$. Because only the *head* can contain functions we must choose at least $h = 7$. If h is 7 then t will be 8, and the length of the gene is $7 + 8 = 15$. Such a gene is given below (where symbol S stands for the *sin* operator): $C = +a * /Sb - bcacabbc$.

The expression encoded by the gene C is: $E = a + b/(c - a) * sin(b)$ and it represents the phenotypic transcription of a chromosome having C as its unique gene. The last five elements of this gene (*cabbc*) are not used.

Usually, a GEP gene is not entirely used for phenotypic transcription. If the first symbol in the gene is a terminal symbol, the expression tree consists of a single node. If all the symbols in the head are function symbols, the expression tree uses all the symbols of the gene.

GEP genes may be linked by a function symbol in order to obtain a fully functional chromosome. In the current version of GEP, the linking functions for algebraic expressions are addition and multiplication. A single type of function is used for linking multiple genes.

This seems to be enough in some situations[5]. However, generally, it is not a good idea to assume that the genes may be linked either by addition or by multiplication.

4.1.2. *Initialization*

GEP chromosomes are initialized randomly but they must fulfill the previously described rules regarding the symbols that can be contained in each of the two parts of a chromosome. Thus, the only restriction for a chromosome is that it's tail must contain only terminal symbols and the initialization process must obey this rule.

4.1.3. *Genetic operators*

The GEP technique uses several operators such as crossover, mutation and transposition in order to obtain new individuals.

Crossover. In GEP there are three kinds of recombination: one-point, two-point and gene recombination. In all the cases, two parent chromosomes are randomly chosen and paired to exchange some material between them.

The one-point and two-point recombination operators in the GEP representation are analogous to the corresponding binary representation operator. Two parents and one, respectively two, cutting-point(s) are chosen. Two offsprings are obtained from the parents, by exchanging genetic material according to the cutting-point(s).

In gene recombination, an entire gene is exchanged during crossover. The exchanged genes are randomly chosen and they need to have occupied the same position in the parent chromosomes. The newly created individuals contain genes from both parents. Note that with this kind of recombination, similar genes can be exchanged but, most of the time, the exchanged genes are very different and new material is introduced in the population.

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

Other GEP operators. Some GEP variants use transposition as one of the genetic operators in order to introduce variety in the population. The transposable elements of the GEP chromosome are fragments of the genome that can be activated and jump to another place in the chromosome[5]. In GEP there are three kinds of transposable elements:

(1) short fragments with a function or terminal in the first position that transpose to the head of genes except for the root,
(2) short fragments with a function in the first position that transpose to the root of genes,
(3) entire genes that transpose to the beginning of the chromosome.

4.1.4. *Algorithm*

GEP uses a generational algorithm.

Algorithm 2 GEP Algorithm

@ Generate the initial population.
while not stop_condition **do**
  @ A fixed number of the best individuals enter the next generation (elitism)
  @ Fill the mating pool by using tournament selection
  @ Pair randomly the individuals from the mating pool and recombine them
  @ Mutate the offspring
  @ Enter the new individuals into the next generation
end while
@ Output S as the best solution (individual) found

4.2. *GEP strengths and weaknesses*

4.2.1. *Strengths*

Dividing the GEP chromosome into two parts (*head* and *tail*), each of them containing specific symbols, provides an original and very efficient way of encoding syntactically correct computer programs. No other corrections are required in order to obtain a valid computer program. This is different from other techniques such as grammatical evolution (see Section 6) which allows chromosomes encoding invalid computer programs in the system.

4.2.2. *Weaknesses*

GEP, as it was described in the original paper, uses a multigenic representation. There are some problems regarding multigenic chromosomes. Generally, it is not a good idea to assume that the genes may be linked either by addition or by multiplication. Providing a particular linking operator means providing partial information to the expression that is being discovered. However, if other additional operators (such as $-$, $/$) are used as linking operators, then the complexity of the problem substantially grows (since the problem of determining how to mix these operators with the genes is as difficult as the initial problem).

Furthermore, the number of genes in the GEP multigenic chromosome raises a problem. As it can be seen in[5], the success rate of GEP increases with the number of genes in the chromosome. However, after a certain value, the success rate decreases if the number of genes in the chromosome is increased. This happens because we cannot force a complex chromosome to encode a less complex expression.

A large part of the chromosome is unused if the target expression is short and the head length is large. Note that this problem usually arises in systems that employ chromosomes with a fixed length.

4.3. *GEP on-line resources*

More information about Gene Expression Programming can be found on the following web pages:

- Gene Expression Programming website, `http://www.gepsoft.com` – last access June 6, 2008
- Heitor Lopes's home page `http://www.cpgei.cefetpr.br/~hslopes/index-english.html` – last access June 6, 2008
- Xin Li's home page `http://www.cs.uic.edu/~xli1/` – last access June 6, 2008
- GEP in C# `http://www.c-sharpcorner.com/Code/2002/Nov/GEPAlgorithm.asp` – last access June 6, 2008

5. Multi Expression Programming

Multi Expression Programming (MEP)[20,6,21,22] is a GP variant that uses a linear representation of chromosomes. MEP individuals are strings of genes encoding complex computer programs.

When MEP individuals encode expressions, their representation is similar to the way in which compilers translate *C* or *Pascal* expressions into machine code[24].

A unique MEP feature is the ability of selecting the best gene to provide the output for the chromosome. This is different from other GP techniques which employ a fixed gene for output. A similar situation is provided by Cartesian GP (see Section 7) where the outputs are evolved in the same manner as all the other symbols in the chromosome.

A single parsing of the chromosome can perform evaluation of the expressions encoded into a MEP individual.

The offspring obtained by crossover and mutation are always syntactically correct MEP individuals (computer programs). Thus, no extra processing for repairing newly obtained individuals is needed.

5.1. *MEP model*

5.1.1. *Representation*

MEP genes are (represented by) substrings of variable length. The number of genes per chromosome is constant. This number defines the length of the chromosome. Each gene encodes a terminal or a function symbol. A gene that encodes a function includes pointers towards the function arguments. Function arguments always have indices of lower values than the position of the function itself in the chromosome.

The MEP representation ensures that no cycle arises while the chromosome is decoded (phenotypically transcripted). According to the MEP representation scheme, the first symbol of the chromosome must be a terminal symbol. In this way, only syntactically correct programs (MEP individuals) are obtained.

Example. Consider a representation where the numbers on the left stand for gene labels. Labels do not belong to the chromosome, as they are provided only for explanation purposes.

For this example, we use the set of functions: $F = \{+, -, *, /, sin\}$, and the set of terminals $T = \{a, b, c\}$.

An example of a chromosome C using the sets F and T is given below (it encodes the expression $a + \frac{b}{c-a} * \sin b$):

1: a
2: b
3: + 1, 2
4: sin(2)
5: c
6: − 5, 1
7: / 2, 3
8: / 2, 6
9: ∗ 8, 4
10: − 7, 8
11: + 1, 9

The maximum number of symbols in the MEP chromosome is given by the formula:

$$\textit{No of Symbols} = (n{+}1) * (\textit{No of Genes} - 1) + 1,$$

where n is the number of arguments of the function with the greatest number of arguments.

The maximum number of effective symbols is achieved when each gene (excepting the first one) encodes a function symbol with the highest number of arguments. The minimum number of effective symbols is equal to the number of genes and it is achieved when all the genes encode terminal symbols only.

Translation of the MEP chromosome into a valid computer program is done top-down. A terminal symbol specifies a simple expression. A function symbol specifies a complex expression obtained by connecting the operands specified by the argument positions with the current function symbol.

For instance, genes 1, 2 and 5 in the previous example encode simple expressions formed by a single terminal symbol. These expressions are:

$$E_1 = a,$$
$$E_2 = b,$$
$$E_5 = c,$$

Gene 3 indicates the operation + on the operands located at positions 1 and 2 of the chromosome. Therefore, gene 3 encodes the expression:

$$E_3 = a + b.$$

Gene 4 indicates the operation *sin* on the operand located at position 2 . Therefore, gene 4 encodes the expression:

$$E_4 = sin(b)$$

Gene 6 indicates the operation $-$ on the operands located at positions 5 and 1 . Therefore, gene 6 encodes the expression:

$$E_6 = c - a.$$

Gene 7 indicates the operation / on the operands located at position 2 and 3. Therefore, this gene encodes the expression:

$$E_7 = b/(a + b).$$

Gene 8 indicates the operation / on the operands located at position 2 and 6. Therefore, this gene encodes the expression:

$$E_8 = b/(c - a).$$

Gene 9 indicates the operation $*$ on the operands located at positions 8 and 4 of the chromosome. Therefore, gene 9 encodes the expression:

$$E_9 = b/(c - a) * sin(b).$$

Gene 10 indicates the operation $-$ on the operands located at positions 7 and 8 of the chromosome. Therefore, gene 10 encodes the expression:

$$E_{10} = b/(a + b) - b/(c - a).$$

Gene 11 indicates the operation + on the operands located at positions 1 and 9 of the chromosome. Therefore, this gene encodes the expression:

$$E_{11} = a + b/(c - a) * sin(b).$$

There is neither practical nor theoretical evidence that one of these genes is better than the others. Moreover, Wolpert and McReady[25,26] proved that we cannot use the behavior of the search algorithm up to a certain moment for a particular test function in order to predict its future behavior on that function. This is why each MEP chromosome allows any gene to provide the output of the chromosome.

The value of these expressions may be computed by reading the chromosome top-down. Partial results are computed by Dynamic Programming[27] and are stored in a conventional manner.

The chromosome fitness is usually defined as the fitness of the best expression encoded by that chromosome. For instance, if we want to solve symbolic regression problems, the fitness of each sub-expression E_i may be computed using the formula:

$$fitness(E_i) = \sum_{k=1}^{m} \left| o_i^k - f^k \right|, \quad (1)$$

where o_i^k is the result obtained by the expression E_i for the fitness case k and f^k is the targeted result for the fitness case k. In this case the fitness needs to be minimized. The fitness of an individual is set to be equal to the lowest fitness of the expressions encoded in the chromosome:

$$fitness(C) = \min_i fitness(E_i). \quad (2)$$

5.1.2. *Initialization*

There are some restrictions for generating a valid MEP chromosome:

- The first gene of the chromosome must contain a terminal. If we have a function in the first position, we also need some pointers to some positions with a lower index. But, there are no other genes above the first gene.
- For all the other genes which encodes functions we have to generate pointers toward the function arguments. All these pointers must indicate toward genes which have a lower index than the current gene.

5.1.3. *Genetic operators*

The search operators used within the MEP algorithm are crossover and mutation. These search operators preserve the chromosome structure. All the offspring are syntactically correct expressions.

Crossover. By crossover, two parents are selected and recombined. Several variants of recombination have been considered for MEP: one-point recombination, two-point recombination and uniform recombination.

Mutation. Each symbol (terminal, function of function pointer) in the chromosome may be the target of the mutation operator. Some symbols in the chromosome are changed by mutation. In order to preserve the consistency of the chromosome, its first gene must encode a terminal symbol.

If the current gene encodes a terminal symbol, it may be changed into another terminal symbol or into a function symbol. In the latter case, the position(s) indicating the function argument(s) is(are) randomly generated. If the current gene encodes a function, the gene may be mutated into a terminal symbol or into another function (function symbol and pointers towards arguments).

We may say that the crossover operator occurs between genes and the mutation operator occurs inside genes.

5.1.4. *Algorithm*

The standard MEP algorithm uses steady-state evolutionary model[28] as its underlying mechanism.

Algorithm 3 MEP Algorithm

@ Randomly create the initial population ($P(0)$)
while not stop condition **do**
  @ Select two parents p_1 and p_2 from the current population
  @ Crossover the parents p_1 and p_2, obtaining the offspring o_1 and o_2
  @ Mutate the offspring o_1 and o_2
  if Fitness(the best offspring) is better than Fitness(the worst individual) **then**
    @ Replace the worst individual with the best offspring
  end if
end while
@ Output S as the best solution (individual) found

5.2. *MEP strengths and weaknesses*

5.2.1. *Strengths*

The output of a GP chromosome is usually provided by a fixed node. By contrast, MEP has a dynamic mechanism for selecting the gene which will provide the output. Namely, the best gene is chosen to represent the chromosome (by supplying the fitness of the individual). When more than one gene shares the best fitness, the first detected is chosen to represent the chromosome.

The dynamic-output chromosome has some advantages over the fixed-output chromosome especially when the complexity of the target expression is not known (see the numerical experiments). This feature also acts as a provider of variable-length expressions. Other techniques (such as GE or LGP) employ special genetic operators (which insert or remove chromosome parts) in order to achieve such a complex functionality.

The expression encoded in a MEP chromosome may have exponential length when the chromosome has polynomial length due to code reuse.

5.2.2. *Weaknesses*

There are problems where the complexity of the MEP decoding process is higher than the complexity of the GE, GEP, and LGP decoding processes. This situation usually arises when the set of training data is not a priori known (e.g., when game strategies are evolved).

5.3. *MEP on-line resources*

More information about MEP can be found in the following web pages:

- Mihai Oltean's home page `http://www.cs.ubbcluj.ro/~moltean` – last access June 6, 2008
- Crina Groşan's home page `http://www.cs.ubbcluj.ro/~cgrosan` – last access June 6, 2008

- Multi Expression Programming web page `http://www.mep.cs.ubbcluj.ro` – last access June 6, 2008
- MEP in C# `http://www.c-sharpcorner.com` – last access June 6, 2008

6. Grammatical Evolution

Grammatical Evolution(GE)[29,30,7] uses Backus - Naur Form (BNF)[31] in order to express computer programs. BNF is a notation that allows a computer program to be expressed as a grammar. GE chromosomes are binary strings of variable length. They are converted into integer strings and later into complex computer programs by using a grammar. This process is briefly depicted into Figure 2.

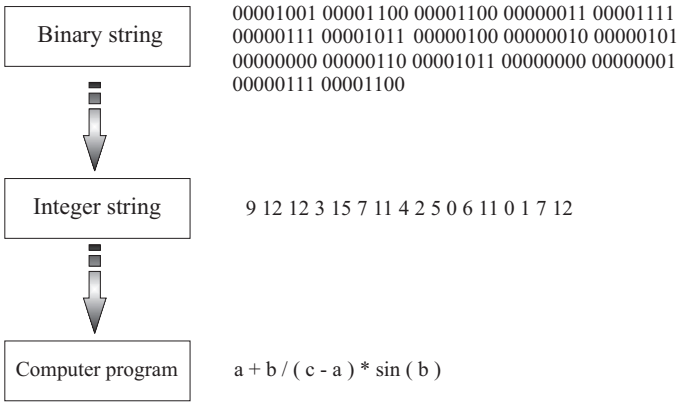

Fig. 2. A schematic view of the GE decoding process. A GE chromosome is represented as a binary string which is transformed into an integer string by grouping each set of 8 bits into a *codon*. In the final stage, the integer string is decoded into a complex computer program by using a BNF grammar.

GE is very similar to Genetic Algorithms for Deriving Software (GADS)[32]. The GADS genotype is a list of integers representing productions in a syntax. This is used to generate the phenotype, which is a program in the language defined by the syntax. Syntactically invalid phenotypes cannot be generated, though there may be phenotypes with residual nonterminals. GADS can be implemented on a traditional genetic algorithm.

6.1. *GE model*

6.1.1. *Representation*

Each GE individual is a variable length binary string that contains in its *codons* (groups of 8 bits) the necessary information for selecting a production rule from a BNF grammar.

A BNF grammar consists of a terminal and a non-terminal set of symbols, a set of production rules and a start symbol. Grammar symbols may be re-written, using

production rules, in other terminal and non-terminal symbols. An example from a BNF grammar is given by the following production rules[7,33]:

$$\begin{array}{lccc} S ::= & expr & (0) & | \\ & if\_stmt & (1) & | \\ & loop & (2) & \end{array}$$

These production rules state that the start symbol S can be replaced (re-written) by either one of the non-terminals: *expr*, *if-stmt*, or by *loop*.

The grammar is used in a generative process in order to construct a program by applying the production rules given by the genome, beginning with the starting symbol of the grammar.

In order to select a GE production rule, the next codon value on the genome is generated and used in the following formula:

$$Rule\_Index = Codon\_Value \bmod Num\_Rules.$$

If the next *Codon_Value* is 4, knowing that we have 3 rules to select from, as in the example above, we get $4 \bmod 3 = 1$. Therefore, S will be replaced with the non-terminal *if-stmt*, corresponding to the second production rule.

Beginning from the left side of the genome codon, integer values are generated and used for selecting rules from the BNF grammar, until one of the following situations arises:

(i) A complete program is generated. This occurs when all the non-terminals in the expression being mapped are turned into elements from the terminal set of the BNF grammar. Some genes might remain unexpressed in some cases.

(ii) The end of the genome is reached, in which case the *wrapping* operator is invoked. This means that the following evaluated *codon* will be the first one and the evaluation process is restarted from this position. The restarted process continues until a higher threshold that represents the maximum number of wrapping events has occurred during the individual mapping process.

In the case that a threshold on the number of wrapping events is exceeded and that the individual is still incompletely mapped, the mapping process is halted and the individual is assigned the worst possible fitness value.

Example. Consider the grammar: $G = \{N, T, S, P\}$, where the terminal set is: $T = \{+, -, *, /, \sin, (,)\}$ and the nonterminal symbols are: $N = \{expr,\ op_2,\ op_1\}$. The start symbol is $S = \langle expr \rangle$. The production rules P are:

$$
\begin{array}{rcll}
\langle expr \rangle ::= & a & (0) & | \\
& b & (1) & | \\
& c & (2) & | \\
& \langle expr \rangle \langle op_2 \rangle \langle expr \rangle & (3) & | \\
& (\langle expr \rangle \langle op_2 \rangle \langle expr \rangle) & (4) & | \\
& \langle op_1 \rangle \langle expr \rangle & (5) & \\
 & & & \\
\langle op_2 \rangle ::= & + & (0) & | \\
& - & (1) & | \\
& * & (2) & | \\
& / & (3) & \\
 & & & \\
\langle op_1 \rangle ::= & \sin & (0) &
\end{array}
$$

An example of a GE chromosome is the following:

$$
\begin{array}{rl}
C_{GE} = & (00001001\ \ 00001100\ \ 00001100\ \ 00000011\ \ 00001111\ \ 00000111 \\
& \ \ 00001011\ \ 00000100\ \ 00000010\ \ 00000101\ \ 00000000\ \ 00000110 \\
& \ \ 00001011\ \ 00000000\ \ 00000001\ \ 00000111\ \ 00001100)
\end{array}
$$

.

Translated into integer GE codons, the chromosome is: C^*_{GE} = (9 12 12 3 15 7 11 4 2 5 0 6 11 0 1 7 12). We can now start to translate this chromosome into a computer program.

The start symbol is $S = <expr>$. We have six possibilities (productions) to choose from. In order to make a choice we read the first gene of the chromosome, which is number 9. This number *modulo* the number of possibilities will give us the production to choose. In this case the choice is the fourth production (because the productions are indexed starting with 0): 9 *mod* 6 = 3 . We obtain a new string: $\langle expr \rangle \langle op_2 \rangle \langle expr \rangle$.

We start again with the first nonterminal symbol, $<expr>$, and we extract the second gene of the chromosome in order to see which production to choose. We have again 6 possibilities and the second gene has value 12. We choose production 12 *mod* 6 = 0 which is actually the first production. We obtain the string: $a \langle op_2 \rangle \langle expr \rangle$.

The first nonterminal symbol in this string is $\langle op_2 \rangle$. We have 4 productions for this symbol and the next gene has value 12. Thus we choose rule 12 *mod* 4 = 0, which indicates the first production for the nonterminal symbol $\langle op_2 \rangle$. The newly obtained string is: $a + \langle expr \rangle$.

The next nonterminal is $<expr>$ and we have 6 productions to choose from. According to the next gene we choose production 3 *mod* 6 = 3 which is the fourth production for the current nonterminal. We obtain the string: $a+\langle expr \rangle \langle op_2 \rangle \langle expr \rangle$.

Next nonterminal to expand is $<expr>$. We have 6 possibilities and the next gene which will decide the production has value 15. Thus, we choose the fourth production, 15 *mod* 6 = 3, and we obtain the string: $a+\langle expr \rangle \langle op_2 \rangle \langle expr \rangle \langle op_2 \rangle \langle expr \rangle$.

Now the next nonterminal symbol is $<expr>$ and next value of gene is 1. We choose production 1 mod 6 = 1 which is the second one and we obtain: $a + b \langle op_2 \rangle \langle expr \rangle \langle op_2 \rangle \langle expr \rangle$.

By applying the same reasoning we succesivelly obtain the following intermediate expressions:

$a + b/ \langle expr \rangle \langle op_2 \rangle \langle expr \rangle$
$a + b/(\langle expr \rangle \langle op_2 \rangle \langle expr \rangle) \langle op_2 \rangle \langle expr \rangle$
$a + b/(c \langle op_2 \rangle \langle expr \rangle) \langle op_2 \rangle \langle expr \rangle$
$a + b/(c - \langle expr \rangle) \langle op_2 \rangle \langle expr \rangle$
$a + b/(c - a) \langle op_2 \rangle \langle expr \rangle$
$a + b/(c - a) * \langle expr \rangle$
$a + b/(c - a) * \langle op_1 \rangle \langle expr \rangle$
$a + b/(c - a) * \sin \langle expr \rangle$

The last nonterminal symbol is $\langle expr \rangle$ and the next gene of the GE chromosome has value 1. In this case we have to choose the second production, for this nonterminal, which leads to the expression:

$$E = a + b/(c - a) * \sin(b)$$

Note that in some cases not all the GE genes are used. For instance, in our example we have two genes which were not used, because the translation process was ended after using the first 15 genes (no more nonterminals were available for expanding).

The obtained computer program depends on the BNF grammar used for translation. Different grammars (but with similar purposes) lead to different computer programs even if the GE chromosome employed is the same.

6.1.2. *Initialization*

GE chromosomes are binary strings that can be initialized without any restriction. For each position, we generate a random value (either 0 or 1).

Note that even if the chromosomes are correct they can generate invalid computer programs. However, this fact will be known only after decoding the entire chromosome.

6.1.3. *Genetic operators*

Genetic operators employed by GE are similar to those used in conjunction with binary encoding[29]. Other two operators: *duplicate* and *prune* have been tested within the GE system[7].

Crossover. Standard GE crossover is similar to the one cutting point crossover employed by the binary encoding. This operator implies the selection of two individuals (the parents). Then two cutting points (one in each individual) are randomly chosen. The segments on the right side of the cutting points are swapped[29].

Mutation. Mutation is performed as in the case of binary encoding[34]. This operator randomly flips some of the bits in a chromosome. Mutation can occur in any position in a chromosome with a small mutation probability.

Remark. Some algorithms also used as genetic operators duplicate and prune. By duplication, a randomly chosen sequence of genes is copied into the position of the last gene of the chromosome. The prune operator is usually applied for reducing the number of introns (unused genes[11]) and thus increasing the likelihood of beneficial crossover. Genes not used in the genotype-phenotype mapping process are discarded by the prune operator[7].

6.1.4. *Algorithm*

Standard GE algorithm uses steady-state evolutionary model[28] as its underlying mechanism. A generational algorithm was initially used[7], but due to its poor performance it was later replaced by a steady-state approach[33].

Algorithm 4 GE Algorithm

@ Randomly create the initial population ($P(0)$)
while not stop condition **do**
  @ Select two parents p_1 and p_2 from the current population
  @ Crossover the parents p_1 and p_2, obtaining the offspring o_1 and o_2
  @ Mutate the offspring o_1 and o_2
  if Fitness(the best offspring) is better than Fitness(the worst individual) **then**
    @ Replace the worst individual with the best offspring
  end if
end while
@ Output S as the best solution found

6.2. *GE strengths and weaknesses*

6.2.1. *Strengths*

The use of BNF grammars provides a general and a very natural way of evolving computer programs written in programming languages whose instructions may be

expressed as BNF rules. In the case of mathematical expressions, their representation is not limited to a single form (such as infix, prefix, postfix etc) as in the case of some other GP techniques. The representation can be simply changed by changing the grammar.

The wrapping operator provides a very original way of translating short chromosomes into very long expressions. Wrapping also provides an efficient way of avoiding invalid expressions.

6.2.2. *Weaknesses*

The GE mapping process also has some disadvantages. Wrapping may never end in some situations. For instance, consider the G_{GE} grammar defined earlier in the first example. In this case the chromosome $C'_{\mathrm{GE}} = 0, 0, 0, 0, 0$ cannot be translated into a valid expression because it does not contain operands. In order to prevent infinite cycling, a fixed number of wrapping occurrences is allowed. If this threshold is exceeded the expression obtained is incorrect and the corresponding individual is considered to be invalid. Several strategies for avoiding the generation of invalid computer programs due to infinite wrapping have been investigated in[35].

6.3. *GE on-line resources*

More information about Grammatical Evolution can be found on the following web pages:

- Grammatical Evolution web page, `http://www.grammatical-evolution.org` – last access June 6, 2008
- Conor Ryan's home page, `http://www.csis.ul.ie/staff/conorryan` – last access June 6, 2008
- Michael O'Neill's home page, `http://ncra.ucd.ie/members/oneillm.html` – last access June 6, 2008
- John James Collins's home page, `http://www.csis.ul.ie/staff/jjcollins` – last access June 6, 2008
- Maarten Keijzer's home page, `http://www.cs.vu.nl/~mkeijzer` – last access June 6, 2008
- Anthony Brabazon's home page `http://ncra.ucd.ie/members/brabazont.html` – last access June 6, 2008

7. Cartesian Genetic Programming

Cartesian Genetic Programming (CGP)[8] is a GP technique that encodes chromosomes in graph structures rather than trees like standard GP. The motivation for this representation is that the graphs are more general than the tree structures, thus allowing the construction of more complex computer programs[8].

Cartesian GP used a graph representation very similar to Poli's parallel distributed GP (PDGP)[36,37,38,39,40].

7.1. *CGP model*

7.1.1. *Representation*

CGP is Cartesian in the sense that the graph nodes are represented in a Cartesian coordinate system. This representation was chosen due to the node connection mechanism, which is similar to the GP mechanism. A CGP node contains a function symbol and pointers toward nodes representing function parameters. Each CGP node has an output that may be used as an input for another node.

A sketch of a CGP node is depicted in Figure 3. A CGP program is a set of interconnected nodes.

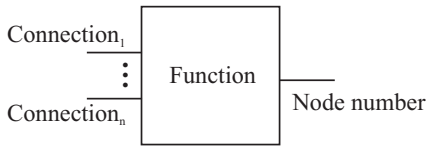

Fig. 3. A CGP node. *Node number* is the number (index) of the current node. The *Function* specifies the operation performed by the current node. $Connection_1$ ... $Connection_n$ are the indexes of the nodes providing input for the function of the current node. The function encoded by this node cannot have more than n arguments because the number of connections to each nodes is n. If the function has less arguments than this maximal value (e.g. there are 2 connections and the encoded function is *sin*) only the first argument(s) (starting with $Connection_1$) are taken into account.

Example. An example of a CGP program is depicted in Figure 4. This program is interpreted as follows: inputs labeled from 0 to 2 are depicted in the left side of the picture. The output of the program is provided by the output of node 13 (as shown in the right side of picture). In order to obtain the set of nodes, which are useful in this architecture, we have to start with node 13, which provides the result of this program. The input 0 and node 11 provide the arguments for the function encoded in this node. We move to node 11 whose inputs are provided by nodes 6 and 8. Inputs of node 6 are provided by nodes 1 and 4 while node 8 is connected to inputs 1 and 0. Node 4 is connected to inputs 2 and 0. In this way, we have obtained the set of nodes, which are used by the program encoded in the chromosome depicted in Figure 4. Not all the other nodes are used for computing the output of this program.

Each CGP program (graph) is defined by several parameters:

- number of rows - n_r,
- number of columns - n_c,
- number of inputs - n_i,
- number of outputs - n_o,
- number of functions - n_f,
- nodes interconnectivity - l.

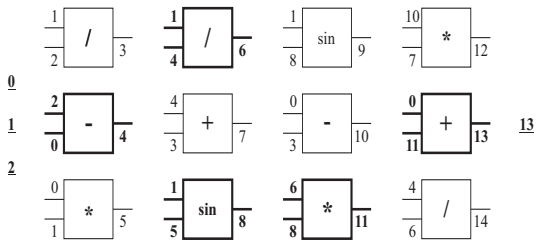

Fig. 4. A CGP program with 3x4 architecture. The program has 3 inputs, 1 output and 5 functions (+, −, ∗, /, sin). The inputs labeled from 0 to 2 are placed on the left side of the picture. The output of the program is provided by node 13 (given in the right side of the picture) which is also subject to evolution. The bold squares represent connected nodes which have influence to the output of the program. All the other nodes are unused.

The nodes interconnectivity is defined as being the number of the previous columns of cells that may have their outputs connected to a node in the current column (the primary inputs are treated as node outputs). This parameter is very important and defines how far two connected nodes can be in the matrix. If nodes' interconnectivity is equal to 1, each node can be connected only with nodes in the previous column. If nodes interconnectivity is equal to the number of rows, we can have a node connected to any other node in the previous columns.

The CGP chromosomes are represented, within the computer memory, as arrays of integer values. In order to achieve this we first need to label each function with an integer value ($+ = 0$, $- = 1$, $* = 2$, $/ = 3$, $sin = 4$) because we want to work with integer strings only.

The CGP chromosomes are encoded as strings by reading the graph columns top-down and printing the input nodes and the function symbol for each node. The index of the node (which is given in the bottom-right side of a node) is not printed because this information does not help the search process. Thus, for each node, we print the following information:

$Connection_1, Connection_2$ ... $Connection_n$ $Function\_Label$.

The CGP chromosome depicted in Figure 4 is encoded as:

$C = (1, 2, 3,$ **2,0,1,** $0, 1, 2,$ **1,4,3,** $4, 3, 0,$ **1,5,4,** $1, 8, 4,$ $0, 3, 1,$ **6,8,2**, $10, 7, 2,$ **0,11,0,** $4, 6, 3,$ $13)$

Nodes used by the program are written in bold, otherwise they are written in normal font. The last value in this array is the index of the node that provides the output of the program.

Genetic operators will modify these genes (providing the output) as all other genes.

7.1.2. *Initialization*

The genes of a CGP chromosome are randomly initialized, but they are subject to several constraints which ensure the generation of a valid chromosome.

7.1.3. *Genetic operators*

String genetic operators can be used within the CGP system. Nodes supplying the outputs of the entire program (see the right side of Figure 4) are not fixed as they may also be subject to genetic operators.

Mutation. The mutation operator requires some special conditions (see the initialization restrictions) to be met. By mutation, some symbols in the current CGP chromosome are modified. Mutation consists of randomly changing one of the integer values in the CGP string of the genotype. Only one value is changed per mutation. A mutation can modify the node function, the node input or the connections in a gene. Since only a part of the genotype is decoded into the phenotype, mutations often do not affect the behavior of a cell.

Remark. Although the crossover operator was not used by the CGP technique, it may be applied without any restrictions.

7.1.4. *Algorithm*

The CGP algorithm suggested in[8] is a simple (1+λ)-Evolution Strategy[41] where λ is usually 4. The algorithm may be described as follows:

Algorithm 5 CGP Algorithm

@ Generate a random solution S.
while not stop_condition **do**
  @ Generate λ solutions by mutating S
  @ Replace S by the best individual out of the currently existing (1+λ) individuals
end while
@ Output S as the best solution (individual) found

7.2. *CGP strengths and weaknesses*

7.2.1. *Strengths*

Evolving the indexes of the cells which will provide the output for the problem is an interesting feature which introduces further variety in the chromosome.

In standard GP[11] the evolved program has only one output. In CGP it is possible to have as many outputs as necessary.

7.2.2. *Weaknesses*

The inappropriate choice for the number of columns required by the CGP chromosome might lead to poor results. For instance if the number of columns is set to 1 we

can have only simple solutions (made up of a single function with its arguments). This is why the number of columns should be big enough.

7.3. *CGP on-line resources*

More information about CGP can be found on the following web pages:

- Julian. F. Miller's home page `http://www.elec.york.ac.uk/intsys/users/jfm7/` – last access June 6, 2008
- Lukás Sekanina's home page `http://www.fit.vutbr.cz/~sekanina/` – last access June 6, 2008

8. Stack-Based Genetic Programming

Stack-based Genetic programming, introduced by Perkis in[9], represents programs as lists of nodes of functions or terminals that consume their inputs from a stack and place their outputs on another stack. These implementations, including the early work of Bruce[42], Stoffel[43] and later Spector[44], do not try to preserve the stack correctness of the individuals in the population, but rather rely on the evaluation framework to identify any stack underflow or overflow. In contrast, in GP with stack-correct (Forth) crossover, introduced by Tchernev in[45] and[46,47], the crossover operators manipulate the post order representation of the program tree. Because the crossover points are chosen to have compatible stack depths, no malformation is possible. If the initial population is stack-correct (no individuals have underflow, and the final stack depth equals the desired number of outputs), it is guaranteed that all individuals produced by using stack-correct crossover will be stack-correct.

8.1. *SBGP model*

8.1.1. *Representation*

The programs from a stack-GP system are LISP *S*-expressions that contain terminals and functions. Unlike the standard GP[11], in stack-based GP the functions receive arguments from a numerical stack and return their result by pushing it on the stack. Function calls are not nested: programs consist of flat linear sequences of functions and terminals.

Terminals are simply a class of functions which push preset variables onto the stack when they are executed.

In stack-GP one additional type of closure constraint must be imposed on the functions. If the stack does not contain sufficient elements for applying one of function (with other words, if the stack deep is less than the function arity), then it will do nothing. In the previous example, the operator + from the third position of the chromosome will not be executed because the stack contains only one argument.

For instance, the following is an example of a program which encodes the mathematical expression $a + b/(c - a) * sin(b)$.

C = ((sin) (**a**) (+) (**b**) (**c**) (**a**) (**-**) (/) (**b**) (***sin***) (∗) (+))

Chromosome										Stack
$\underline{sin}\,a$	$+$	b	c	a	$-$	$/$	b	$\sin *$	$+$	
$\sin \underline{a}$	$+$	b	c	a	$-$	$/$	b	$\sin *$	$+$	a
$\sin a$	$\underline{+}$	b	c	a	$-$	$/$	b	$\sin *$	$+$	a
$\sin a$	$+$	$\underline{b}$	c	a	$-$	$/$	b	$\sin *$	$+$	a, b
$\sin a$	$+$	b	$\underline{c}$	a	$-$	$/$	b	$\sin *$	$+$	a, b, c
$\sin a$	$+$	b	c	$\underline{a}$	$-$	$/$	b	$\sin *$	$+$	a, b, c, a
$\sin a$	$+$	b	c	a	$\underline{-}$	$/$	b	$\sin *$	$+$	a, b, $c-a$
$\sin a$	$+$	b	c	a	$-$	$\underline{/}$	b	$\sin *$	$+$	a, $b/(c-a)$
$\sin a$	$+$	b	c	a	$-$	$/$	$\underline{b}$	$\sin *$	$+$	a, $b/(c-a)$, b
$\sin a$	$+$	b	c	a	$-$	$/$	b	$\underline{sin}*$	$+$	a, $b/(c-a)$, $\sin(b)$
$\sin a$	$+$	b	c	a	$-$	$/$	b	$\sin \underline{*}$	$+$	a, $b/(c-a)*\sin(b)$
$\sin a$	$+$	b	c	a	$-$	$/$	b	$\sin *$	$\underline{+}$	$a+b/(c-a)*\sin(b)$

The stack is protected from underflow by this constraint; and stack overflow has not been a problem, and has been limited in practice so far by specifying a maximum allowable program length[9].

Numerical calculations are performed in Reverse Polish Notation (RPN)[48,49]. The advantage of RPN in this context is that the parse tree for the calculation is expressed simply by the order of the functions and terminals in the sequence and not by a constrained syntax demanding balanced parentheses.

8.1.2. *Initialization*

Initial individuals are just random sequences of symbols from the function set chosen for the problem: due to the nature of RPN, this is sufficient to generate program parse trees of varied shapes and depths.

8.1.3. *Genetic operators*

Since there are no syntactical constraints on the critter sequences, the genetic operators applied on the stack-GP chromosome could be any of those used in a traditional GA. These string-based genetic operators are applied directly on the linear programs. The safety of the resulting programs was guaranteed by specifying that all the functions take their arguments from the stack. The function calls that occur with too few items on the stack are ignored, doing nothing[43].

Therefore, a two-cutting point crossover can be utilized: two points are randomly picked in each of the parents, and one child sequence is created by inserting the

sequence enclosed by the points in the father into the space defined by the cutting-points in the mother chromosome[9].

In addition to crossover, one point mutation, which consists of changing some one function call in the sequence to some other function in the current function set, can be applied because the stack-GP chromosomes can be treated as strings.

8.2. *SBGP strengths and weaknesses*

8.2.1. *Strengths*

The stack is free to accumulate "junk" without effecting fitness: each program contains, along with the code that determines the final result, "introns", code sequences that perform calculations which create that stack junk.

The genetic operations of Perkis's model are performed directly on the linear program (e.g. string-based crossover). All the functions take their arguments from the stack. When there are no sufficient items on the stack, the function will do nothing. This scheme guaranties the safety of the resulting programs and determines lower computational efforts than were required using traditional S-expression-based genetic programming[9].

8.2.2. *Weaknesses*

In the stack-GP system there is no mechanism for allowing branching program execution. There are many problems for which branching execution is not necessary. But in many problems of planning and strategy the side effects of functions are of primary importance, and the actual sequence of the program execution is of interest rather than the final calculated result[42].

8.3. *SBGP on-line resources*

- Lee Spector's home page `http://helios.hampshire.edu/lspector/` – last access June 6, 2008
- Samuel Landau's home page `http://samuel.landau.free.fr/index.php.en` – last access June 6, 2008
- Sebastien Picault's home page `http://www2.lifl.fr/~picault/index.html` – last access June 6, 2008

9. Applications

A set of problems where a particular GP technique (LGP, GEP, MEP, GE, CGP or SBGP) has been applied are presented in Tables 1, 2 and 3.

A key question is which method should be selected when a problem has to be solved. This is difficult to answer, since all GP variants can be applied to the same problems. We will still make a comparative discussion over ability to solve problems with the considered methods:

- Ease of implementation. The source code of LGP, GEP, MEP, CGP and SBGP are simpler compared to GE because GE needs a grammar interpreter for computing the fitness of an individual. However, if a grammar interpreter is available (like that one for mathematical expressions, available on the GE website (see section 6.3)), the source code of GE is very simple being actually like a steady-state GA with binary encoding.
- Invalid individuals. By allowing invalid individuals, the system will spend processor time for performing useless computations. The only GP variant which allows invalid individuals is GE.
- Low level implementation. By making a machine code implementation we can get some significant improvements regarding the speed. All techniques can be implemented at low level, however some of them (LGP, MEP and CGP) have a special structure for chromosomes which - if implemented in machine-code - don't need an interpreter to be evaluated.
- Additional parameters. Some methods require the setting of some special parameters (expecting population size, number of generations, code length, probabilities for applying genetic operators) to work well: LGP needs to specify the number of additional registers, multigenic GEP requires to set a linking symbol to combine genes. Setting these parameters incorrectly might lead to poor results. Additional experiments are required for setting these parameters correctly.
- Availability of the source code. According to author's investigations the code for all methods is freely available on the internet. Note that for some methods, the source code might be available only in some particular languages and not for solving any kind of problems. This is why some active involvement of the users is required in almost all cases.

10. Conclusions and further work

Several Genetic Programming variants having linear representation have been reviewed in this paper. A complete description has been given for each method. Individual representation, genetic operators, the main algorithm and other particular features have been thoroughly deeply analyzed. For each method we have presented a set of strengths and weaknesses. This section could be useful helping the reader to select the appropriate method for a specific application. A list of problems where these techniques have been applied was also given. Moreover, a set of internet sites where the reader can find more information about the methods has been given in the appropriate sections. These sites belong to either the authors of the methods or to other researchers who have performed a long-term research in that area.

Further work directions will be focused on:

- Comparing all GP variants (not only those with linear representation). As soon as new variants are invented, the comparison should be updated.

Table 1. Applications of various GP techniques (1).

	Domain	LGP	GEP	MEP	GE	CGP	SBGP
digital circuits, evolvable hardware, robotics	solving Boolean functions (digital circuits)	14,16,50	5	22,23,51,52 53,54,55,56	57	58,59,60,61 62,63,64	65,83
	robot control and design	66,67,68,69			70	71	
	automatic synthesis of micro-controller assembly code	72					
	electronic hardware fault monitoring	73	73	74			
	modeling the equivalent circuit for electrochemical impedance spectroscopy		75				
	design of antena			76			
regression, prediction and classification	symbolic regression for artificially constructed problems	13,14,16,21,77,95	5,21,78,79,80	21,23	7,21,29,81,82	8,21,40,77	43,65,83,84
	modeling electricity demand prediction	85					
	predicting the amount of gas emitted from coal face		86				
	prediction for artificially constructed problems	13		87		88	89
	classification	13	19,90,91,92,93	5	21,40,94,137		95
neural and other networks	designing neural networks		96		97	98	
	Petri net modeling of high-order genetic systems				99		
documents and images processing tasks	compression of images and sound	100					
	partitioning of images	16					95
	visual learning method	101					
	construct sentence ranking functions for text summarization		102,103				
	designing fractal curves				104		
	image filter design					105,106	
	image processing tasks		107		108	109	
	natural language recognisers					39	

Table 2. Applications of various GP techniques (2).

	Domain	LGP	GEP	MEP	GE	CGP	SBGP
bioinformatics	protein localization	110,111					
	controlling complex pharmacogenomic systems	112,113					
	evolution of developmental program in a cell for creating multi-cellular organisms					114	
	eukaryotic promoter recognition				115		
medical	mining formula-syndrome relation in traditional Chinese medicine		116				
	classification of fetal heart rate	13			117		
	automated detection of breast cancer	118					
economical	business intelligence from web usage mining	119					
	financial modeling	120		120,121	122,123,124		
	time-series modeling		125				
	corporate failure prediction				126		
	modeling the corporate bond-issuer credit rating process				126		
	classifiers for modeling the relationship between strategy and corporate performance				127		
	anticipating bankruptcy reorganization from raw financial data				128		
	diagnostic corporate stability				129		
	bond-issuer credit rating				130		
	evolving trading rules for spot foreign-exchange markets				131		
	constant generation for the financial domain				132		
	credit classification				94		
	adaptive trading				133		

Table 3. Applications of various GP techniques (3).

	Domain	LGP	GEP	MEP	GE	CGP	SBGP
others	data analysis for real world problems	4,14,134	135	21		136,137	
	simulation model of a waste incinerator	138					
	intrusion detection	139		140			
	web usage mining	141					
	evolution of concurrent control software	142					
	discrimination of unexploded ordonance from clutter	143					
	predict a wastewater treatment plant's effluent concentration	138					
	hybrid multi-agent framework	144					145,146
	decision support systems	147	147	147			
	block stacking		5				148
	traveling salesman problem		149				
	function mining		150,151,152				
	back-calculation of pavement layer thickness		153				
	modeling the deflection basin of flexible highway pavements		154				
	plan on short path avoiding obstructions		155				
	solving Fredholm first kind integral equations		156				
	knowledge discovery		157				
	evolving evolutionary algorithms	158		6,159			
	evolving play strategies for Nim game			160			
	evolving heuristics for traveling salesman problem			161			
	solving trigonometric identities				162		
	Santa Fe ant trail				29,81,82	8	47
	generation of caching algorithms				163		
	composition of music				164		
	mastermind game				165		
	complex systems regulation				166		
	generate hashing functions				167		
	evolve digital surfaces				168		
	0/1 multi-constrained knapsack problem				169		
	function estimation				170		
	solving differential equations				171		
	portrait painter programs					172	
	post docking filtering					173	

- Including, in comparison, other minor GP variants (such as Traceless Genetic Programming[174,175]), which were not discussed here due to their limited representation in the literature.
- Involving in comparison the variants of the methods investigated in this paper. Some methods, like GE and SBGP have several variants used for some particular problems. These variants have some features which makes them good candidates in particular cases.
- Performing a numerical comparison of the considered methods. This would be very interesting, but in the same time very difficult to achieve. The main problem is related to the fairness of the comparison. The standard way to compare 2 methods is to use the same parameters for both of them. However, minor parameters (such as probabilities for applying various operators) can affect the quality of the results. A possibility to make the comparison fair is to find the best parameter settings for each method. In this way no other discussions over the quality of the results would be possible. However, running each method with its best parameters implies hard work and a lot of experiments.

Acknowledgments

The authors thank to anonymous reviewers for their useful suggestions. This research was supported by grant IDEI-543 from CNCSIS.

References

1. M. Brameier ,"On linear genetic programming,", PhD Thesis, Universitat Dortmund, Germany, 2003.
2. M. Brameier and W. Banzhaf, *Linear Genetic Programming*. No. XVI in Genetic and Evolutionary Computation, Springer, 2007.
3. M. Zhang, "Improving object detection performance with genetic programming," *IJAIT*, vol. 16, no. 5, pp. 849–873, 2007.
4. M. Brameier and W. Banzhaf, "A comparison of linear genetic programming and neural networks in medical data mining," *IEEE-EC*, vol. 5, no. 1, pp. 17–26, 2001.
5. C. Ferreira, "Gene expression programming: a new adaptive algorithm for solving problems," *Complex Systems*, vol. 13, no. 2, pp. 87–129, 2001.
6. M. Oltean and C. Groşan, "Evolving evolutionary algorithms using multi expression programming," in *ECAL* (W. Banzhaf (et al.) eds.), vol. 2801 of LNAI, pp. 651–658, Springer, 2003.
7. C. Ryan, J. J. Collins, and M. O'Neill, "Grammatical evolution: Evolving programs for an arbitrary language," in *European Workshop on Genetic Programming* (W. Banzhaf (et al.) eds.), vol. 1391 of LNCS, pp. 83–95, Springer, 1998.
8. J. F. Miller and P. Thomson, "Cartesian genetic programming," in *EuroGP* (R. Poli (et al.) eds.), vol. 1802 of LNCS, pp. 121–132, Springer, 2000.
9. T. Perkis, "Stack-based genetic programming," in *IEEE WCCI*, vol. 1, pp. 148–153, IEEE Press, 1994.
10. J. R. Koza and R. Poli, "Genetic programming," in *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques* (E. K. Burke and G. Kendall, eds.), ch. 5, Springer, 2005.

11. J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.
12. J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.
13. M. Brameier and W. Banzhaf, "Evolving teams of predictors with linear genetic programming," *Genetic Programming and Evolvable Machines*, vol. 2, no. 4, pp. 381–407, 2001.
14. M. Brameier and W. Banzhaf, "Explicit control of diversity and effective variation distance in linear genetic programming," in *EuroGP* (A. G. B. Tettamanzi (et al.) eds.), vol. 2278 of LNCS, pp. 37–49, Springer, 2002.
15. P. Nordin, W. Banzhaf, and F. D. Francone, "Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover," in *Advances in Genetic Programming 3* (L. Spector (et al.) eds.), ch. 12, pp. 275–299, MIT Press, 1999.
16. P. Nordin, F. Hoffmann, F. D. Francone, M. Brameier, and W. Banzhaf, "AIM-GP and parallelism," in *CEC* (P. J. Angeline (et al.) eds.), vol. 2, pp. 1059–1066, IEEE Press, 1999.
17. P. Nordin, "A compiling genetic programming system that directly manipulates the machine code," in *Advances in Genetic Programming* (K. E. Kinnear, Jr., ed.), ch. 14, pp. 311–332, MIT Press, 1994.
18. N. L. Cramer, "A representation for the adaptive generation of simple sequential programs," in *ICGA*, Carnegie Mellon University, 1985.
19. C. Ferreira, "Discovery of the Boolean functions to the best density-classification rules using gene expression programming," in *EuroGP* (A. G. B. Tettamanzi (et al.) eds.), vol. 2278 of LNCS, pp. 50–59, Springer, 2002.
20. M. Oltean, "Improving the search by encoding multiple solutions in a chromosome," in *Evolutionary Machine Design: Methodology and Applications* (N. Nedjah and L. de Macedo Mourelle, eds.), ch. 4, pp. 85–110, Nova Publishers, 2005.
21. M. Oltean and C. Groşan, "A comparison of several linear genetic programming techniques," *Complex Systems*, vol. 14, no. 4, pp. 285–313, 2004.
22. M. Oltean and C. Groşan, "Evolving digital circuits using multi expression programming," in *NASA/DoD Conference on Evolvable Hardware* (R. S. Zebulum (et al.) eds.), pp. 87–90, IEEE CS Press, 2004.
23. M. Oltean, "A-Brain: a general system for solving data analysis problems," *J. Exp. Theor. Artif. Intell.*, vol. 19, no. 4, pp. 333–353, 2007.
24. V. A. Alfred, S. Ravi, and D. U. Jeffrey, *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.
25. D. H. Wolpert and W. G. Macready, "No free lunch theorems for search," Tech. Rep. SFI-TR-95-02-010, Santa Fe Institute, 1995.
26. D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.
27. R. Bellman, *Dynamic Programming*. Princeton University Press, 1957.
28. G. Syswerda, "A study of reproduction in generational and steady state genetic algorithms," in *FOGA* (G. J. E. Rawlins, ed.), pp. 94–101, Morgan Kaufmann, 1991.
29. M. O'Neill and C. Ryan, "Under the hood of grammatical evolution," in *GECCO* (W. Banzhaf (et al.) eds.), vol. 2, pp. 1143–1148, Morgan Kaufmann, 1999.
30. M. O'Neill and C. Ryan, *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, vol. 4 of *Genetic programming*. Kluwer Academic Publishers, 2003.
31. J. Backus, "Programming language semantics and closed applicative languages,"

ACM Symp. on the Principles of Programming Languages, pp. 71–86, 1973.
32. N. R. Paterson and M. Livesey, "Distinguishing genotype and phenotype in genetic programming," in *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University, 1996* (J. R. Koza, ed.), pp. 141–150, Stanford Bookstore, 1996.
33. C. Ryan and M. O'Neill, "Grammatical evolution: A steady state approach," in *Late Breaking Papers at the Genetic Programming Conference* (J. R. Koza, ed.), Stanford University Bookstore, 1998.
34. M. Mitchell, *An Introduction to Genetic Algorithms*. The MIT Press, 1996.
35. C. Ryan, M. Keijzer, and M. Nicolau, "On the avoidance of fruitless wraps in grammatical evolution," in *GECCO* (E. Cantú-Paz (et al.) eds.), vol. 2724 of *LNCS*, pp. 1752–1763, Springer, 2003.
36. R. Poli, "Evolution of recursive transition networks for natural language recognition with parallel distributed genetic programming," Tech. Rep. CSRP-96-19, University of Birmingham, School of Computer Science, 1996.
37. R. Poli, "Evolution of graph-like programs with parallel distributed genetic programming," in *Genetic Algorithms: Proceedings of the Seventh International Conference* (T. Back, ed.), pp. 346–353, Morgan Kaufmann, 1997.
38. R. Poli, "Discovery of symbolic, neuro-symbolic and neural networks with parallel distributed genetic programming," in *ICANNGA97* (G. D. Smith (et al.) eds.), Springer, 1997.
39. R. Poli, "Evolution of recursive transition networks for natural language recognition with parallel distributed genetic programming," in *Evolutionary Computing* (D. Corne and J. L. Shapiro, eds.), vol. 1305 of *LNCS*, pp. 163–177, Springer, 1997.
40. R. Poli, "Parallel distributed genetic programming," in *New Ideas in Optimization* (D. Corne, M. Dorigo, and F. Glover, eds.), Advanced Topics in Computer Science, ch. 27, pp. 403–431, McGraw-Hill, 1999.
41. H.-G. Beyer and H.-P. Schwefel, "Evolution strategies: a comprehensive introduction," *Natural Computing*, vol. 1, no. 1, pp. 3–52, 2002.
42. W. S. Bruce, "The lawnmower problem revisited: Stack-based genetic programming and automatically defined functions," in *Genetic Programming 1997: Proceedings of the Second Annual Conference* (J. R. Koza (et al.) eds.), pp. 52–57, Morgan Kaufmann, 13-16 1997.
43. K. Stoffel and L. Spector, "High-performance, parallel, stack-based genetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference* (J. R. Koza (et al.) eds.), pp. 224–229, MIT Press, 1996.
44. L. Spector and A. J. Robinson, "Genetic programming and autoconstructive evolution with the push programming language," *Genetic Programming and Evolvable Machines*, vol. 3, no. 1, pp. 7–40, 2002.
45. E. Tchernev, "Forth crossover is not a macromutation?," in *Genetic Programming 1998: Proceedings of the Third Annual Conference* (J. R. Koza (et al.) eds.), pp. 381–386, Morgan Kaufmann, 1998.
46. E. B. Tchernev, "Stack-correct crossover methods in genetic programming," in *Late Breaking papers at GECCO-2002* (E. Cantú-Paz, ed.), pp. 443–449, AAAI, 2002.
47. E. B. Tchernev and D. S. Phatak, "Control structures in linear and stack-based genetic programming," in *Late Breaking Papers at GECCO 2004* (M. Keijzer, ed.), 2004.
48. C. L. Hamblin, "Translation to and from polish notation," *The Computer Journal*, vol. 5, no. 3, pp. 210–213, 1962.
49. C. L. Hamblin, "Computer languages," *Australian Computer Journal*, vol. 17, no. 4,

pp. 195–198, 1985.
50. W. Banzhaf and A. Leier, "Evolution on neutral networks in genetic programming," in *Genetic Programming Theory and Practice III* (T. Yu (et al.) eds.), vol. 9 of *Genetic Programming*, ch. 14, pp. 207–221, Springer, 2005.
51. F. Z. Hadjam, C. Moraga, and M. Benmohamed, "Cluster-based evolutionary design of digital circuits using all improved multi-expression programming," in *Late breaking paper at (GECCO'2007)* (P. A. N. Bosman, ed.), pp. 2475–2482, ACM Press, 2007.
52. F. Z. Hadjam, C. Moraga, and L. Hildebrand, "Evolutionary design of digital circuits using improved multi-expression programming,", Research Report 812, Faculty of Informatics, University of Dortmund, Germany, 2007.
53. M. Oltean, "Solving even-parity problems using multi expression programming," in *Proceedings of Joint Conference on Information Sciences* (K. Chen (et al), eds.), vol. 1, pp. 315–318, Association for Intelligent Machinery, 2003.
54. M. Oltean, "Improving multi expression programming: An ascending trail from sea-level even-3-parity problem to alpine even-18-parity problem," in *Evolvable Machines: Theory & Practice* (N. Nedjah and L. de Macedo Mourelle, eds.), ch. 10, pp. 229–256, Springer, 2004.
55. M. Oltean, "Evolving reversible circuits for the even-parity problem," in *EvoWorkshops: Applications of Evolutionary Computing* (F. Rothlauf (et al.) eds.), vol. 3449 of LNCS, pp. 225–234, Springer, 2005.
56. M. Oltean, C. Groşan, and M. Oltean, "Evolving digital circuits for the knapsack problem," in *ICCS* (M. Bubak (et al.) eds.), vol. 3038 of *LNCS*, pp. 1257–1264, Springer, 2004.
57. U. R. Karpuzcu, "Automatic verilog code generation through grammatical evolution," in *GECCO 2005* (F. Rothlauf, ed.), pp. 394–397, ACM, 2005.
58. M. Collins, "Finding needles in haystacks is harder with neutrality," in *GECCO* (H.-G. Beyer and U.-M. O'Reilly, eds.), pp. 1613–1618, ACM, 2005.
59. J. Miller, "What bloat? Cartesian Genetic Programming on Boolean problems," in *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers* (E. D. Goodman, ed.), pp. 295–302, 2001.
60. J. F. Miller, "An empirical study of the efficiency of learning Boolean functions using a cartesian genetic programming approach," in *(GECCO)* (W. Banzhaf (et al.) eds.), vol. 2, pp. 1135–1142, Morgan Kaufmann, 1999.
61. J. F. Miller, D. Job, and V. K. Vassilev, "Principles in the evolutionary design of digital circuits—part I," *Genetic Programming and Evolvable Machines*, vol. 1, no. 1–2, pp. 7–35, 2000.
62. J. F. Miller, P. Thomson, and T. Fogarty, "Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study," in *Genetic Algorithms and Evolution Strategy in Engineering and Computer Science* (D. Quagliarella (et al.) eds.), pp. 105–131, John Wiley and Sons, 1998.
63. L. Sekanina, "Evolutionary design of gate-level polymorphic digital circuits," in *EvoWorkshops: Applications of Evolutionary Computing* (F. Rothlauf (et al.) eds.), vol. 3449 of *LNCS*, pp. 185–194, Springer, 2005.
64. T. Yu and J. Miller, "Neutrality and the evolvability of Boolean function landscape," in *EuroGP'2001* (J. F. Miller (et al.) eds.), vol. 2038 of *LNCS*, pp. 204–217, Springer, 2001.
65. R. Crawford-Marks and L. Spector, "Size control via size fair genetic operators in the PushGP genetic programming system," in *GECCO 2002* (W. B. Langdon (et al.) eds.), pp. 733–739, Morgan Kaufmann, 2002.
66. W. Banzhaf, P. Nordin, and M. Olmer, "Generating adaptive behavior for a real

robot using function regression within genetic programming," in *Annual Conference on Genetic Programming* (J. R. Koza (et al.) eds.), pp. 35–43, Morgan Kaufmann, 1997.
67. P. Nordin and W. Banzhaf, "An on-line method to evolve behavior and to control a miniature robot in real time with genetic programming," *Adaptive Behavior*, vol. 5, no. 2, pp. 107–140, 1996.
68. P. Nordin, W. Banzhaf, and M. Brameier, "Evolution of a world model for a miniature robot using genetic programming," *Robotics and Autonomous Systems*, vol. 25, no. 1-2, pp. 105–116, 1998.
69. K. Wolff and P. Nordin, "Learning biped locomotion from first principles on a simulated humanoid robot using linear genetic programming.," in *GECCO* (M. Ades and L. M. Deschaine eds.), vol. 2723 of *LNCS*, pp. 495–506, Springer, 2003.
70. M. ONeill and J. Collins, "Automatic generation of generic robot behaviours using grammatical evolution," in *AROB5* (C. Ryan and J. Buckley, eds.), pp. 21–29, Limerick University Press, 2000.
71. S. Harding and J. F. Miller, "Evolution of robot controller using cartesian genetic programming," in *EuroGP* (M. Keijzer (et al.) eds.), vol. 3447 of *LNCS*, pp. 62–73, Springer, 2005.
72. D. Mota Dias, M. A. C. Pacheco, and J. F. M. Amaral, "Automatic synthesis of microcontroller assembly code through linear genetic programming," in *Genetic Systems Programming: Theory and Experiences* (N. Nedjah, A. Abraham, and L. de Macedo Mourelle, eds.), pp. 195–234, Springer, 2006.
73. A. Abraham and C. Groşan, "Automatic programming methodologies for electronic hardware fault monitoring," *Journal of Universal Computer Science*, vol. 12, no. 4, pp. 408–431, 2006.
74. A. Abraham and C. Groşan, "Genetic programming approach for fault modeling of electronic hardware," in *CEC* (D. Corne (et al.) eds.), vol. 2, pp. 1563–1569, IEEE Press, 2005.
75. H. Cao, J. Yu, and L. Kang, "An evolutionary approach for modeling the equivalent circuit for electrochemical impedance spectroscopy," in *CEC* (R. Sarker (et al.) eds.), pp. 1819–1825, IEEE Press, 2003.
76. J. Braunstein, H.-S. Kim, S. Kahng, and S.-H. Ha, "A multi-expression programming application to the design of planar antennae," in *Electromagnetic Field Computation, 2006 12th Biennial IEEE Conference on*, pp. 123–123, 2006.
77. G. C. Wilson and W. Banzhaf, "A comparison of cartesian genetic programming and linear genetic programming," in *EuroGP 2008* (M. O'Neill (et al.) eds.), vol. 4971 of *LNCS*, pp. 182–193, Springer, 2008.
78. E. Bautu, A. Bautu, and H. Luchian, "Adagep - an adaptive gene expression programming algorithm," in *SYNASC'05*, pp. 403–406, IEEE Computer Society, 2005.
79. E. Bautu, A. Bautu, and H. Luchian, "Symbolic regression on noisy data with genetic and gene expression programming," in *SYNASC'05*, pp. 321–324, IEEE Computer Society, 2005.
80. H. Yun Quan and G. Yang, "Gene expression programming with DAG chromosome," in *ISICA 2007* (L. Kang (et al.) eds.), vol. 4683 of *LNCS*, pp. 271–275, Springer, 2007.
81. M. O'Neill and C. Ryan, "Genetic code degeneracy: Implications for grammatical evolution and beyond," in *ECAL* (D. Floreano (et al.) eds.), vol. 1674 of *LNAI*, pp. 149–153, Springer, 1999.
82. M. O'Neill, C. Ryan, M. Keijzer, and M. Cattolico, "Crossover in grammatical evolution: The search continues," in *EuroGP* (J. F. Miller (et al.) eds.), vol. 2038 of *LNCS*, pp. 337–347, Springer, 2001.

83. P. L. Lanzi, "XCS with stack-based genetic programming," in *CEC2003* (R. Sarker (et al.) eds.), pp. 1186–1191, IEEE Press, 2003.
84. L. Spector, "Adaptive populations of endogenously diversifying pushpop organisms are reliably diverse," in *rtificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems* (R. K. Standish (et al.) eds.), pp. 142–145, MIT Press, 2002.
85. M. Bhattacharya, A. Abraham, and B. Nath, "A linear genetic programming approach for modeling electricity demand prediction Victoria," in *HIS* (A. Abraham and M. Köppen, eds.), Advances in Soft Computing, pp. 379–393, Physica, 2001.
86. Q. Li, Z. Cai, L. Zhu, and Y. Zhao, "Application of gene expression programming in predicting the amount of gas emitted from coal face," *Journal of Basic Science and Engineering*, vol. 12, no. 1, pp. 49–54, 2004.
87. A. Baykasoglu and L. Ozbakir, "MEPAR-miner: Multi-expression programming for classification rule mining," *European Journal of Operational Research*, vol. 183, no. 2, pp. 767–784, 2007.
88. J. A. Walker and J. F. Miller, "Predicting prime numbers using cartesian genetic programming," in *EuroGP* (M. Ebner (et al.) eds.), vol. 4445 of *LNCS*, pp. 205–216, Springer, 2007.
89. M. Defoin-Platel, M. Chami, M. Clergue, and P. Collard, "Teams of genetic predictors for inverse problem solving," in *EuroGP* (M. Keijzer (et al.) eds.), vol. 3447 of *LNCS*, pp. 341–350, Springer, 2005.
90. S. W. Wilson, "Classifier conditions using gene expression programming," tech. rep., IlliGAL Report No. 2008001, University of Illinois at Urbana-Champaign, USA, 2008.
91. C. Zhou, W. Xiao, T. M. Tirpak, and P. C. Nelson, "Evolving accurate and compact classification rules with gene expression programming," *IEEE TEC*, vol. 7, no. 6, pp. 519–531, 2003.
92. M. H. Marghny and I. E. El-Semman, "Exact logical classification rules with gene expression programming; microarray case study," in *AIML* (H. Elmahdy, ed.), 2005.
93. W. Wang, Q. Li, and Z. Cai, "Finding compact classification rules with parsimonious gene expression programming," in *ICNN&B* (M. Zhao and Z. Shi, eds.), vol. 2, pp. 702–705, IEEE Press, 2005.
94. A. Brabazon and M. O'Neill, "Credit classification using grammatical evolution," *Informatica*, vol. 30, no. 3, pp. 325–335, 2006.
95. K. Holladay, K. Robbins, and J. von Ronne, "FIFTH: A stack based GP language for vector processing," in *EuroGP* (M. Ebner (et al.) eds.), vol. 4445 of *LNCS*, Springer, 2007.
96. C. Ferreira, "Designing neural networks using gene expression programming," in *Online World Conference on Soft Computing in Industrial Applications* (A. Abraham and M. Köppen, eds.), 2004.
97. I. Tsoulos, D. Gavrilis, and E. Glavas, "Neural network construction using grammatical evolution," in *IEEE ISSPIT*, pp. 827– 831, IEEE Press, 2005.
98. B. Sharma, "Cartesian genetic programming for evolving neural networks: Application in clinical data analysis," Master's thesis, University of Birmingham, 2002.
99. J. H. Moore and L. W. Hahn, "Petri net modeling of high-order genetic systems using grammatical evolution," *BioSystems*, vol. 72, no. 1-2, pp. 177–86, 2003.
100. P. Nordin and W. Banzhaf, "Programmatic compression of images and sound," in *Annual Conference on Genetic Programming* (J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, eds.), pp. 345–350, MIT Press, 1996.
101. K. Krawiec and B. Bhanu, "Coevolutionary computation for synthesis of recognition systems," in *Proceedings of IEEE Workshop on Learning in Computer Vision and*

Pattern Recognition, vol. 6, 2003.
102. J. Z. Xie, "Machine learning in automatic text summarization: From extracting to abstracting," tech. rep., University of Illinois at Chicago, 2005.
103. Z. Xie, X. Li, B. Di Eugenio, W. Xiao, T. M. Tirpak, and P. C. Nelson, "Using gene expression programming to construct sentence ranking functions for text summarization," in *COLING*, pp. 1381–1384, 2004.
104. A. Ortega, A. Dalhoum, and M. Alfonseca, "Using grammatical evolution to design curves with a given fractal dimension," in *ICEIS* (O. Camp(et al.) eds.), pp. 395–398, ICEIS Press, 2003.
105. L. Sekanina, "Image filter design with evolvable hardware," in *EvoWorkshops 2002, Applications of Evolutionary Computing* (S. Cagnoni (et al.) eds.), vol. 2279 of LNCS, pp. 255–266, Springer, 2002.
106. L. Sekanina and R. Ruzicka, "Easily testable image operators: The class of circuits where evolution beats engineers," in *NASA/DoD Conference on Evolvable Hardware*, pp. 135–144, IEEE CS Press, 2003.
107. W. Zhu and H. Timmermans, "Exploring heuristics underlying pedestrian shopping decision processes an application of gene expression programming," in *Innovations in Design and Decision Support Systems in Architecture and Urban Planning. Part 2* (J. P. Van Leeuwen and H. J. P. Timmermans, eds.), pp. 121–136, Springer, 2007.
108. M. O'Driscoll, S. McKenna, and J. J. Collins, "Synthesising edge detectors with grammatical evolution," in *GECCO 2002* (A. M. Barry, ed.), pp. 137–140, AAAI, 2002.
109. H. A. Montes and J. L. Wyatt, "Cartesian genetic programming for image processing tasks," in *IASTED ICNNCI*, pp. 185–190, IASTED/ACTA Press, 2003.
110. A. Heddad, M. Brameier, and R. MacCallum, "Evolving regular expression-based sequence classifiers for protein nuclear localisation," in *EvoWorkshops: Applications of Evolutionary Computing* (G. R. Raidl (et al.), eds.), vol. 3005 of LNCS, pp. 31–40, Springer, 2004.
111. W. B. Langdon and W. Banzhaf, "Repeated sequences in linear genetic programming genomes," *Complex Systems*, vol. 15, no. 4, pp. 285–306, 2005.
112. A. Floares, "Computation intelligence tools for modeling and controlling pharmacogenomic systems: Genetic programming and neural networks," in *IEEE WCCI 2006*, (G. G. Yen (et al.) eds.), pp. 7510–7517, IEEE Press, 2006.
113. A. Floares, "Genetic programming and neural networks feedback linearization for modeling and controlling complex pharmacogenomic systems," in *WILF 2005* (I. Bloch (et al.) eds.), vol. 3849 of LNCS, pp. 178–187, Springer, 2005.
114. J. F. Miller, "Evolving developmental programs for adaptation, morphogenesis, and self-repair," in *ECAL* (W. Banzhaf (et al.) eds.), vol. 2801 of *LNAI*, pp. 256–265, Springer, 2003.
115. M. O'Neill, C. Adley, and A. Brabazon, "A grammatical evolution approach to eukaryotic promoter recognition," in *Bioinformatics Inform Workshop and Symposium*, 2005.
116. X. U, C.-J. Tang, H. Zhang, S. Qiao, Y. Jiang, J. Liu, and P. Han, "Mining formula-syndrome relationship in traditional chinese medicinewith gene expression programming," *Computer Applications*, vol. 25, no. 11, pp. 2679–2680, 2005.
117. D. Gavrilis and I. Tsoulos, "Classification of fetal heart rate using grammatical evolution," in *IEEE Workshop on Signal Processing Systems Design and Implementation*, pp. 425–429, 2005.
118. W. Sheta, N. Eltonsy, G. Tourassi, and A. Elmaghraby, "Automated detection of breast cancer from screening mammograms using genetic programming," *IJICIS*,

vol. 5, no. 1, pp. 1–10, 2005.
119. A. Abraham, "Natural computation for business intelligence from web usage mining," in *SYNASC*, pp. 3–10, IEEE Computer Society Press, 2005.
120. C. Groşan and A. Abraham, "Stock market modeling using genetic programming ensembles," in *Genetic Systems Programming: Theory and Experiences* (N. Nedjah, A. Abraham, and L. de Macedo Mourelle, eds.), vol. 13 of *Studies in Computational Intelligence*, pp. 133–148, Springer, 2006.
121. C. Groşan, A. Abraham, S. Y. Han, and V. Ramos, "Stock market prediction using multi expression programming," in *Portuguese Conference on Artificial Intelligence, Workshop on Artificial Life and Evolutionary Algorithms* (A. C. C. Bento and G. Dias, eds.), vol. 13 of *Studies in Computational Intelligence*, pp. 73–78, IEEE Press, 2005.
122. M. O'Neill and A. Brabazon, "Recent advances in grammatical evolution: the opportunities for financial modeling," in *Proceedings of International Conference on Numerical Methods for Finance*, 2006.
123. M. O'Neill, A. Brabazon, C. Ryan, and J. J. Collins, "Evolving market index trading rules using grammatical evolution," in *EvoWorkshops: Applications of Evolutionary Computing* (E. J. W. Boers (et al.) eds.), vol. 2037 of *LNCS*, pp. 343–352, Springer, 2001.
124. M. O'Neill, A. Brabazon, C. Ryan, and J. J. Collins, "Developing a market timing system using grammatical evolution," in *GECCO* (L. Spector (et al.) eds.), pp. 1375–1381, Morgan Kaufmann, 2001.
125. H. Lopes and W. Weinert, "A gene-expression programming system for time-series modeling," 2004.
126. A. Brabazon, M. O'Neill, R. Matthews, and C. Ryan, "Grammatical evolution and corporate failure prediction," in *GECCO* (W. B. Langdon (et al.), ed.), pp. 1011–1018, Morgan Kaufmann, 2002.
127. A. Brabazon, M. O'Neill, C. Ryan, and R. Matthews, "Evolving classifiers to model the relationship between strategy and corporate performance using grammatical evolution," in *EuroGP* (J. A. Foster (et al.) eds.), vol. 2278 of *LNCS*, pp. 103–112, Springer, 2002.
128. A. Brabazon and M. O'Neill, "Anticipating bankruptcy reorganisation from raw financial data using grammatical evolution," in *EvoWorkshops: Applications of Evolutionary Computing* (G. R. Raidl (et al.), eds.), vol. 2611 of LNCS, pp. 368–377, Springer, 2003.
129. A. Brabazon and M. O'Neill, "Diagnosing corporate stability using grammatical evolution," *International Journal of Applied Mathematics and Computer Science*, vol. 14, no. 3, pp. 317–333, 2004.
130. A. Brabazon and M. O'Neill, "Bond-issuer credit rating with grammatical evolution," in *EvoWorkshops: Applications of Evolutionary Computing* (G. R. RRaidl (et al.), ed.), vol. 3005 of *LNCS*, pp. 270–279, Springer, 2004.
131. A. Brabazon and M. O'Neill, "Evolving technical trading rules for spot foreign-exchange markets using grammatical evolution," *Computational Management Science*, vol. 1, no. 3–4, pp. 311–327, 2004.
132. I. Dempsey, "Constant generation for the financial domain using grammatical evolution," in *GECCO* (F. R. (et al.), ed.), pp. 350–353, ACM, 2005.
133. I. Dempsey, M. O'Neill, and A. Brabazon, "Adaptive trading with grammatical evolution," in *CEC*, pp. 2587–2592, IEEE Press, 2006.
134. L. M. Deschain, "Tackling real world environmental engineering challenges with linear genetic programming," *PCAI Magazine*, vol. 15, no. 5, pp. 35–37, 2000.

135. K. Eldrandaly and A. Negm, "Performance evaluation of gene expression programming for hydraulic data mining," *International Arab Journal of Information Technology*, vol. 5, no. 2, pp. 126–131, 2008.
136. J. A. Walker and J. F. Miller, "Solving real-valued optimisation problems using Cartesian Genetic Programming," in *GECCO '07* (D. Thierens (et al.) eds.), vol. 2, pp. 1724–1730, ACM Press, 2007.
137. J. A. Walker and J. F. Miller, "Changing the genospace: Solving GA problems with cartesian genetic programming," in *EuroGP* (M. Ebner (et al.) eds.), vol. 4445 of LNCS, pp. 261–270, Springer, 2007.
138. L. M. Deschain, J. J. Patel, R. D. Guthrie, J. T. Grimski, and M. J. Ades, "Using linear genetic programming to develop a C/C++ simulation model of a waste incinerator," in *Proceedings of Advanced Technology Simulation Conference* (M. Ades, ed.), 2001.
139. D. Song, M. I. Heywood, and A. N. Zincir-Heywood, "A linear genetic programming approach to intrusion detection," in *GECCO* (E. Cantú-Paz (et al.), eds.), vol. 2724 of LNCS, pp. 2325–2336, Springer, 2003.
140. A. Abraham and C. Groşan, "Evolving intrusion detection systems," in *Genetic Systems Programming: Theory and Experiences* (N. Nedjah, A. Abraham, and L. de Macedo Mourelle, eds.), vol. 13 of *Studies in Computational Intelligence*, pp. 57–80, Springer, 2006. Forthcoming.
141. A. Abraham and V. Ramos, "Web usage mining using artificial ant colony clustering and genetic programming," in *CEC* (R. Sarker (et al.) eds.), pp. 1384–1391, IEEE Press, 2003.
142. J. Hart and M. Shepperd, "The evolution of concurrent control software using genetic programming," in *EuroGP* (M. Keijzer (et al.) eds.), vol. 3003 of LNCS, pp. 289–298, Springer, 2004.
143. F. D. Francone, L. M. Deschaine, T. Battenhouse, and J. J. Warren, "Discrimination of unexploded ordnance from clutter using linear genetic programming," in *Genetic Programming Theory and Practice III* (T. Yu (et al.) eds.), vol. 9 of *Genetic Programming*, ch. 4, pp. 49–64, Springer, 2005.
144. S. Mukkamala, A. H. Sung, and A. Abraham, "Hybrid multi agent framework for detection of stealthy probes,," *Applied Soft Computing Journal*, vol. 5, no. 3, pp. 631 641, 2007.
145. L. Spector, J. Klein, C. Perry, and M. Feinstein, "Emergence of collective behavior in evolving populations of flying agents," *Genetic Programming and Evolvable Machines*, vol. 6, pp. 111–125, 2005.
146. L. Spector and A. Robinson, "Multi-type, self-adaptive genetic programming as an agent creation tool," in *GECCO 2002* (A. M. Barry, ed.), pp. 73–80, AAAI, 2002.
147. A. Abraham, C. Groşan, T. Cong, and J. Lakhmi, "A concurrent neural network - genetic programming model for decision support systems," in *ICKM* (S. Hawamdeh, ed.), pp. 231–245, World Scientific, 2005.
148. L. Spector, J. Klein, and M. Keijzer, "The push3 execution stack and the evolution of control," in *GECCO 2005* (H.-G. Beyer (et al.) eds.), vol. 2, pp. 1689–1696, ACM Press, 2005.
149. C. Ferreira, "Combinatorial optimization by gene expression programming: Inversion revisited," in *Proceedings of Argentine Symposium on Artificial Intelligence* (J. M. Santos and A. Zapico, eds.), pp. 160–174, 2002.
150. D. Jiang, Z. Wu, and L. Kang, "Parameter identifications in differential equations by gene expression programming," in *ICNC 2007*, pp. 644–648, IEEE Computer Society, 2007.

151. Y. Liu, J. English, and E. Pohl, "Application of gene expression programming in the reliability of consecutive-k-out-of-n: F systems with identical component reliabilities," in *ICIC 2007*, (A. D.-S. Huang (et al.) eds.), pp. 217–224, Springer, 2007.
152. C. Tang, L. Duan, J. Peng, H. Zhang, and Y. Zhong, "The strategies to improve performance of function mining by gene expression programming-genetic modifying, overlapped gene, backtracking and adaptive mutation," in *Proceedings of DBSJ Annual Conference: DEWJ*, 2006.
153. M. Saltan and S. Terzi, "Comparative analysis of using artificial neural networks (ANN) and gene expression programming (GEP) in back calculation of pavement layer thickness," *Indian Journal of Engineering and Materials Sciences*, vol. 12, no. 1, pp. 42–50, 2005.
154. S. Terzi, "Modeling the deflection basin of flexible highway pavements by gene expression programming," *Journal of Applied Sciences*, vol. 5, no. 2, pp. 309–314, 2005.
155. H.-y. Shi and G.-m. Dai, "Plan on short path avoiding obstructions based on gene expression programming," *Application Research of Computers*, vol. 22, no. 11, pp. 82–84, 2005.
156. E. Bautu, A. Bautu, and H. Luchian, "A gep-based approach for solving fredholm first kind integral equations," in *SYNASC*, pp. 325–328, IEEE Computer Society, 2005.
157. C. Tang, J. J. Peng, H. Zhang, and Y. Zhong, "Three new techniques for knowledge discover by gene expression programming–transgene, overlapped gene expression and backtracking evolution," *Journal of Computer Applications*, vol. 25, pp. 1978–1981, 2005.
158. M. Oltean, "Evolving evolutionary algorithms using linear genetic programming," *Evolutionary Computation*, vol. 13, pp. 387–410, 2005.
159. M. Oltean, "Evolving evolutionary algorithms with patterns," *Soft Computing*, vol. 11, no. 6, pp. 503–518, 2006.
160. M. Oltean, "Evolving winning strategies for nim-like games," in *World Computer Congress - Student Forum (IFIP)* (M. Kaâniche, ed.), pp. 353–364, Kluwer, 2004.
161. M. Oltean and D. Dumitrescu, "Evolving TSP heuristics using multi expression programming," in *ICCS* (M. Bubak (et al.) eds.), vol. 3037 of LNCS, pp. 670–673, Springer, 2004.
162. C. Ryan, M. O'Neill, and J. J. Collins, "Grammatical evolution: Solving trigonometric identities," in *International Mendel Conference on Genetic Algorithms, Optimisation Problems, Fuzzy Logic, Neural Networks, Rough Sets.*, pp. 111–119, Technical University of Brno, 1998.
163. M. O'Neill and C. Ryan, "Automatic generation of caching algorithms," in *Proceedings of Evolutionary Algorithms in Engineering and Computer Science* (K. Miettinen (et al.) eds.), pp. 127–134, John Wiley & Sons, 1999.
164. A. Ortega, R. S. Alfonso, and M. Alfonseca, "Automatic composition of music by means of grammatical evolution," in *APL*, vol. 32, pp. 148–155, ACM Press, 2002.
165. C. Ryan, M. Nicolau, and M. O'Neill, "Genetic algorithms using grammatical evolution," in *EuroGP* (J. A. Foster (et al.) eds.), vol. 2278 of *LNCS*, pp. 278–287, Springer, 2002.
166. S. Amarteifio and M. O'Neill, "An evolutionary approach to complex system regulation using grammatical evolution," in *International Conference on the Simulation and Synthesis of Living Systems* (J. Pollack (et al.) eds.), pp. 551–556, The MIT Press, 2004.
167. P. Berarducci, D. Jordan, D. Martin, and J. Seitzer, "GEVOSH: Using grammatical evolution to generate hashing functions," in *GECCO* (R. Poli (et al.) eds.), vol. 3102

of LNCS, pp. 31–39, Springer, 2004.
168. M. Hemberg and U.-M. O'Reilly, "Extending grammatical evolution to evolve digital surfaces with genr8," in *EuroGP* (M. Keijzer (et al.) eds.), vol. 3003 of *LNCS*, pp. 299–308, Springer, 2004.
169. R. Cleary and M. O'Neill, "An attribute grammar decoder for the 01 multiconstrained knapsack problem," in *EvoCOP* (G. R. Raidl and J. Gottlieb, eds.), vol. 3448 of *LNCS*, pp. 34–35, Springer, 2005.
170. I. G. Tsoulos, D. Gavrilis, and E. Dermatas, "GDF: A tool for function estimation through grammatical evolution," *Computer Physics Communications*, vol. 174, no. 7, pp. 555–559, 2006.
171. I. G. Tsoulos and I. E. Lagaris, "Solving differential equations with genetic programming," *Genetic Programming and Evolvable Machines*, vol. 7, no. 1, pp. 33–54, 2007.
172. S. DiPaola, "Evolving creative portrait painter programs using darwinian techniques with an automatic fitness function," in *Electronic Imaging & Visual Arts*, 2005.
173. A. Garmendia-Doval, S. Morley, and S. Juhos, "Post docking filtering using cartesian genetic programming," in *ICAE* (P. Liardet (et al.) eds.), vol. 2936 of *LNCS*, pp. 189–200, Springer, 2003.
174. M. Oltean, "Solving Even-Parity Problems using Traceless Genetic Programming,", *CEC*, (G. Greenwood (et al.) eds.), pp. 1813–1819, IEEE Press, 2004.
175. M. Oltean, C. Groşan, "Solving Multiobjective Optimization Problems using Traceless Genetic Programming,", *Journal of theoretical and experimental artificial intelligence*, vol. 19, pp. 227–248, 2007.