A Comparison of Several Linear Genetic Programming Techniques

Mihai Oltean[*]
Crina Groşan[†]
Department of Computer Science,
Faculty of Mathematics and Computer Science,
Babes-Bolyai University, Kogalniceanu 1,
Cluj-Napoca, 3400, Romania

A comparison between four Genetic Programming techniques is presented in this paper. The compared methods are Multi-Expression Programming, Gene Expression Programming, Grammatical Evolution, and Linear Genetic Programming. The comparison includes all aspects of the considered evolutionary algorithms: individual representation, fitness assignment, genetic operators, and evolutionary scheme. Several numerical experiments using five benchmarking problems are carried out. Two test problems are taken from PROBEN1 and contain real-world data. The results reveal that Multi-Expression Programming has the best overall behavior for the considered test problems, closely followed by Linear Genetic Programming.

1. Introduction

Genetic Programming (GP) [1, 2] is an evolutionary technique used for breeding a population of computer programs. GP individuals are represented and manipulated as nonlinear entities, usually trees. A particular GP subdomain consists of evolving mathematical expressions. In that case the evolved program is a mathematical expression, program execution means evaluating that expression, and the output of the program is usually the value of the expression.

Several linear variants of GP have recently been proposed. Some of them are: Multi-Expression Programming (MEP)[1] [3, 4], Grammatical Evolution (GE) [5, 6], Gene Expression Programming (GEP) [7], Linear Genetic Programming (LGP) [8, 9], Cartesian Genetic Programming (CGP) [10], and Genetic Algorithm for Deriving Software (GADS) [11].

All mentioned GP variants make a clear distinction between the genotype and the phenotype of an individual [12]. Thus, the individuals are

[*]Electronic mail address: moltean@nessie.cs.ubbcluj.ro.
[†]Electronic mail address: cgrosan@nessie.cs.ubbcluj.ro.
[1]MEP source code is available at www.mep.cs.ubbcluj.ro.

represented as linear entities (strings) that are decoded and expressed like nonlinear entities (trees).

In this paper, a systematic comparison of four GP techniques for solving symbolic regression problems is provided. The compared methods are MEP, GEP, GE, and LGP. The comparison includes all aspects of the considered evolutionary techniques: individual representation, genetic operators, fitness assignment, exception handling, and evolutionary scheme.

Several numerical experiments using MEP, GEP, GE, and LGP are carried out. Five difficult test problems are used for comparing the methods. Three test problems are artificially constructed and two test problems contain real-world data that have been taken from PROBEN1 [13].

The relationship between the success rate and the population size and the number of generations is analyzed for the artificially constructed problems.

For the real-world problems the mean of the absolute differences between the expected output value and the value obtained by the best individual over 100 runs is reported.

The results obtained reveal that MEP has the best overall behavior on the considered test problems, closely followed by LGP.

The paper is organized as follows. The representation of MEP, LGP, GE, and GEP individuals is described in section 2. Advantages and shortcomings of each representation are described in detail in this section. Section 3 discusses the selection strategies used in conjunction with the considered algorithms. Genetic operators employed by each algorithm are presented in section 4. Section 5 describes the evolutionary schemes used by each algorithm. The formulae used for computing the fitness of each individual are presented in section 6. The mechanisms used for handling the exceptions are presented in section 7. Several numerical experiments are performed in section 8.

2. Individual representation

In this section, individual representations are compared. GEP, GE, and LGP encode a single solution, while MEP encodes multiple solutions in a chromosome.

GEP, MEP, and LGP use integer and real numbers for individual encoding while GE uses binary strings for encoding rules of a Backus–Naur form grammar.

A GEP chromosome consists of several genes linked by the symbols $+$ or $*$. A GEP gene has the same functionality as a MEP, LGP, or GE chromosome.

2.1 Multi-Expression Programming

MEP representation [3, 4] is similar to the way in which C and *Pascal* compilers translate mathematical expressions into machine code [14].

MEP genes are substrings of a variable length. The chromosome length is constant and equal to the number of genes in that chromosome. Each gene encodes a terminal or a function symbol. A gene encoding a function includes pointers towards the function arguments. Function parameters always have indices of lower values than the position of that function in the chromosome.

According to this representation scheme, the first symbol of the chromosome must be a terminal symbol.

Example

An example of a C_{MEP} chromosome is given below. Numbers to the left stand for gene labels, or memory addresses. Labels do not belong to the chromosome. They are provided only for explanatory purposes.

$$\begin{array}{ll} 1: & a \\ 2: & b \\ 3: & +1,2 \\ 4: & c \\ 5: & d \\ 6: & +4,5 \end{array}$$

When MEP individuals are translated into computer programs (expressions) they are read top-down starting with the first position. A terminal symbol specifies a simple expression. A function symbol specifies a complex expression (made up by linking the operands specified by the argument positions with the current function symbol).

For instance, genes 1, 2, 4, and 5 in the previous example encode simple expressions composed of a single terminal symbol. The expressions associated with genes 1, 2, 4, and 5 are:

$$E_1 = a$$
$$E_2 = b$$
$$E_4 = c$$
$$E_5 = d.$$

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

$$E_3 = a + b.$$

Gene 6 indicates the operation + on the operands located in positions 4 and 5. Therefore gene 6 encodes the expression:

$$E_6 = c + d.$$

There is neither practical nor theoretical evidence that one of these expressions is better than the others. Moreover, Wolpert and Macready

[15] proved that we cannot use the search algorithm's behavior so far, for a particular test function, to predict its future behavior on that function. This is why each MEP chromosome is allowed to encode a number of expressions equal to the chromosome length (number of genes).

The expression associated with each position is obtained by reading the chromosome bottom-up from the current position and following the links provided by the function pointers.

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

$$\text{Number of Symbols} = (N + 1)\text{Number of Genes} - N,$$

where N is the number of arguments of the function symbol with the greatest number of arguments.

2.1.1 Multi-Expression Programming strengths

A GP chromosome generally encodes a single expression (computer program). By contrast, a MEP chromosome encodes several expressions (it allows representing multiple solutions). The best of the encoded expressions 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.

When solving symbolic regression problems, the MEP chromosome decoding process has the same complexity as other techniques such as GE and GEP (see [3] and section 6.5 this paper).

The Multi-Expression chromosome has some advantages over the Single-Expression 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) to achieve such a complex functionality.

The expression encoded in a MEP chromosome may have exponential length when the chromosome has polynomial length. For instance the expression:

$$E = \underbrace{a * a * \cdots * a}_{2^n},$$

can be encoded by the following chromosome (it is assumed that the terminal set is $T = \{a\}$ and the function set is $F = \{+, -, *, /\}$):

$$\begin{array}{ll} 1: & a \\ 2: & *1,1 \\ 3: & *2,2 \\ \ldots & \\ n: & *n-1, n-1. \end{array}$$

Thus, an expression of exponential length (2^n symbols) is encoded in a chromosome of polynomial length ($3n - 2$ symbols). This is possible by repeatedly using the same subexpression in a larger (sub)expression (code-reuse).

The code-reuse ability is not employed by the GEP technique but it is used by LGP. The code-reuse ability is similar to Automatically Defined Functions (ADFs) [1, 2], a mechanism of GP.

2.1.2 Multi-Expression Programming weaknesses

If code-reuse ability is not utilized, the number of symbols in a MEP chromosome is usually three times greater than the number of symbols in a GEP or GE chromosome encoding the same expression.

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).

2.2 Gene Expression Programming

GEP [7] uses linear chromosomes that store expressions in breadth-first form. A GEP gene is a string of terminal and function symbols. GEP genes are composed of a *head* and a *tail*. The head contains both function and terminal symbols. The tail may contain terminal symbols only.

For each problem the head length (denoted h) is chosen by the user. The tail length (denoted by t) is evaluated by:

$$t = (n - 1)h + 1,$$

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

Let us consider a gene made up of symbols in the sets F and T:

$$F = \{*, /, +, -\}.$$
$$T = \{a, b\}.$$

In this case $n = 2$. If we choose $h = 10$, then we get $t = 11$, and the length of the gene is $10 + 11 = 21$. Such a gene is given below:

$$C_{\text{GEP}} = + * ab - +aab + ababbbababb.$$

The *expression* encoded by the gene C_{GEP} is:

$$E = a + b * ((a + b) - a).$$

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.

2.2.1 Gene Expression Programming strengths

The separation of the GEP chromosome in two parts (head and tail), each of them containing specific symbols, provides an original and very efficient way of encoding syntactically correct computer programs.

2.2.2 Gene Expression Programming weaknesses

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 which is discovered. But, if all the operators $\{+, -, *, /\}$ 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 can be seen in [7], the success rate of GEP increases with the number of genes in the chromosome. But, 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 arises usually in systems that employ chromosomes with a fixed length.

2.3 Grammatical Evolution

GE [5, 6] uses the Backus–Naur form (BNF) to express computer programs. BNF is a notation that allows a computer program to be expressed as a grammar.

A BNF grammar consists of terminal and nonterminal symbols. Grammar symbols may be rewritten in other terminal and nonterminal symbols.

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

An example from a BNF grammar is given by the following production rules:

$$
\begin{aligned}
S ::= {} & \text{expr}| \ (0) \\
& \text{if-stmt}| \ (1) \\
& \text{loop.} \ (2)
\end{aligned}
$$

These production rules state that the start symbol S can be replaced (rewritten) either by one of the nonterminals (expr or if-stmt), or by loop.

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

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

$$\text{Rule} = \text{Codon\_Value } \textbf{MOD} \text{ Num\_Rules}.$$

If the next Codon integer value is four, knowing that we have three rules to select from, as in the example above, we get 4 **MOD** 3 = 1.

Therefore, *S* will be replaced with the nonterminal 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.

1. A complete program is generated. This occurs when all the nonterminals in the expression being mapped are turned into elements from the terminal set of the BNF grammar.
2. The end of the genome is reached, in which case the *wrapping* operator is invoked. This results in the return of the genome reading frame to the left side of the genome once again. The reading of the codons will then continue unless a higher threshold representing the maximum number of wrapping events has occurred during this individual mapping process.

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

Example

Consider the grammar:

$$G = \{N, T, S, P\},$$

where the terminal set is:

$$T = \{+, -, *, /, \sin, \exp, (,)\},$$

and the nonterminal symbols are:

$$N = \{\text{expr}, \text{op}, \text{pre\_op}\}.$$

The start symbol is:

$$S = \text{<expr>}.$$

The production rules *P* are:

```
<expr> ::= <expr><op><expr>| (0)
           (<expr><op><expr>)| (1)
           <pre_op>(<expr>)| (2)
           <var>. (3)
```

$$
\begin{aligned}
\text{<op>} ::= &+ | \ (0)\\
&- | \ (1)\\
&* | \ (2)\\
&/. \ (3)\\
\text{<pre\_op>} ::= &\sin | \ (0)\\
&\exp . \ (1)
\end{aligned}
$$

Here is an example of a GE chromosome:

$$C_{\text{GE}} = 000000000000001000000001000000110000001000000011.$$

Translated into GE codons, the chromosome is:

$$C_{\text{GE}} = 0, 2, 1, 3, 2, 3.$$

This chromosome is translated into the expression:

$$E = \exp(x) * x.$$

2.3.1 Grammatical Evolution strengths

Using the BNF grammars for specifying a chromosome provides a natural way of evolving programs written in programming languages whose instructions may be expressed as BNF rules.

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.

2.3.2 Grammatical Evolution 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 these conditions the chromosome

$$C'_{\text{GE}} = 0, 0, 0, 0, 0$$

cannot be translated into a valid expression because it does not contain operands. 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.

Since the debate regarding the supremacy of binary encoding over integer encoding has not finished yet, we cannot say which one is better. However, as the translation from binary representations to integer/real representations takes some time we suspect that the GE system is a little slower than other GP techniques that use integer representation.

2.4 Linear Genetic Programming

LGP [8, 9] uses a specific linear representation of computer programs. Instead of the tree-based GP expressions of a functional programming language (like *LISP*) programs of an imperative language (like *C*) are evolved.

A LGP individual is represented by a variable-length sequence of simple *C* language instructions. Instructions operate on one or two indexed variables (registers) r, or on constants c from predefined sets. The result is assigned to a destination register, for example, $r_i = r_j * c$.

Here is an example LGP program.

```
void LGP(double v[8])
{
v[0] = v[5] + 73;
v[7] = v[3] – 59;
if (v[1] > 0)
if (v[5] > 21)
v[4] = v[2] * v[1];
v[2] = v[5] + v[4];
v[6] = v[7] * 25;
v[6] = v[4] – 4;
v[1] = sin(v[6]);
if (v[0] > v[1])
v[3] = v[5] * v[5];
v[7] = v[6] * 2;
v[5] = v[7] + 115;
if (v[1] <= v[6])
v[1] = sin(v[7]);
}
```

A LGP can be turned into a functional representation by successive replacements of variables starting with the last effective instruction.

The maximum number of symbols in a LGP chromosome is four times the number of instructions.

2.4.1 Linear Genetic Programming strengths

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

2.4.2 Linear Genetic Programming weaknesses

An important LGP parameter is the number of registers used by a chromosome. The number of registers 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 (see [1] and section 8 of this paper). In that case we have to use several supplementary registers. The number of supplementary registers depends on the complexity of the expression being discovered. An inappropriate choice can have disastrous effects on the program being evolved.

2.5 Discussion on individual representation

One of the most common problems that arises in a GP system is "bloat" [16], that is, the tendency of GP individuals to grow in size without increasing in quality [17]. Bloat slows down the evolutionary search and takes a lot of memory, making unprotected GP systems useless after only a few generations. Several mechanisms for preventing bloat have been proposed. Two of them are: maximal depth restriction [1] and parsimony pressure [17].

To avoid bloat, MEP and GEP use fixed-length chromosomes. LGP uses variable-size chromosomes that are limited to a maximum number of instructions (genes). Standard GE does not use any mechanism to prevent bloat. However, in order to provide a fair comparison, in all of the experiments performed in this paper, the size of the GE chromosomes has been limited to the same number of symbols employed by the MEP, GEP, and LGP techniques.

3. Selection

Several selection strategies (e.g., binary tournament or roulette wheel) have been tested with the considered techniques. GEP has been tested [7] with a special selection strategy that reduces the selection range as the search process advances.

However, in order to provide a fair comparison, in all experiments performed in this paper MEP, GE, LGP, and GEP use the same binary tournament selection strategy.

4. Genetic operators

In this section, genetic operators used with MEP, GEP, GE, and LGP are described. By applying specific genetic operators MEP, GEP, and LGP offspring are always syntactically correct expressions (computer programs). GE offspring may sometimes encode invalid individuals that are generated by an incomplete mapping process.

4.1 Multi-Expression Programming

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

4.1.1 Recombination

By recombination, two parents exchange genetic materials in order to obtain two offspring. Several variants of recombination have been considered and tested within our MEP implementation: one-point recombination, two-point recombination, and uniform recombination. One-point recombination is used in the experiments performed in this paper. By applying the recombination operator, one crossover point is randomly chosen and the parents exchange the sequences after the crossover point.

4.1.2 Mutation

Every MEP gene may be subject to mutation. The first gene of a chromosome must encode a terminal symbol in order to preserve the consistency of the chromosome. There is no restriction in symbols changing for other genes.

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 positions indicating the function arguments are also generated by mutation.

If the current gene encodes a function, the former may be mutated into a terminal symbol or into another function (function symbol and pointers towards arguments).

4.2 Gene Expression Programming

Chromosomes are modified by mutation, transposition, root transposition, gene transposition, gene recombination, one-point recombination, and two-point recombination. A detailed description of the GEP genetic operators can be found in [7]. The one-point recombination and point mutation operators are the only ones used and described in this paper.

4.2.1 Recombination

The one-point recombination operator in the GEP representation is analogous to the corresponding binary representation operator. Two parents and one cut-point are chosen. Two offspring are obtained from the parents, by exchanging genetic material according to the cut-point.

4.2.2 Mutation

Any symbol may be changed with any other symbol in the head of the chromosome. Terminal symbols may only be changed into other terminal symbols in the chromosome's tail.

4.3 Grammatical Evolution

Standard binary genetic operators (point mutation and two-point crossover) are used with GE [5]. GE also makes use of a *duplication* operator that duplicates a random number of codons and inserts them into the penultimate codon position on the genome.

4.4 Linear Genetic Programming

The variation operators are crossover and mutation.

4.4.1 Recombination

LGP uses two-point string crossover [8]. A segment of random position and random length is selected in both parents and exchanged between them. If one of the resulting offspring would exceed the maximum length, crossover is abandoned and restarted by exchanging equally-sized segments.

4.4.2 Mutation

An operand or an operator of an instruction is changed by mutation into another symbol over the same set.

LGP also employs a special kind of mutation (called *macro mutation*) which deletes or inserts an entire instruction.

5. Evolutionary scheme

In this section, the evolutionary algorithms employed by MEP, GEP, GE, and LGP are described. All the considered algorithms start with a randomly chosen population of individuals. Each individual in the current population is evaluated by using a fitness function that depends on the problem being solved.

5.1 Multi-Expression Programming

MEP uses a steady-state [18] evolutionary scheme. The initial population is randomly generated. The following steps are repeated until a termination criterion is reached: two parents are selected (out of four individuals) using binary tournament and are recombined in order to obtain two offspring. The offspring are considered for mutation. The best offspring replaces the worst individual in the current population if the offspring is better than the latter.

5.2 Gene Expression Programming

GEP uses a generational algorithm. The initial population is randomly generated. The following steps are repeated until a termination criterion is reached: A fixed number of the best individuals enter the next generation (elitism). The mating pool is filled by using binary tournament selection. The individuals from the mating pool are randomly paired and recombined. Two offspring are obtained by recombining two parents. The offspring are mutated and they enter the next generation.

5.3 Grammatical Evolution

GE uses a steady-state [18] algorithm (similar to the MEP algorithm).

5.4 Linear Genetic Programming

LGP uses a modified steady-state algorithm. The initial population is randomly generated. The following steps are repeated until a termination criterion is reached: 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 and the offspring are mutated and then replace the losers of the tournament.

6. Fitness assignment

The fitness assignment strategies employed by MEP, GEP, GE, and LGP are described in this section.

6.1 Multi-Expression Programming

MEP uses a special kind of fitness assignment. The value of each expression encoded in a chromosome is computed during the individual evaluation (in concordance to the description given in section 2, a MEP individual encodes a number of expressions equal to the number of its genes). This evaluation is performed by reading the chromosome only once and storing partial results by using dynamic programming [19]. The best expression is chosen to represent the chromosome (i.e., to assign the fitness of the chromosome).

Thus, the fitness of a MEP chromosome may be computed by using the formula:

$$f = \min_{k=1,L} \left\{ \sum_{j=1}^{N} |E_j - O_j^k| \right\},$$

where N is the number of fitness cases, O_j^k is the value returned (for the jth fitness case) by the *kth* expression encoded in the current chromosome, L is the number of chromosome genes, and E_j is the expected value for the fitness case j.

6.2 Gene Expression Programming

In [7] the fitness of a GEP chromosome was expressed by the equation:

$$f = \sum_{j=1}^{N} (M - |O_j - E_j|),$$

where M is the selection range (see [7] for more information), O_j is the value returned by a chromosome for the fitness case j, and E_j is the expected value for the fitness case j.

6.3 Grammatical Evolution

The fitness of a GE individual may be computed using the equation:

$$f = \sum_{j=1}^{N} (|O_j - E_j|),$$

with the same parameters as above.

6.4 Linear Genetic Programming

The fitness of a LGP individual may be computed by using the equation:

$$f = \sum_{j=1}^{N} (|O_j - E_j|),$$

with the same parameters as above.

6.5 The complexity of the fitness assignment process

The complexity of the MEP, GEP, GE, and LGP fitness assignment process for solving symbolic regression problems is O(NG), where NG is the number of genes in a chromosome.

When solving symbolic regression problems, the MEP algorigthm does not have a higher complexity than the other GP techniques that encode a single expression in each chromosome [3]. This is due to its special mechanism used for expression evaluation.

7. Exceptions handling

Exceptions are conditions that require special handling. They can include errors such as division by zero, overflow and underflow (that arise when variable storage capacity is exceeded), invalid arguments, and so on. Exception handling is a mechanism that recognizes and fixes errors.

The exception handling techniques employed by the compared evolutionary algorithms are described in this section.

7.1 Multi-Expression Programming

When a gene generates an exception, that gene is automatically changed (mutated) into a randomly chosen terminal symbol. In this way no infertile individual may enter the next generation. This exception handling mechanism allows changing the chromosome structure during the evaluation process. In this way the value of the currently evaluated gene is the only one which needs to be recomputed.

When the training examples are divided into three subsets (training, validation, and test sets [13]) the described exception handling mechanism is used only for the training stage. When applied for the validation and test sets the expression may not be changed. If an exception occurs during the evaluation of these subsets it is recommended as the corresponding gene to return a predefined value (e.g., 1.0).

7.2 Gene Expression Programming

If an instruction contained in a chromosome generates an exception, that chromosome will be considered invalid and it will receive the lowest fitness possible.

7.3 Grammatical Evolution

GE uses a *protected exception* handling mechanism [1]; that is, if a subexpression generates an exception, the result of that subexpression will be a predefined (symbolic) value (e.g., for division by zero the predefined result may be 1.0).

7.4 Linear Genetic Programming

LGP uses a protected exception handling mechanism like that used by GE.

8. Numerical experiments

Several numerical experiments with MEP, GEP, GE, and LGP are carried out in this section. Five test problems are chosen for these experiments. Three of them are artificially constructed. The other two problems contain real-world data and have been taken from PROBEN1 [13] (which have been adapted from UCI Machine Learning Repository [20]).

Each problem taken from PROBEN1 has three versions. The first one reflects the task formulation as it was provided by the collectors, and the other two are random permutations of the examples, simplifying the problem to one of interpolation.

8.1 Test problems

In this section, five test problems used in the numerical experiments are described.

T_1. Find a function that best satisfies a set of fitness cases generated by the quartic polynomial [1] function

$$f_1(x) = x^4 + x^3 + x^2 + x.$$

T_2. Find a function that best satisfies a set of fitness cases generated by the function

$$f_2(x) = \sin(x^4 + x^2).$$

T_3. Find a function that best satisfies a set of fitness cases generated by the function

$$f_3(x) = \sin(\exp(\sin(\exp(\sin(x))))).$$

The set of fitness cases for these problems was generated using (for the variable x) 20 randomly chosen values in the interval [0, 10].

T_4. *Building*. The purpose of this problem is to predict the electric power consumption in a building. The problem was originally designed for predicting the hourly consumption of hot and cold water and electrical energy based on the date, time of day, outdoor temperature, outdoor air, humidity, solar radiation, and wind speed. In this paper, the original problem was split into three subproblems (predicting the consumption of electrical power, hot water, and cold water) because not all GP techniques are designed for handling multiple outputs of a problem. The prediction of cold water consumption is the only one considered here. The other two problems (the prediction of hot water and power consumption) may be handled in a similar manner.

The "Building" problem has 14 inputs restricted to the interval [0, 1]. The data set contains 4208 examples [13].

In PROBEN1 three different variants of each dataset are given concerning the order of the examples. This increases the confidence that results do not depend on a certain distribution of the data into training, validation, and test sets. The original problem is called *Building1* and the other two versions are called *Building2* and *Building3*.

T_5. *Heartac*. The purpose of this problem is to predict heart disease. More specifically we have to compute how many vessels (out of four) are reduced in diameter by more than 50%. The decision is made based on personal data such as age, sex, smoking or nonsmoking habits, the subjective pain descriptions provided by the patient, and the results of various medical examinations such as blood pressure and electrocardiogram results.

The dataset has 35 inputs and contains 303 examples. The data were provided by the V. A. Medical Center, Long Beach and Cleveland Clinic Foundation: Robert Detrano, M.D., Ph.D.

The original problem is called *Heartac1* and the other two versions are called *Heartac2* and *Heartac3*.

8.2 General parameter settings

In all the experiments performed in this paper MEP, GEP, and LGP use the same set of function symbols. The set of function symbols is:

$$F = \{+, -, *, /, \sin, \exp\}.$$

For the first three test problems the set of terminal symbols is:

$$T = \{x\}.$$

and for the real-world problems the terminal set consists of the problem inputs.

The grammar used by GE is:

$$G_{\mathrm{GE}} = \{N, T, S, P\},$$

where the terminal set is:

$$T = \{+, -, *, /, \sin, \exp, (,)\},$$

and the nonterminal symbols are:

$$N = \{\mathrm{expr}, \mathrm{op}, \mathrm{pre\_op}\}.$$

The start symbol is:

$$S = \texttt{<expr>}.$$

The production rules P are:

```
<expr> ::= <expr><op><expr>|
           (<expr><op><expr>)|
           <pre_op>(<expr>)|
           <var>.
<op> ::= +| − | * |/.
<pre_op> ::= sin|exp.
```

Binary tournament selection is used in all of the experiments by the compared techniques.

One-point crossover is used by GEP, MEP, and GE with a probability of $p_{\mathrm{cross}} = 0.9$. A LGP-specific crossover operator is also applied with a probability of $p_{\mathrm{cross}} = 0.9$. The mutation probability was set to two mutations per chromosome. Four supplementary registers were used in all of the LGP experiments. 100 runs were carried out for all the test problems.

Since MEP, GEP, GE, and LGP use different chromosome representations we cannot make a direct comparison based on chromosome length. Instead we provide a comparison based on the number of symbols in a chromosome.

The first position of a MEP chromosome is always a terminal symbol. Thus, the maximum number of symbols in a MEP chromosome is given by the formula:

$$\text{Number of Symbols} = 3 * \text{Number of Genes} - 2.$$

A GEP chromosome consists of a single gene with head length h. Thus, the overall number of symbols in a GEP chromosome is $(2h + 1)$.

The GE chromosomes are binary strings. The groups made up of eight consecutive bits are called *codons*. The number of symbols in

Parameter	Value
Population size	100
Number of symbols in chromosome	58 (20 MEP genes, 29 GEP chromosome head length, 58 GE codons, 15 LGP instructions).

Table 1. Parameters for Experiment 1.

the GE chromosomes is considered to be the number of codons. The maximum number of symbols in a LGP chromosome is four times the number of instructions.

Since the GE and LGP individuals have variable length we shall start with shorter individuals whose length will increase as the search process advances. The maximum initial number of the GE and LGP symbols is set to be about 2/3 of the maximum permitted number of symbols in the GE and LGP chromosomes. The length of the LGP and GE chromosomes may increase as an effect of the crossover operator [5, 8]. The GEP selection range (see [7] and section 6 this paper) is 100%. Ten wrappings are allowed for the GE chromosomes.

8.3 Experiment 1

In this experiment, the relationship between the success rate of MEP, GEP, GE, and LGP algorithms and the number of generations is analyzed.

The success rate is computed as:

$$\text{Success rate} = \frac{\text{Number of successful runs}}{\text{Total number of runs}}.$$

The algorithm parameters for MEP, GEP, GE, and LGP are given in Table 1. The success rate of the MEP, GEP, GE, and LGP algorithms depending on the number of generations is depicted in Figure 1.

As can be seen in Figure 1, MEP and LGP have the best overall behavior. For the test problems T_1 and T_2 MEP has the best success. LGP has the best behavior for the third test problem and is followed by MEP. GE and GEP have equally good behaviors for all considered test problems.

8.4 Experiment 2

In this experiment the relationship between the success rate and the population size is analyzed. The algorithm parameters for this experiment are given in Table 2. Experiment results are depicted in Figure 2.

MEP and LGP perform better than GE and GEP for all test problems and for all considered population sizes. For test problem T_1 the MEP success rate is 100% (for all populations larger than 60 individuals). MEP has the best success rate for the second test problem. GEP and

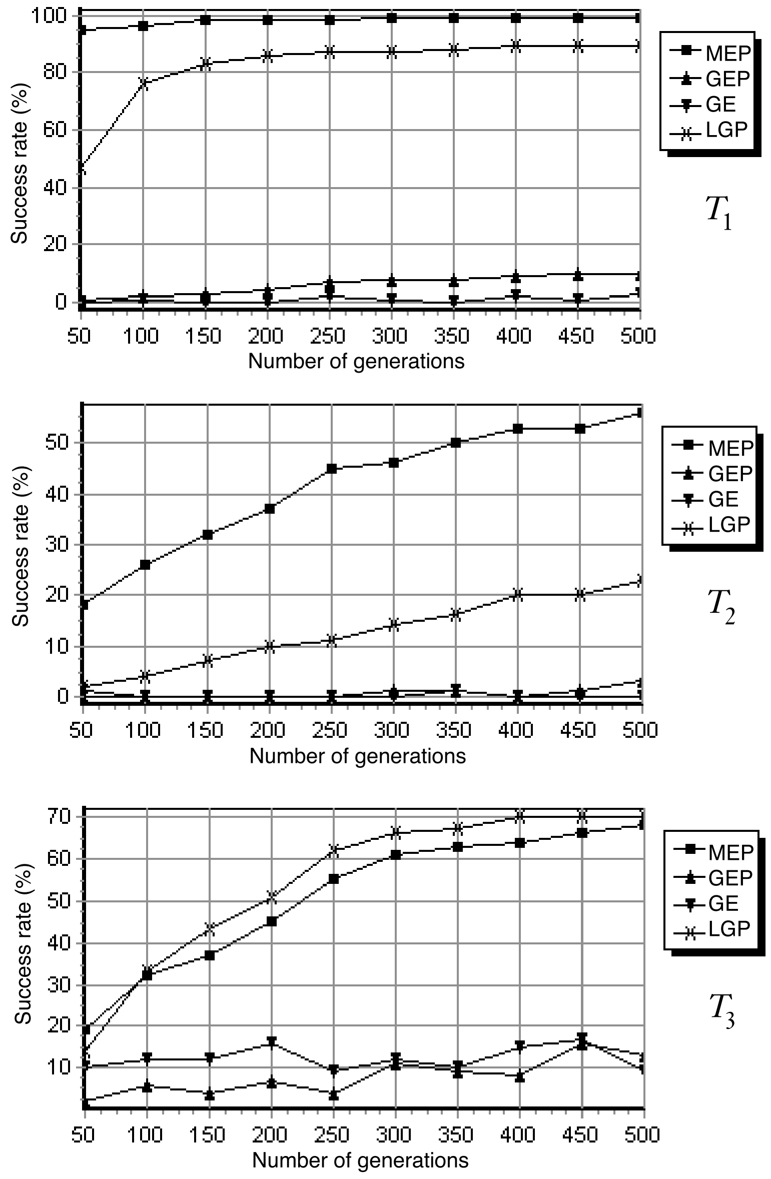
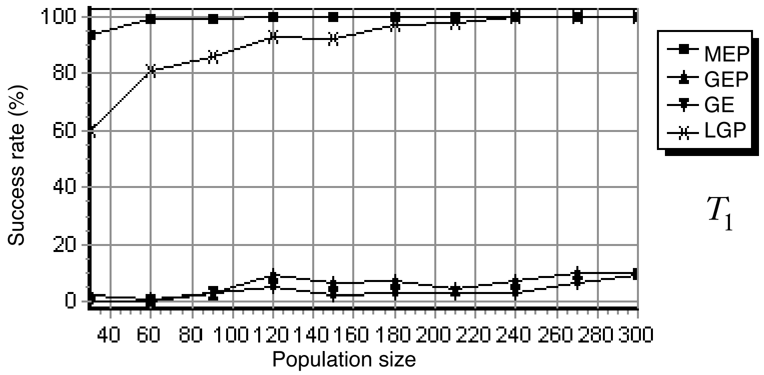
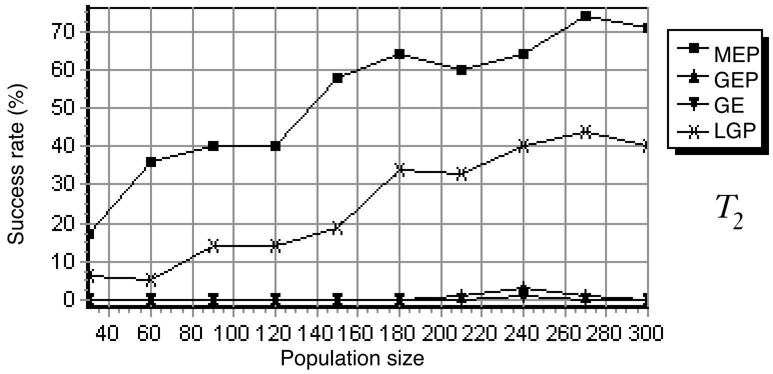


Figure 1. The relationship between the success rate and the number of generations for the test problems T_1, T_2, and T_3. The number of generations varies between 50 and 500. Results are summed over 100 runs.





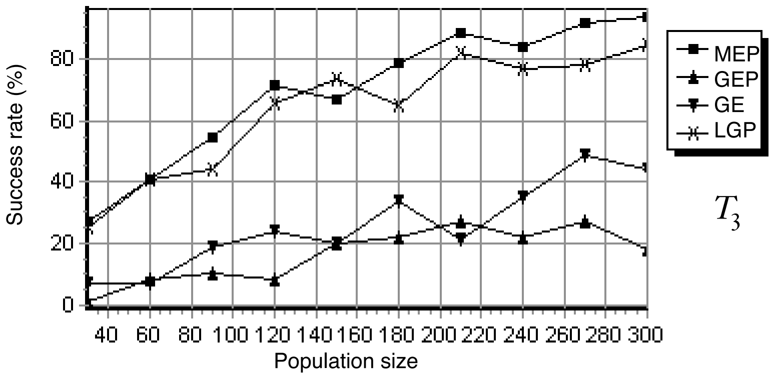


Figure 2. The relationship between the success rate and the population size for the test problems T_1, T_2, and T_3. The population size varies between 30 and 300. Results are summed over 100 runs.

Parameter	Value
Number of generations	100
Number of symbols in chromosome	58 symbols (20 MEP genes, 29 GEP chromosome head length, 58 GE codons, 15 LGP instructions)

Table 2. Parameters for Experiment 2.

GE are not able to solve this problem in a satisfactory manner, having a success rate of less than 10%. For test problem T_3 LGP and MEP have similar success rates followed by GE.

8.5 Experiment 3

The test problems considered in this experiment are the problems *Building* and *Heartac* taken from PROBEN1 [13]. Each dataset is divided into three subsets (training set, 50%; validation set, 25%; and test set, 25%) [13].

A method called *early stopping* is used to avoid overfitting of the population individuals to the particular training examples used [13]. This method consists of computing the test set performance for that chromosome having the smallest validation error during the search process. The generalization performance will be increased by using early stopping.

The error function reported in this experiment is:

$$E = 100\frac{1}{N}\sum_{i=1}^{N}\left|e_i - o_i\right|,$$

where e_i is the expected output value (the value that must be predicted), o_i is the obtained value (the value obtained by the best individual) for the *ith* example, and N is the number of examples.

The standard deviation of the obtained values (over 100 runs) is also computed in conjunction with the error. Minimal and maximal error values of the best individual over all runs are also reported.

Each algorithm uses a population of 100 individuals with 100 symbols (100 GE codons, 34 MEP genes, 49 GEP chromosome head length, and 25 LGP instructions). 100 runs of 100 generations were carried out for each algorithm and for each test problem.

The results obtained by MEP, GEP, GE, and LGP are presented in Tables 3 (for the training set), 4 (for the validation set), and 5 (for the test set).

MEP and LGP have the best mean values for the training, validation, and test sets. However, MEP has the best mean error for the test set thus providing the best generalization ability. The lowest maximal and

Algorithm	Training set			
	Mean	StdDev	Min	Max
Building 1 - cold water consumption				
MEP	***3.81***	0.70	3.19	***5.62***
GEP	12.59	3.39	6.98	17.98
GE	9.00	3.88	3.57	17.98
LGP	4.53	1.86	***2.79***	15.24
Building 2 - cold water consumption				
MEP	***4.00***	0.96	***3.06***	***7.56***
GEP	13.34	3.14	6.01	16.32
GE	9.26	3.41	3.65	16.32
LGP	4.77	1.70	3.54	***7.56***
Building 3 - cold water consumption				
MEP	***4.18***	1.28	***3.11***	***7.74***
GEP	13.06	3.28	5.91	16.20
GE	8.31	3.02	3.61	16.20
LGP	4.77	1.92	3.57	12.97
Heartac 1				
MEP	***19.31***	1.91	***15.47***	***23.14***
GEP	25.12	0.68	22.15	25.30
GE	23.87	2.60	17.51	26.83
LGP	21.50	3.05	16.02	28.22
Heartac 2				
MEP	***20.01***	2.30	15.66	***23.57***
GEP	23.94	0.19	23.18	24.03
GE	23.21	2.05	18.10	26.30
LGP	22.32	2.83	***15.28***	26.30
Heartac 3				
MEP	***18.97***	2.42	14.15	***23.14***
GEP	23.97	0.83	20.90	24.28
GE	22.56	2.74	15.80	28.01
LGP	20.15	3.89	***12.96***	28.01

Table 3. Results obtained by GEP, MEP, GE, and LGP for the training set. Mean and StdDev are the mean and the standard deviation of the error for the best individual error over 100 runs. Min and Max are the minimal and maximal error values for the best individual over 100 runs. The best values for each problem have been highlighted.

Algorithm	Validation set			
	Mean	StdDev	Min	Max
Building 1 - cold water consumption				
MEP	5.12	2.53	***2.72***	***12.16***
GEP	10.51	5.56	5.40	22.29
GE	8.19	4.01	3.92	22.29
LGP	***4.99***	2.20	3.47	18.42
Building 2 - cold water consumption				
MEP	***4.01***	0.93	***3.07***	***7.38***
GEP	13.49	3.23	5.80	16.62
GE	9.28	3.55	3.70	16.62
LGP	4.76	1.62	3.45	***7.38***
Building 3 - cold water consumption				
MEP	***4.22***	1.40	***3.03***	***7.92***
GEP	13.12	3.27	5.99	16.28
GE	8.46	2.99	3.59	16.28
LGP	4.86	2.02	3.59	12.90
Heartac 1				
MEP	***21.97***	2.59	17.97	28.53
GEP	26.66	0.07	26.22	26.67
GE	27.25	3.51	19.60	31.89
LGP	23.68	3.10	***16.97***	***28.4***
Heartac 2				
MEP	***19.62***	2.19	15.39	***24.61***
GEP	26.48	1.29	21.30	27.06
GE	23.98	3.91	16.83	32.61
LGP	23.95	4.44	***15.24***	32.61
Heartac 3				
MEP	***21.38***	1.98	17.66	***24.88***
GEP	26.32	1.33	22.8	26.86
GE	23.66	2.88	16.25	29.81
LGP	22.91	3.14	***16.63***	29.81

Table 4. The results obtained by GEP, MEP, GE, and LGP for the validation set. Mean and StdDev are the mean and the standard deviation of the error for the best individual error over 100 runs. Min and Max are the minimal and maximal error values for the best individual over 100 runs. The best values for each problem have been highlighted.

Algorithm	Test set			
	Mean	StdDev	Min	Max
Building 1 - cold water consumption				
MEP	***4.20***	1.17	***2.96***	***10.15***
GEP	10.81	5.46	4.00	22.07
GE	7.98	3.79	3.08	22.07
LGP	5.11	2.36	3.09	16.55
Building 2 - cold water consumption				
MEP	***3.96***	0.96	***3.25***	***7.47***
GEP	12.97	3.04	5.97	15.88
GE	9.08	3.30	3.60	15.88
LGP	4.73	1.71	3.53	***7.47***
Building 3 - cold water consumption				
MEP	***4.42***	1.39	***3.08***	***8.03***
GEP	13.25	3.29	5.98	16.46
GE	8.62	3.01	3.80	16.46
LGP	5.08	2.03	3.77	12.83
Heartac 1				
MEP	***16.69***	2.63	***10.72***	22.89
GEP	22.12	0.92	17.88	***22.36***
GE	22.35	4.74	13.67	28.03
LGP	19.50	3.83	13.19	29.36
Heartac 2				
MEP	***21.00***	2.47	14.04	***24.71***
GEP	26.60	1.23	23.49	27.36
GE	23.93	2.91	17.89	29.78
LGP	23.69	3.57	***13.99***	29.78
Heartac 3				
MEP	***20.80***	2.53	***17.22***	26.46
GEP	23.97	0.22	23.76	***25.26***
GE	24.78	2.79	17.52	29.86
LGP	22.65	3.03	17.23	29.86

Table 5. The results obtained by GEP, MEP, GE, and LGP for the test set. Mean and StdDev are the mean and the standard deviation of the error for the best individual error over 100 runs. Min and Max are the minimal and maximal error values for the best individual over 100 runs. The best values for each problem have been highlighted.

Problem	I	II	III	IV
Building 1	MEP	LGP(21.66%)	GE(90.00%)	GEP(157.38%)
Building 2	MEP	LGP(19.44%)	GE(129.29%)	GEP(227.52%)
Building 3	MEP	LGP(14.93%)	GE(95.02%)	GEP(199.77%)
Heartac 1	MEP	LGP(16.83%)	GEP(32.53%)	GE(33.91%)
Heartac 2	MEP	LGP(12.80%)	GE(13.95%)	GEP(26.66%)
Heartac 3	MEP	LGP(8.89%)	GEP(15.24%)	GE(19.13%)

Table 6. A hierarchy of the compared algorithms for the considered test problems.

Problem	**MEP - LGP P-values**
Building 1	7E-4
Building 2	1E-4
Building 3	7E-3
Heartac 1	8E-9
Heartac 2	3E-9
Heartac 3	5E-6

Table 7. The P-values (in scientific notation) of the t-test with 99 degrees of freedom.

minimal error values are obtained by MEP and LGP with very few exceptions.

In addition to the error values we are also interested in establishing a hierarchy of the compared algorithms taking into account the mean values in the test set. Consequently, we compute the difference (in percent) between the average errors of the best algorithm and the other three algorithms. The results are presented in Table 6.

In all six cases MEP was ranked as the best algorithm, closely followed by LGP. GE was ranked as the third algorithm in four cases and GEP was ranked as the third one in two cases.

To determine if the differences between MEP and LGP are statistically significant we use a t-test with 95% confidence. Before applying the t-test, an f-test was used to determine if the compared data have the same variance. The P-values of a two-tailed t-test are given in Table 7. It can be seen that the difference between MEP and LGP is statistically significant for all of the test problems.

8.6 Discussion of the obtained results

Numerical experiments revealed the following interesting aspects of the considered algorithms.

1. MEP has the best overall behavior on the considered test problems. However, MEP cannot be the best for all test problems. According to the "no free lunch theorems for search and optimization" [15, 21], several situations where the other algorithms are better than MEP have been identified.

2. The systems that use chromosomes of variable length (such as LGP, GE, and MEP) seem to perform better than the systems that use fixed-length chromosomes (such as GEP). MEP uses fixed-length chromosomes, but its ability to store multiple solutions in a single chromosome works as a provider of variable-length chromosomes.

3. Test problem T_2 is the most difficult. For this problem GE and GEP yield many 0% success rates. We made another numerical experiment with only GE and GEP. We removed the sin function from the problem definition and also removed the operators sin and exp from the GEP function set and from the GE productions. In this case the GEP and GE performance improved substantially, yielding many success rates over 90%. It seems that adding the sin and exp operators greatly reshaped the search space. The same very good results have been obtained by GE and GEP (using the modified set of functions) for the test problem T_1.

4. For the real-world problems (*Building* and *Heartac*), MEP has the best results, in terms of mean error, seconded by LGP.

In order to provide a fair comparison several adjustments have been made to each algorithm, which could lead to some poor behavior. However, we have to keep in mind that each minor factor (such as the mutation and crossover probability) could affect the quality of the comparison presented in this paper.

There are some modifications that can be made in order to improve the quality of the considered algorithms. For instance, GE performance could be improved if we allow chromosomes of any length to appear in the system. However, this feature could lead to bloat [16]. Another improvement to GE can be made by increasing the number of mutations per chromosome. We increased this number to 20 mutations/chromosome and obtained, for test problem T_1 (using the parameters given in section 8.5), a success rate of 21% using a population of 30 individuals and a success rate of 35% using a population of 60 individuals.

GEP performance could be improved by using the multigenic system. However, as indicated in section 2.2.2, this system has some weaknesses and requires multiple trials to find the optimal number of genes in a chromosome. Another improvement to GEP can be made by setting the number of symbols in a chromosome to an optimal value. We reduced this number to eight symbols in the GEP chromosome head and obtained, for test problem T_1 (using the parameters given in section 8.5), a success rate of 42% using a population of 30 individuals and a success rate of 49% using a population of 60 individuals. We could not use this number of symbols in a chromosome (17) because the corresponding LGP chromosome would be too short to encode a solution for the considered test problems.

The performance of the considered algorithms could be improved by choosing other parameters for the numerical experiments. This could include the fine tuning of all algorithm parameters for a given problem. An example is GEP, which has been proved to be quite sensitive to chromosome length [7]. Other changes could affect the function set (by extending it with other functions or by reducing it to an optimal subset of function symbols). As mentioned before, GEP and GE perform very well for the test problem T_1 if we remove the unary operators (sin and exp) from the function set, respectively from the GE production rules.

Implementation of the algorithm-specific features to other algorithms would also be of interest. For instance, the multi-expression paradigm could be implemented and tested within other evolutionary techniques such GEP, GE, and LGP.

Another reason that MEP and LGP perform better than GE and GEP could be due to their code-reuse ability. Standard Genetic Programming employs this feature by using the Automatically Defined Functions (ADFs) mechanism [2].

9. Conclusions and future work

A systematic comparison of four evolutionary techniques for solving symbolic regression problems has been presented in this paper. All aspects of the compared techniques have been analyzed. Several numerical experiments have been performed using five test problems.

Further efforts will focus on comparing the considered techniques on the basis of more optimization problems.

Acknowledgments

The authors are grateful to anonymous referees for their constructive comments and criticism of earlier versions of this paper.

References

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

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

[3] M. Oltean and D. Dumitrescu, “Multi Expression Programming,” Technical Report UBB-01 (available from www.mep.cs.ubbcluj.ro, 2002).

[4] M. Oltean and C. Groşan, "Evolving Evolutionary Algorithms by using Multi Expression Programming," in *The Seventh European Conference on Artificial Life*, edited by W. Banzhaf (*et. al*), LNAI 2801 (Springer-Verlag, Berlin, 2003).

[5] C. Ryan, J. J. Collins, and M. O'Neill, "Grammatical Evolution: Evolving Programs for an Arbitrary Language," in *Proceedings of the First European Workshop on Genetic Programming*, edited by W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty (Springer-Verlag, Berlin, 1998).

[6] C. Ryan and M. O'Neill, "Grammatical Evolution: A Steady State Approach," in *Late Breaking Papers, Genetic Programming*, edited by J. R. Koza (University of Wisconsin, Madison, Wisconsin, 1998).

[7] C. Ferreira, "Gene Expression Programming: A New Adaptive Algorithm for Solving Problems," *Complex Systems*, **13** (2001) 87–129.

[8] M. Brameier and W. Banzhaf, "A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining," *IEEE Transactions on Evolutionary Computation*, **5** (2001) 17–26.

[9] M. Brameier and W. Banzhaf, "Explicit Control of Diversity and Effective Variation Distance in Linear Genetic Programming," in *Proceedings of the Fourth European Conference on Genetic Programming*, edited by E. Lutton, J. Foster, J. Miller, C. Ryan, and A. Tettamanzi (Springer-Verlag, Berlin, 2002).

[10] J. Miller and P. Thomson, "Cartesian Genetic Programming," in *Proceedings of the Third European Conference on Genetic Programming*, edited by Riccardo Poli, Wolfgang Banzhaf, Bill Langdon, Julian Miller, Peter Nordin, and Terence C. Fogarty (Springer-Verlag, Berlin, 2002).

[11] N. Patterson, *Genetic Programming with Context-Sensitive Grammars*, Ph.D. Thesis, School of Computer Science, University of Scotland (2002).

[12] W. Banzhaf, "Genotype-Phenotype-Mapping and Neutral Variation: A Case Study in Genetic Programming," in *Proceedings of the Third International Conference on Parallel Problem Solving from Nature*, edited by Y. Davidor, H.-P. Schwefel, and R. Männer (Springer-Verlag, Berlin, 1994).

[13] L. Prechelt, "PROBEN1: A Set of Neural Network Problems and Benchmarking Rules," Technical Report 21, (1994), University of Karlsruhe, (available from ftp://ftp.cs.cmu.edu/afs/cs/project /connect/bench/contrib/prechelt/proben1.tar.gz).

[14] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools* (Addison-Wesley, Reading, MA, 1986).

[15] D. H. Wolpert and W. G. Macready, "No Free Lunch Theorems for Search," *Technical Report SFI-TR-05-010*, Santa Fe Institute, 1995 (available from ftp://ftp.santafe.edu/pub/dhwftp/nfl.search.TR.ps.Z).

[16] W. Banzhaf and W. B. Langdon, "Some Considerations on the Reason for Bloat," *Genetic Programming and Evolvable Machines*, **3** (2002) 81–91.

[17] S. Luke and L. Panait, "Fighting Bloat with Nonparametric Parsimony Pressure," in *Proceedings of the Seventh International Conference on Parallel Problem Solving from Nature*, edited by J. J. Merelo Guervos, P. Adamidis, H-G. Beyer, J. L. Fernandez-Villacanas Martin, and H-P. Schwefel (Springer-Verlag, Berlin, 2002).

[18] G. Syswerda, "Uniform Crossover in Genetic Algorithms," in *Proceedings of the Third International Conference on Genetic Algorithms*, edited by J. D. Schaffer (Morgan Kaufmann Publishers, San Mateo, CA, 1989).

[19] R. Bellman, *Dynamic Programming* (Princeton University Press, New Jersey, 1957).

[20] UCI Machine Learning Repository (available from www.ics.uci.edu/~mlearn/MLRepository.html)

[21] D. H. Wolpert and W. G. Macready, "No Free Lunch Theorems for Optimization," *IEEE Transaction on Evolutionary Computation*, **1** (1997) 67–82.