Multi Expression Programming

Mihai Oltean, D. Dumitrescu

Department of Computer Science,
Faculty of Mathematics and Computer Science,
Babeş-Bolyai University, Kogălniceanu 1,
Cluj-Napoca, 3400, Romania.
email: {moltean, ddumitr}@nessie.cs.ubbcluj.ro

Abstract. Multi Expression Programming (MEP) is a new evolutionary paradigm intended for solving computationally difficult problems. MEP individuals are linear entities that encode complex computer programs. MEP chromosomes are represented in the same way as C or Pascal compilers translate mathematical expressions into machine code. MEP is used for solving certain difficult problems such as symbolic regression, classification, multiplexer and game strategy discovering. MEP is compared to Genetic Programming (GP), Gene Expression Programming (GEP), Linear Genetic Programming (LGP) and Cartesian Genetic Programming (CGP) by using several well-known test problems. Compared with these techniques, MEP performs better in most of the cases, as numerical experiments show.

Keywords: Multi Expression Programming, Genetic Programming, Gene Expression Programming, Linear Genetic Programming, Symbolic Regression, Even-Parity, Multiplexer, Discovering Game Strategy.

1. Introduction

Genetic Programming (GP) [13, 14, 15] is an evolutionary technique used for breeding a population of computer programs. Several linear variants of GP have been proposed. Some of them are: Cartesian Genetic Programming (CGP) [17], Grammatical Evolution (GE) [16], Linear GP [3], Gene Expression Programming (GEP) [6] and Genetic Algorithms for Deriving Software (GADS) [19].

All linear GP variants have a common feature: individuals are represented as linear entities (strings) that are decoded and expressed as non-linear entities (trees).

A new evolutionary technique called *Multi Expression Programming*[1] (MEP) is proposed. MEP uses a simple and compact representation. This representation allows that variation operators generate only feasible individuals. A MEP individual encodes many different expressions, thereby offering some implicit building block structures and parallelism. This mechanism is called strong implicit parallelism.

Some of the MEP features are summarized below:

a) MEP individuals are strings of genes encoding complex computer programs,

b) When MEP individuals encode expressions, their representation is similar to the way in which compilers translate *C* or *Pascal* expressions into machine code [1]. This may lead to very efficient implementation into assembler languages. The ability of evolving machine code (leading to very important speedups) has been considered by others researchers, too. For instance Nordin [18] evolves programs represented in machine code. Poli and Langdon proposed sub-machine-code GP [20], which exploits the processor's ability to perform some operations simultaneously. Compared to these approaches, MEP has the advantage that it uses a representation that is more compact, simpler, and independent of any programming language.

c) A salient MEP feature is the ability of storing multiple solutions of a problem in a single chromosome. Usually, the best solution is chosen for fitness assignment. When solving symbolic

[1] MEP source code is available at www.mep.cs.ubbcluj.ro.

regression or classification problems (or any other problems for which the training set is known before the problem is solved) MEP has the same complexity as other techniques storing a single solution in a chromosome (such as GP, CGP, GEP or GE).

d) Evaluation of the expressions encoded into a MEP individual can be performed by a single parsing of the chromosome.

e) As we already have noticed, 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.

Section 2 contains a review of GP techniques with a special emphasize on linear representation approaches. MEP technique is described in section 3.

MEP technique is used for solving various problems such as symbolic regression, multiplexer, even-parity and discovering game strategy. Numerical experiments have proved that MEP performs very well for each problem that has been considered.

MEP technique is compared to GP, GEP and CGP on several well-known benchmark problems. Symbolic regression is addressed in section 4.1. MEP results are compared to those supplied by GEP, CGP and by standard GP. Even parity problem is considered in section 4.2 and the multiplexer problem in section 4.3. In section 4.4, MEP is used for discovering heuristics strategies for solving complex problems. TTT game is considered as an illustration. The results show that MEP algorithm outperforms the mentioned above techniques on most of the examples.

2. Related work

In this section some GP techniques are briefly reviewed.

2.1. Genetic Programming

Whereas the evolutionary schemes employed by GP are similar to those used by other techniques (such as GA, EP, ES), the individual representation and the corresponding genetic operators are specific only to GP. Due to its nonlinear individual representation (GP individuals are usually represented as trees) GP is widely known as a technique that creates computer programs [13, 14, 15].

Each of the GP individuals is usually represented as a tree encoding a complex computer program. The genetic operators used with GP are crossover and mutation. The crossover operator takes two parents and generates two offspring by exchanging sub-trees between the parents. The mutation operator generates an offspring by changing a sub-tree of a parent into another sub-tree.

For efficiency reasons, each GP program tree is stored as a vector using the Polish form (see [15] chapter 63). A mathematical expression in Infix and Polish notations and the corresponding GP program tree are depicted in Figure 1.

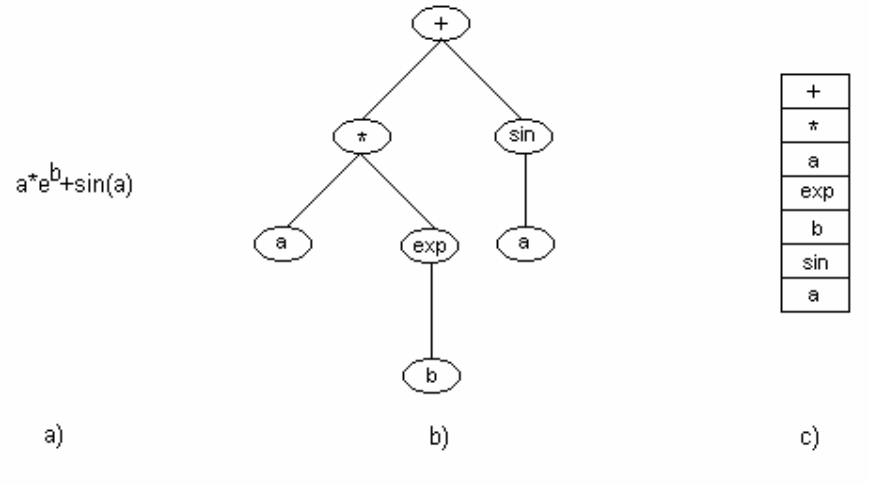

Figure 1. A mathematical expression in infix form (a), Polish form (c) and the corresponding program tree (b).

Each element in this vector contains a function or a terminal symbol. Since each function has a unique arity we can clearly interpret each vector that stores an expression in Polish notation. In this notation, a sub-tree of a program tree corresponds to a particular contiguous subsequence of the vector. When applying the crossover or the mutation operator, the exchanged or changed subsequences can easily be identified and manipulated.

2.2. Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) [17] is a GP technique that encodes chromosomes in graph structures rather than standard GP trees. 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 [15].

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 GP mechanism. A CGP node contains a function symbol and pointers towards nodes representing function parameters. Each CGP node has an output that may be used as input for another node.

An example of CGP program is depicted in Figure 2.

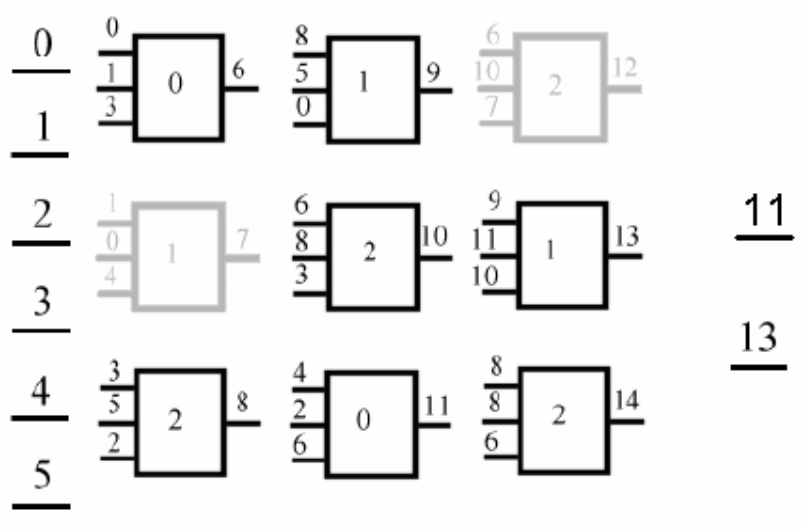

Figure 2. A CGP program with 5 inputs, 2 outputs and 3 functions (0, 1, 2 inside square nodes). The grey squares represent unconnected nodes.

Each CGP program (graph) is defined by several parameters: number of rows (n_r), number of columns (n_c), number of inputs, number of outputs, and number of functions. The nodes interconnectivity is defined as being the number (l) of 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).

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 CGP chromosome depicted in Figure 1 is encoded as:

C = **0 1 3 0** 1 0 4 1 **3 5 2 2 8 5 0 1 6 8 3 2 4 2 6 0** 6 10 7 2 **9 11 10 1 8 8 6 2 11 13**

Standard string genetic operators (crossover and mutation) are used within CGP system. Crossover may be applied without any restrictions. Mutation operator requires that some conditions are met. Nodes supplying the outputs are not fixed as they are also subject to crossover and mutation.

2.3. Gene Expression Programming

Gene Expression Programming (GEP) [6] uses linear chromosomes. A chromosome is composed of genes containing terminal and function symbols. Chromosomes are modified by mutation, transposition, root transposition, gene transposition, gene recombination, one-point and two-point recombination.

GEP genes are composed of a *head* and a *tail*. The head contains both functions and terminals symbols. The tail contains only terminal symbols.

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

$$t = (n - 1)*h + 1, \quad (1)$$

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

A tree–program is translated into a GEP gene is by means of breadth-first parsing.

Let us consider a gene made up of symbols in the set S:

$S = \{*, /, +, –, a, b\}$.

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

$C = +*ab/+aab+ababbbababb$.

The expression encoded by the gene C is

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

The expression E represents the phenotypic transcription of a chromosome having C as its unique gene.

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 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 situation [6]. But, generally, it is not a good idea to assume that the genes may be linked either by addition or by multiplication. If the functions {+, -, *, /} are used as linking operators then, the complexity of the problem grows substantially (since the problem of determining how to mixed these operators with the genes is as hard as the initial problem).

When solving computationally difficult problems (like automated code generation) one should not assume that a unique kind of function symbol (like **for, while** or **if** instructions) is necessary for inter-connecting different program parts.

Moreover, the success rate of GEP increases with the number of genes in the chromosome [5]. However, after a certain value, the success rate decreases if the number of genes in the chromosome increases. This is because one can not force a complex chromosome to encode a less complex expression.

Thus, when using GEP one must be careful with the number of genes that form the chromosome. The number of genes in the chromosome must be somehow related to the complexity of the expression that one wants to discover.

According to [6], GEP performs better than standard GP for several particular problems.

2.4. Linear genetic programming

Linear Genetic Programming (LGP) [3] 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, e.g. $r_i = r_j * c$.

An example of the LGP program is the following one:

```
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] = [7] + 115;
  if (v[1] <= v[6])
    v[1] = sin(v[7]);
}
```

A linear genetic program can be turned into a functional representation by successive replacements of variables starting with the last effective instruction. The variation operators used here are crossover and mutation. By crossover, continuous sequences of instructions are selected and exchanged between parents. Two types of mutations 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.

3. MEP technique

In this section *Multi Expression Programming* (MEP) paradigm is described. MEP uses a new representation technique and a special phenotype transcription model. A MEP chromosome usually encodes several expressions.

3.1 Standard MEP Algorithm

The standard MEP algorithm uses steady state [23] as its underlying mechanism.

The MEP algorithm starts by creating a random population of individuals. The following steps are repeated until a given number of generations is reached: Two parents are selected using a standard selection procedure. The parents are recombined in order to obtain two offspring. The offspring are considered for mutation. The best offspring *O* replaces the worst individual *W* in the current population if *O* is better than *W*.

The variation operators ensure that the chromosome length is a constant of the search process. The algorithm returns as its answer the best expression evolved along a fixed number of generations.

The standard MEP algorithm is outlined below:

STANDARD MEP ALGORITHM

```
S1. Randomly create the initial population P(0)
S2. for t = 1 to Max_Generations do
S3.    for k = 1 to |P(t)| / 2 do
S4.          p1 = Select(P(t));       // select one individual from the current
                                      // population
S5.          p2 = Select(P(t));       // select the second individual
S6.          Crossover (p1, p2, o1, o2); // crossover the parents p1 and p2
                                      // the offspring o1 and o2 are obtained
S7.          Mutation (o1);           // mutate the offspring o1
S8.          Mutation (o2);           // mutate the offspring o2
S9.          if Fitness(o1) < Fitness(o2)
S10.         then if Fitness(o1) < the fitness of the worst individual
                                      in the current population
S11.               then Replace the worst individual with o1;
S12.         else if Fitness(o2) < the fitness of the worst individual
                                      in the current population
S13.               then Replace the worst individual with o2;
S14.   endfor
```

S_{15}. **endfor**

3.2 MEP representation

MEP genes are (represented by) substrings of a 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 proposed representation ensures that no cycle arises while the chromosome is decoded (phenotypically transcripted). According to the proposed 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 positions 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

$F = \{+, *\}$,

and the set of terminals

$T = \{a, b, c, d\}$.

An example of chromosome using the sets F and T is given below:

1: *a*
2: *b*
3: + 1, 2
4: *c*
5: *d*
6: + 4, 5
7: * 3, 6

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

$$Number\_of\_Symbols = (n + 1) * (Number\_of\_Genes - 1) + 1, \quad (2)$$

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 genes encode terminal symbols only.

3.3 Decoding the MEP chromosome and fitness assignment process

Now we are ready to describe how MEP individuals are translated into computer programs. This translation represents the phenotypic transcription of the MEP chromosomes.

Phenotypic translation is obtained by parsing the chromosome 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, 4 and 5 in the previous example encode simple expressions formed by a single terminal symbol. These expressions are:

$E_1 = a$,

$E_2 = b$,

$E_4 = c$,

$E_5 = d$,

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

$E_6 = c + d$.

Gene 7 indicates the operation * on the operands located at position 3 and 6. Therefore gene 7 encodes the expression:

$E_7 = (a + b) * (c + d)$.

E_7 is the expression encoded by the whole chromosome.

There is neither practical nor theoretical evidence that one of these expressions is better than the others. Moreover, Wolpert and McReady [25] 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 chromosome described above encodes the following expressions:

$E_1 = a$,

$E_2 = b$,

$E_3 = a + b$,

$E_4 = c$,

$E_5 = d$,

$E_6 = c + d$,

$E_7 = (a + b) * (c + d)$.

The value of these expressions may be computed by reading the chromosome top down. Partial results are computed by dynamic programming [2] and are stored in a conventional manner.

Due to its multi expression representation, each MEP chromosome may be viewed as a forest of trees rather than as a single tree, which is the case of Genetic Programming.

As MEP chromosome encodes more than one problem solution, it is interesting to see how the fitness is assigned.

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:

$$f(E_i) = \sum_{k=1}^{n} \left| o_{k,i} - w_k \right|, \quad (3)$$

where $o_{k,i}$ is the result obtained by the expression E_i for the fitness case k and w_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:

$$f(C) = \min_i f(E_i). \quad (4)$$

When we have to deal with other problems, we compute the fitness of each sub-expression encoded in the MEP chromosome. Thus, the fitness of the entire individual is supplied by the fitness of the best expression encoded in that chromosome.

3.4 The MEP representation: A discussion

Generally a GP chromosome encodes a single expression (computer program). This is also the case for GEP and GE chromosomes. By contrast, a MEP chromosome encodes several expressions (as it allows a multi-expression representation). Each of the encoded expressions may be chosen to represent the chromosome, i.e. to provide the phenotypic transcription of the chromosome. Usually, the best expression that the chromosome encodes supplies its phenotypic transcription (represents the chromosome).

Therefore, the MEP technique is based on a special kind of implicit parallelism. A chromosome usually encodes several well-defined expressions. The ability of MEP chromosome to encode several syntactically correct expressions in a chromosome is called *strong implicit parallelism* (SIP).

Although, the ability of storing multiple solutions in a single chromosome has been suggested by others authors, too (see for instance [12]), and several attempts have been made for implementing this ability in GP technique. For instance Handley [10] stored the entire population of GP trees in a single graph. In this way a lot of memory is saved. Also, if partial solutions are efficiently stored, we can get a considerable speed up.

Linear GP [3] is also very suitable for storing multiple solutions in a single chromosome. In that case the multi expression ability is given by the possibility of choosing any variable as the program output.

It can be seen that the effective length of the expression may increases exponentially with the length of the chromosome. This is happening because some sub-expressions may be used more than once to build a more complex (or a bigger) expression. Consider, for instance, that we want to obtain a chromosome that encodes the expression a^{2^n}, and only the operators {+, -, *, /} are allowed. If we use a GEP representation the chromosome has to contain at least $(2^{n+1} - 1)$ symbols since we need to store 2^n terminal symbols and $(2^n - 1)$ function operators. A GEP chromosome that encodes the expression $E = a^8$ is given below:

$C =$ *******$aaaaaaaa$.

A MEP chromosome uses only $(3n + 1)$ symbols for encoding the expression a^{2^n}. A MEP chromosome that encodes expression $E = a^8$ is given below:

1: *a*
2: * 1, 1
3: * 2, 2
4: * 3, 3

As a further comparison, when $n = 20$, a GEP chromosome has to have 2097151 symbols, while MEP needs only 61 symbols.

MEP representation is similar to GP and CGP, in the sense that each function symbol provides pointers towards its parameters. Whereas both GP and CGP have complicated representations (trees and graphs), MEP provides an easy and effective way to connect (sub) parts of a computer program. Moreover, the motivation for MEP was to provide an individual representation close to the way in which C or Pascal compilers interpret mathematical expressions [1]. That code is also called three addresses code or intermediary code.

Some GP techniques, like Linear GP, remove non-coding sequences of chromosome during the search process. As already noted ([3], [6]) this strategy does not give the best results. The reason is that sometimes, a part of the useless genetic material has to be kept in the chromosome in order to maintain population diversity.

3.5 Search operators

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

3.6.1 Crossover

By crossover two parents are selected and are recombined.

Three variants of recombination have been considered and tested within our MEP implementation: one-point recombination, two-point recombination and uniform recombination.

3.6.1.1 One-point recombination

One-point recombination operator in MEP representation is similar to the corresponding binary representation operator [5, 9]. One crossover point is randomly chosen and the parent chromosomes exchange the sequences at the right side of the crossover point.

Example

Consider the parents C_1 and C_2 given below. Choosing the crossover point after position 3 two offspring, O_1 and O_2 are obtained as follows:

Parents		Offspring	
C_1	C_2	O_1	O_2
1: ***b***	1: *a*	1: ***b***	1: *a*
2: *** 1, 1**	2: *b*	2: *** 1, 1**	2: *b*
3: **+ 2, 1**	3: + 1, 2	3: **+ 2, 1**	3: + 1, 2
4: ***a***	4: *c*	4: *c*	4: ***a***
5: *** 3, 2**	5: *d*	5: *d*	5: *** 3, 2**
6: ***a***	6: + 4, 5	6: + 4, 5	6: ***a***
7: **- 1, 4**	7: * 3, 6	7: * 3, 6	7: **- 1, 4**

3.6.1.2 Two-point recombination

Two crossover points are randomly chosen and the chromosomes exchange genetic material between the crossover points.

Example

Let us consider the parents C_1 and C_2 given below. Suppose that the crossover points were chosen after positions 2 and 5. In this case the offspring O_1 and O_2 are obtained:

Parents		Offspring	
C_1	C_2	O_1	O_2
1: ***b***	1: *a*	1: ***b***	1: *a*
2: *** 1, 1**	2: *b*	2: *** 1, 1**	2: *b*
3: **+ 2, 1**	3: + 1, 2	3: + 1, 2	3: **+ 2, 1**
4: ***a***	4: *c*	4: *c*	4: ***a***
5: *** 3, 2**	5: *d*	5: *d*	5: *** 3, 2**
6: ***a***	6: + 4, 5	6: ***a***	6: + 4, 5
7: **- 1, 4**	7: * 3, 6	7: **- 1, 4**	7: * 3, 6

3.6.1.3 Uniform recombination

During the process of uniform recombination, offspring genes are taken randomly from one parent or another.

Example

Let us consider the two parents C_1 and C_2 given below. The two offspring O_1 and O_2 are obtained by uniform recombination as follows:

Parents		Offspring	
C_1	C_2	O_1	O_2
1: ***b***	1: *a*	1: *a*	1: ***b***
2: *** 1, 1**	2: *b*	2: *** 1, 1**	2: *b*
3: **+ 2, 1**	3: + 1, 2	3: **+ 2, 1**	3: + 1, 2
4: ***a***	4: *c*	4: *c*	4: ***a***
5: *** 3, 2**	5: *d*	5: *** 3, 2**	5: *d*
6: ***a***	6: + 4, 5	6: + 4, 5	6: ***a***
7: **- 1, 4**	7: * 3, 6	7: **- 1, 4**	7: * 3, 6

3.6.2 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. To preserve the consistency of the chromosome, its first gene must encode a terminal symbol.

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

If the current gene encodes a terminal symbol, it may be changed into another terminal symbol or into a function symbol. In the later case, the positions indicating the function arguments 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).

Example

Consider the chromosome *C* given below. If the boldfaced symbols are selected for mutation an offspring *O* is obtained as follows:

C	*O*
1: *a*	1: *a*
2: * 1, 1	2: * 1, 1
3: ***b***	3: **+ 1, 2**
4: * 2, 2	4: * 2, 2
5: *b*	5: *b*
6: + **3**, 5	6: **+ 1**, 5
7: *a*	7: *a*

3.7 Handling exceptions within MEP

Exceptions are special situations that interrupt the normal flow of expression evaluation (program execution). An example of exception is *division by zero,* which is raised when the divisor is equal to zero.

Exception handling is a mechanism that performs special processing when an exception is thrown.

Usually, GP techniques use a *protected exception* handling mechanism [13]. For instance, if a division by zero exception is encountered, a predefined value (for instance 1 or the numerator) is returned.

GEP uses a different mechanism: if an individual contains an expression that generates an error, this individual receives the lowest fitness possible.

MEP uses a new and specific mechanism for handling exceptions. When an exception is encountered (which is always generated by a gene containing a function symbol), the gene that generated the exception is mutated into a terminal symbol. Thus, no infertile individual appears in a population.

3.8 MEP complexity

Let *NG* be the number of genes in a chromosome.

When solving symbolic regression, classification or any other problems for which the training set is known in advance (before computing the fitness), the fitness of an individual can be computed in O(*NG*) steps by dynamic programming [2]. In fact, a MEP chromosome needs to be read once for computing the fitness.

Thus, MEP decoding process does not have a higher complexity than other GP - techniques that encode a single computer program in each chromosome.

4 Solving Problems with Multi Expression Programming

In this section, several well-known problems belonging to different classes are solved using the MEP technique.

4.1 Symbolic regression

In this section, MEP technique is used for solving symbolic regression problems. The aim of symbolic regression is to discover a function that satisfies a set of fitness cases.

We also provide a comparison of MEP with standard GP, GEP and CGP.

Two well-known problems are used for testing the MEP ability of solving symbolic regression problems. The problems are:

i) The quartic polynomial [13]. Find a mathematical expression that satisfies best a set of fitness cases generated by the function:

$$f(x) = x^4 + x^3 + x^2 + x. \quad (5)$$

ii) The sextic polynomial [14]. Find a mathematical expression that satisfies best a set of fitness cases generated by the function:

$$f(x) = x^6 - 2x^4 + x^2. \quad (6)$$

A set of 20 fitness cases was randomly generated over the interval [-1.0, 1.0] and used in the experiments performed.

4.1.1 Experiment 1

The success rate of the MEP algorithm is analyzed in this experiment. The chromosome length is gradually increased. MEP algorithm parameters are given in Table 1.

Parameter	**Value**
Population size	30
Number of generations	50
Mutation	2 symbols / chromosome
Crossover type	Uniform-crossover
Crossover probability	0.9
Selection	Binary tournament
Terminal Set	$T = \{x\}$
Function Set	$F = \{+, –, *, /\}$

Table 1. Algorithm parameters for the Experiment 1.

The success rate of the MEP algorithm depending on the number of symbols in the chromosome is depicted in Figure 3.

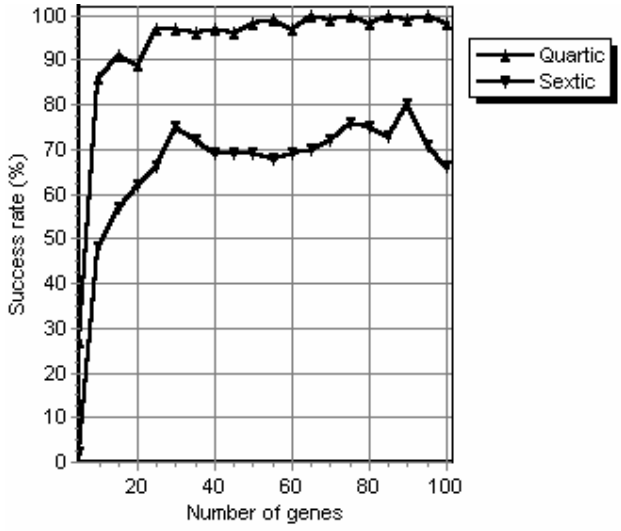

Figure 3. The relationship between the success rate and the number of symbols in a MEP chromosome. The number of symbols in chromosome varies between 5 and 100. The results are summed over 100 runs.

The success rate of the MEP algorithm increases with the chromosome length and never decreases towards very low values. When the search space (chromosome length) increases, an increased number of expressions are encoded by MEP chromosomes. Very large search spaces (very long chromosomes) are extremely beneficial for MEP technique due to its multi expression representation. This behavior is different from those obtained with the GP variants that encode a single solution in a chromosome (such as GEP). Figure 3 also shows that the sextic polynomial is more difficult to solve with MEP (with the parameters described in Table 1) than the quatic polynomial.

4.1.2 Experiment 2

From Experiment 1 we may infer that for the considered problem, the MEP success rate never decreases to very low values as the number of genes increases. To obtain an experimental evidence for this assertion longer chromosomes are considered. We extend chromosome length up to 300 genes (898 symbols).

The success rate of MEP is depicted in Figure 4.

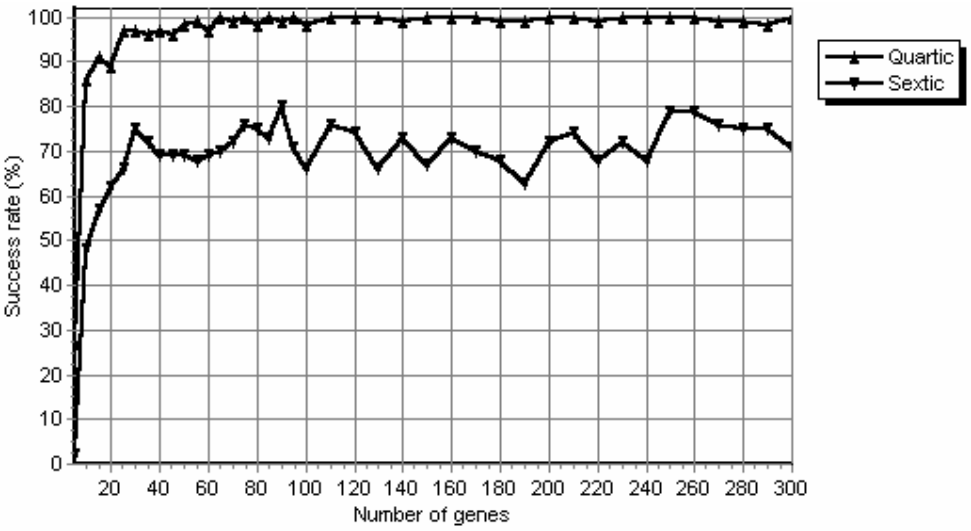

Figure 4. The relationship between the success rate and the number of symbols in a MEP chromosome. The number of symbols in chromosome varies between 5 and 300. The results are summed over 100 runs.

It can be seen from Figure 4 that, when solving the quartic (sextic) polynomial problem, the MEP success rate, lies in the interval [90, 100] ([60, 80]) for the chromosome length larger than 20.

One may note that after that the chromosome length becomes 10, the success rate never decrease more than 90% (for the quartic polynomial) and never decrease more than 60% (for the sextic polynomial). It also seems that, after a certain value of the chromosome length, the success rate does not improve significantly.

4.1.3 Experiment 3

In this experiment the relationship between the success rate and the population size is analyzed. Algorithm parameters for this experiment are given in Table 2.

Parameter	**Value**
Number of generations	50
Chromosome length	10 genes
Mutation	2 symbols / chromosome
Crossover type	Uniform-crossover
Crossover probability	0.9
Selection	Binary tournament
Terminal Set	$T = \{x\}$
Function Set	$F = \{+, –, *, /\}$

Table 2. Algorithm parameters for Experiment 3.

Experiment results are given in Figure 5.

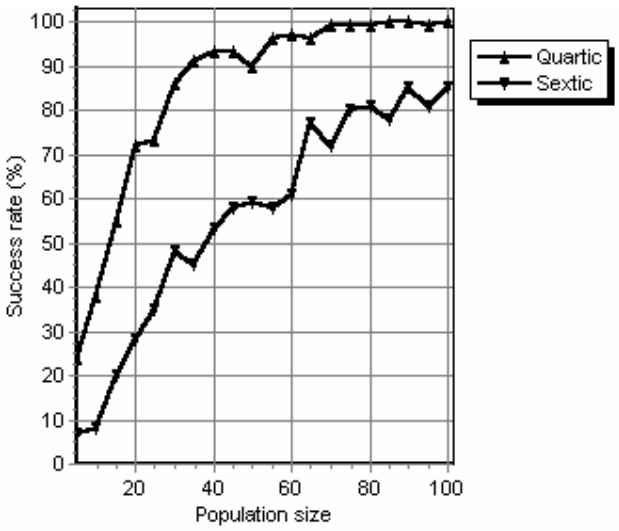

Figure 5. Success rate of the MEP algorithm. Population size varies between 5 and 100. The results are summed over 100 runs.

For the quartic problem and for the MEP algorithm parameters given in Table 2, the optimal population size is 70. The corresponding success rate is 99%. A population of 100 individuals yields a success rate of

88% for the sextic polynomial. This result suggests that even small MEP populations may supply very good results.

4.1.4 Experiment 4

In this experiment the relationship between the MEP success rate and the number of generations used in the search process is analyzed.

MEP algorithm parameters are given in Table 3.

Parameter	**Value**
Population size	20
Chromosome length	12 genes
Mutation	2 genes / chromosome
Crossover type	Uniform-crossover
Crossover probability	0.9
Selection	Binary tournament
Terminal Set	$T = \{x\}$
Function Set	$F = \{+, –, *, /\}$

Table 3. Algorithm parameters for Experiment 4.

Experiment results are given in Figure 5.

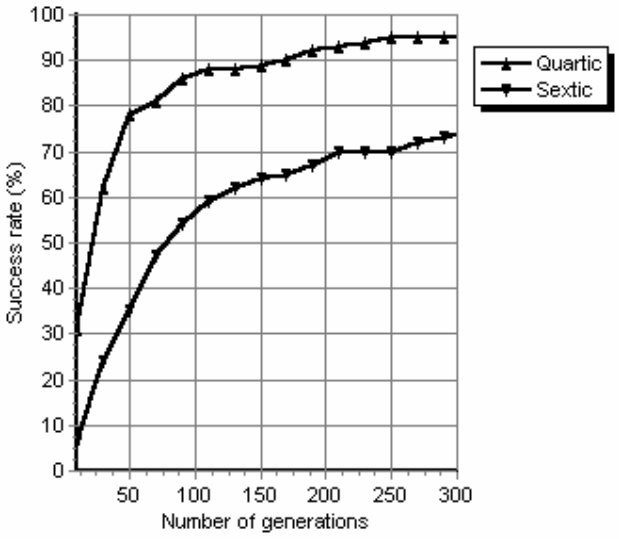

Figure 5. Relationship between the success rate of MEP algorithm and the number of generations used in the search process. The number of generations varies between 10 and 300. The results are summed over 100 runs.

The success rate of the MEP algorithm rapidly increases from 34%, respectively 8% (when the number of generations is 10) up to 95%, and 74% respectively (when the number of generations is 300).

4.1.5 MEP vs. GEP

In [6] GEP has been used for solving the quartic polynomial based on a set of 10 fitness cases randomly generated over the interval [0, 20]. Several numerical experiments analyzing the relationship between the success rate and the main parameters of the GEP algorithm have been performed in [6]. In what follows we will perform similar experiments with MEP.

The first experiment performed in [6] analyses the relationship between the GEP chromosome length and the success rate. GEP success rate increases up to 80% (obtained when the GEP chromosome length is 30) and then decreases. This indicates that very long GEP chromosomes cannot encode short expressions efficiently. The length of the GEP chromosome must be somehow related to the length of the expression that must be discovered.

Using the same parameter setting (i.e. Population Size = 30, Number of Generations = 50; Crossover probability = 0.7, Mutations = 2 / chromosome), the MEP obtained a success rate of 98% when the chromosome length was set to 20 genes (58 symbols).

The second experiment performed in [6] analyses the relationship between the population size used by the GEP algorithm and the success rate. For a population of size 30, the GEP success rate reaches 80%, and for a population of 50 individuals the GEP success rate reaches 90%.

Using the same parameter setting (i.e. Number of Symbols in Chromosome = 49 (17 MEP genes), Number of Generations = 50; Crossover probability = 0.7, Mutations = 2 / chromosome), MEP obtained a success rate of 99% (in 99 runs out of 100 MEP has found the correct solution) using a population of only 30 individuals.

In another experiment performed in [6], the relationship between the number of generations used by GEP and the rate of success is analysed. The success rate of 69% was obtained by GEP when the number of generations was 70 and a success rate of 90% was obtained only when the number of generations reached 500. For the considered generation range GEP success rate never reached 100%.

Using the same parameter setting (i.e. Number of Symbols in Chromosome = 80 (27 MEP genes), Population Size = 30; Crossover probability = 0.7, Mutations = 2 / chromosome), the MEP obtained a success rate of 97% (in 97 runs out of 100 MEP has found the correct solution) using 30 generations only. This is an improvement (regarding the number of generations used to obtain the same success rate) with more than one order of magnitude.

We may conclude that for the quartic polynomial, the MEP has a higher success rate than GEP using the previously given parameter settings.

4.1.6 MEP vs. Cartesian Genetic Programming

CGP has been used [] for symbolic regression of the sextic polynomial problem.

In this section, the MEP technique is used to solve the same problem using parameters settings similar to those of CGP. To provide a fair comparison, all experimental conditions described in [17] are carefully reproduced for the MEP technique.

CGP chromosomes were characterized by the following parameters: $n_r = 1$, $n_c = 10$, $l = 10$. MEP chromosomes are set to contain 12 genes (in addition MEP uses two supplementary genes for the terminal symbols {1.0, x}).

MEP parameters (similar to those used by CGP) are given in Table 4.

Parameter	**Value**
Chromosome length	12 genes
Mutation	2 genes / chromosome
Crossover type	One point crossover
Crossover probability	0.7
Selection	Binary tournament
Elitism size	1
Terminal set	$T = \{x, 1.0\}$
Function set	$F = \{+, -, *, /\}$

Table 4. Algorithm parameters for the MEP vs. CGP experiment.

In the experiment with CGP a population of 10 individuals and a number of 8000 generations have been used. We performed two experiments. In the first experiment, the MEP population size is set to 10 individuals and we compute the number of generations needed to obtain the success rate (61 %) reported in [17] for CGP.

When the MEP run for 800 generations, the success rate was 60% (in 60 runs (out of 100) MEP found the correct solution). Thus MEP requires 10 times less generations than CGP to solve the same problem (the sextic polynomial problem in our case). This represents an improvement of one order of magnitude.

In the second experiment, the number of generations is kept unchanged (8000) and a small MEP population is used. We are interested to see which is the optimal population size required by MEP to solve this problem.

After several trials, we found that MEP has a success rate of 70% when a population of 3 individuals is used and a success rate of 46% when a population of 2 individuals is used. This means that MEP requires 3 times less individuals than CGP for solving the sextic polynomial problem.

4.1.7 MEP vs. Standard GP

In [13] GP was used for symbolic regression of the quartic polynomial function.

GP parameters are given in Table 5.

Parameter	**Value**
Population Size	500
Number of generations	51
Crossover probability	0.9
Mutation probability	0
Maximum tree depth	17
Maximum initial tree depth	6
Terminal set	$T = \{x\}$
Function set	$F = \{+, -, *, \%, Sin, Cos, Exp, RLog\}$

Table 5. GP parameters used for solving the quartic problem.

It is difficult to compare MEP with GP since the experimental conditions were not the same. The main difficulty is related to the number of symbols in chromosome. While GP individuals may increase, MEP chromosomes have fixed length. Individuals in the initial GP population are trees having a maximum depth of 6 levels. The number of nodes in the largest tree containing symbols from $F \cup T$ and having 6 levels is $2^6 - 1 = 63$ nodes. The number of nodes in the largest tree containing symbols from $F \cup T$ and having 17 levels (maximum depth allowed for a GP tree) is $2^{17} - 1 = 131071$ nodes.

Due to this reason we cannot compare MEP and GP relying on the number of genes in a chromosome. Instead, we analyse different values for the MEP chromosome length.

MEP algorithm parameters are similar to those used by GP [13]. The results are depicted in Figure 6.

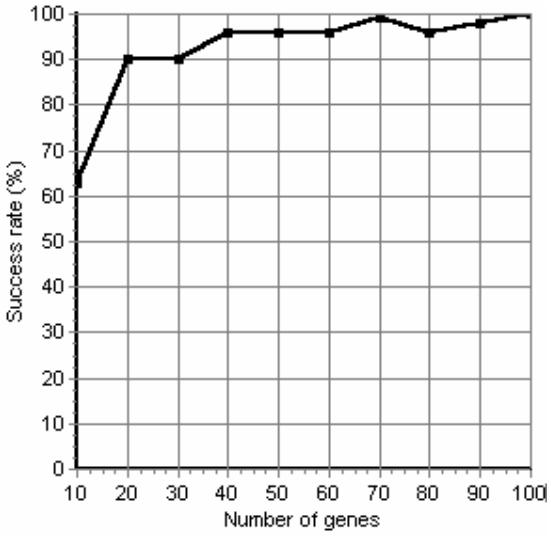

Figure 6. The relationship between the number of genes in a chromosome and the MEP success rate. The number of genes in a chromosome varies between 10 and 100. The results are averaged over 100 runs.

For this problem, the GP cumulative probability of success is 35% (see [13]). From Figure 6, it can be seen that the lowest success rate for MEP is 65%, while the highest success rate is 100% (for the considered chromosome length domain). Thus, MEP outperforms GP on the quartic polynomial problem (when the parameters given in Table 5 are used).

4.2 Even parity

The Boolean even-parity function of k Boolean arguments returns **T** (**True**) if an even number of its arguments are **T**. Otherwise the even-parity function returns **NIL** (**False**) [13].

In applying MEP to the even-parity function of k arguments, the terminal set T consists of the k Boolean arguments d_0, d_1, d_2, ... d_{k-1}. The function set F consists of four two-argument primitive Boolean functions: AND, OR, NAND, NOR. According to [13] the Boolean even-parity functions appear to be the most difficult Boolean functions to be detected via a blind random search.

The set of fitness cases for this problem consists of the 2^k combinations of the k Boolean arguments. The fitness of an MEP chromosome is the sum, over these 2^k fitness cases, of the Hamming distance (error) between the returned value by the MEP chromosome and the correct value of the Boolean function. Since the standardized fitness ranges between 0 and 2^k, a value closer to zero is better (since the fitness is to be minimized).

The parameters for these experiments are given in Table 6.

Parameter	**Value**
Number of generations	51
Crossover type	Uniform
Crossover probability	0.9
Mutation probability	0.2
Terminal set	$T_3 = \{D_0, D_1, D_2\}$ for even-3-parity $T_4 = \{D_0, D_1, D_2, D_3\}$ for even-4-parity
Function set	F = {AND, OR, NAND, NOR}

Table 6. The MEP algorithm parameters for the numerical experiments with even-parity problems.

In order to reduce the length of the chromosome all the terminals are kept on the first positions of the MEP chromosomes. The selection pressure is also increased by using higher values (usually 10% of the population size) for the q-tournament size.

Several numerical experiments with MEP have been performed for solving the even-3-parity and the even-4-parity problems. After several trials we have found that a population of 100 individuals having 300 genes was enough to yield a success rate of 100% for the even-3-parity problem and a population of 400 individuals with 200 genes yielded a success rate of 43% for the even-4-parity problem. GP without Automatically Defined Functions has been used for solving the even-3 and even-4 parity problems using a population of 4000 individuals [13]. The cumulative probability of success was 100% for the even-3-parity problem and 42% for the even-4-parity problem [13]. Thus, MEP outperforms GP for the even-3 and even-4 parity problems with more than one order of magnitude. However, we already mentioned, a perfect comparison between MEP and GP cannot be drawn due to the incompatibility of the respective representations.

4.3 Multiplexer

In this section, the MEP technique is used for solving the 6-multiplexer and the 11-multiplexer problems [13].

The input to the Boolean N-multiplexer function consists of k address bits a_i and 2^k data bits d_i, where

$N = k + 2^k$. That is, the input consists of the $k+2^k$ bits a_{k-1}, ... , a_1, a_0, d_2k_{-1}, ... , d_1, d_0. The value of the Boolean multiplexer function is the Boolean value (0 or 1) of the particular data bit that is singled out by the k address bits of the multiplexer. Another way to look at the search space is that the Boolean multiplexer function with $k+2^k$ arguments is a particular function of 2^{k+2^k} possible Boolean functions of $k+2^k$ arguments. For example, when k=3, then $k+2^k = 11$ and this search space is of size 22^{11}. That is, the search space is of size 2^{2048}, which is approximately 10^{616}.

The terminal set for the 6-multiplexer problem consists of the 6 Boolean inputs, and for the 11-multiplexer problem consists of the 11 Boolean inputs. Thus, the terminal set T for the 6-multiplexer is of $T = \{a_0, a_1, d_0, d_1, \dots , d_4\}$ and for the 11-multiplexer is of $T = \{a_0, a_1, a_2, d_0, d_1, \dots , d_7\}$.

The function set F for this problem is F = {AND, OR, NOT, IF} taking 2, 2, 1, and 3 arguments, respectively [13]. The function IF returns its 3rd argument if its first argument is set to 0. Otherwise it returns its second argument.

There are 2^{11} = 2,048 possible combinations of the 11 arguments $a_0a_1a_2d_0d_1d_2d_3d_4d_5d_6d_7$ along with the associated correct value of the 11-multiplexer function. For this particular problem, we use the entire set of 2048 combinations of arguments as the fitness cases for evaluating fitness.

4.3.1 Experiments with 6-multiplexer

Two main analyses are of high interest: the relationship between the success rate and the number of genes in a MEP chromosome and the relationship between the success rate and the size of the population used by the MEP algorithm. For these experiments the parameters are given in Table 7.

Parameter	**Value**
Number of generations	51
Crossover type	Uniform
Crossover probability	0.9
Mutation probability	0.1
Terminal set	$T = \{a_0, a_1, d_0, d_1, \dots , d_4\}$

Function set	F = {AND, OR, NOT, IF}

Table 7. MEP algorithm parameters for the numerical experiments with 6-multiplexer problem.

A population of 100 individuals is used when the influence of the number of genes is analysed and a code length of 100 genes is used when the influence of the population size is analysed. For reducing the chromosome length we keep all the terminals on the first positions of the MEP chromosomes. We also increased the selection pressure by using larger values (usually 10% of the population size) for the tournament sample.

The results of these experiments are given in Figure 8.

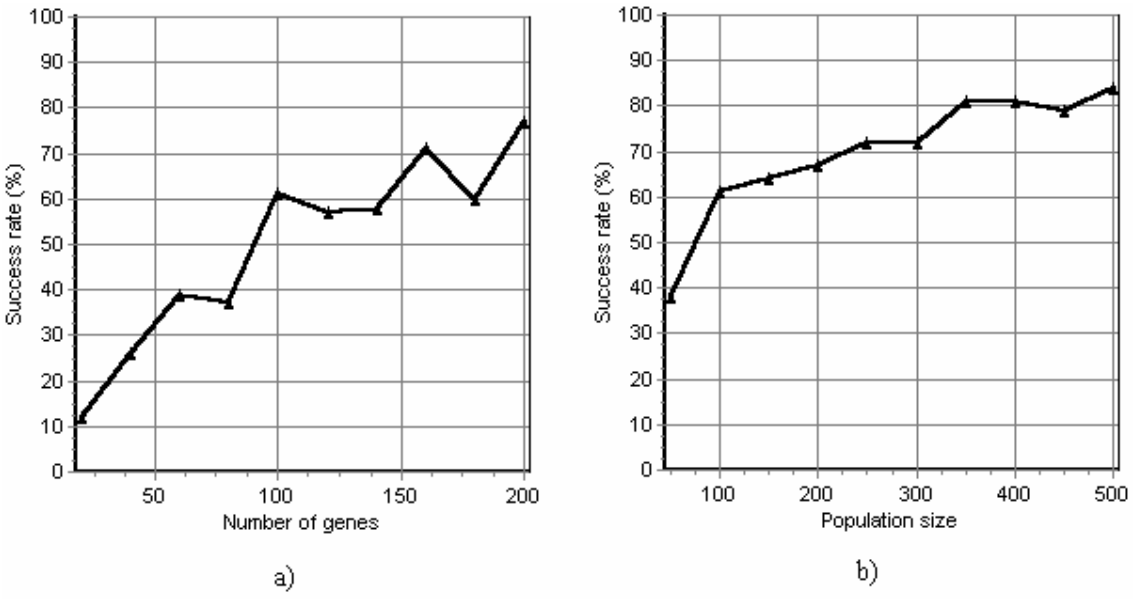

Figure 8. The success rate of the MEP algorithm for solving the 6-multiplexer problem.
(a) The relationship between the success rate and the chromosome length.
(b) The relationship between the success rate and the population size.
Results are summed over 100 runs.

From Figure 8 it can be seen that MEP is capable to solve the 6-multiplexer problem very well. A population of 500 individuals yields a success rate of 84%. A similar experiment using the GP technique with a population of 500 individuals has been reported in [21]. The reported probability of success is a little less (79,5%) than the one obtained with MEP (84%).

4.3.2 Experiments with 11-multiplexer

We also performed several experiments with the 11-multiplexer problem. We have used a population of 500 individuals and three values (100, 200 and 300) for the number of genes in a MEP chromosome. In all these experiments, MEP was able to find a perfect solution (out of 30 runs), thus yielding a success rate of 3.33%. When the number of genes was set to 300, the average of the best fitness of each run taken as a percentage of the perfect fitness was 91.13%, with a standard deviation of 4.04. As a comparison, GP was not able to obtain a perfect solution by using a population of 500 individuals and the average of the best fitness of each run taken as a percentage of the perfect fitness was 79.2% (as reported in [21]).

4.4 Discovering game strategies with MEP

In this section we investigate the application of MEP technique for discovering game strategies.

Koza [11] suggested that GP can be applied to discover game strategy. The game-playing strategy may be viewed as a computer program that takes the information about the game as its input and produces a move as output.

The available information may be an explicit history of previous moves or an implicit history of previous moves in the form of a current state of game (e.g. the position of each piece on the chess board) [11].

Tic-tac-toe (TTT, or naughts and crosses) is a game with simple rules, but complex enough to illustrate the ability of MEP to discover game strategy.

4.4.1 TTT game description

In Tic-Tac-Toe there are two players and a 3×3 grid. Initially the grid is empty. Each player moves in turn by placing a marker in an open square. By convention, the first player's marker is "X" and the second player's marker is "0".

The player that put three markers of his type ("X" for the first player and "0" for the second player) in a row is declared the winner.

The game is over when one of the players wins or all squares are marked and no player wins. In the second case, the game ends with draw (none of the players win). Enumerating the game tree shows that the second player can obtain at least a draw.

A well-known evolutionary algorithm that evolves game strategy has been proposed in [4]. This algorithm will be reviewed in the next section.

4.4.2 Chellapilla's approach of TTT

In [5] an evolutionary algorithm has been used in order to obtain a good strategy (that never loses) for the Tic-Tac-Toe game. A strategy is encoded in a neural network. A population of strategies encoded by neural networks is evolved.

Each network receives a board pattern as input and yields a move as output. The aim is to store in a neural network the function that gives the quality of a configuration. When a configuration is presented to the network, the network output (supplies) the next move.

Each neural network has an input layer with 9 nodes, an output layer with 9 nodes, and a hidden layer with a variable number of nodes.

Fogel's algorithm starts with a random population of 50 neural networks. For each network the number of nodes from the hidden layer is randomly chosen with a uniform distribution over integers 1...10. The initial weighted connection strengths and bias terms are randomly distributed according to a uniform distribution ranging over [-0.5, 0.5].

From each parent a single offspring is obtained by mutation. Mutation operator affects the hidden layer structure, weight connections and bias terms.

Each strategy encoded in a neural network was played 32 times against a heuristic rule base procedure.

The payoff function has several values corresponding to winning, loosing and draw.

The best individuals from a generation are retained to form the next generation.

The process is evolved for 800 generations. According to [4], the best obtained neural network is able to play to win or draw with a perfect play strategy.

4.4.3 MEP approach of TTT

In this section we illustrate the use of MEP to discover an unbeatable play strategy for Tic-Tac-Toe.

We are searching for a mathematical function *F* that gives the quality of each game configuration. Using this function the best configurations that can be reached in one move from the current configuration, is selected. Therefore function *F* supplies the move to be performed for each game configuration.

Function *F* evaluating each game configuration is represented as a MEP chromosome. The best expression encoded by a chromosome is chosen to be the game strategy of that chromosome.

Without any loose of generality we may allow MEP strategy to be the first player in each game.

All expressions in the chromosome are considered in the fitness assignment process. Each expression is evaluated using an "all-possibilities" procedure. This procedure executes all moves that are possible for the second player. The fitness of an expression *E* is the number of games that the strategy encoded by the expression *E* loses. Obviously the fitness has to be minimized.

Let us denote by

$P = (p_0, p_1 \ldots p_8)$

a game configuration.

Each p_k describes the states "X", "0" or an empty square. In our experiments the set {5, -5, 2} has been used for representing the symbols "X", "0" and the empty square.

The game board has been linearized by scanning board squares from up to down and from left to right. Thus the squares in the first line have indices 0, 1 and 2, etc. (see Figure 9).

0	1	2
3	4	5
6	7	8

Figure 9. Game board linearized representation.

Algorithm parameters are given in Table 8.

Parameter	**Value**
Population size	50
Chromosome length	50 genes
Mutation probability	0.05
Crossover type	Two-point-crossover
Selection	Binary tournament
Elitism size	1
Terminal set	$T = \{p_0, p_1, \ldots, p_8\}$
Function set	$F = \{+, -, *, /\}$

Table 8. Algorithm parameters for TTT game.

The MEP algorithm is able to evolve a *perfect*, non-loosing, game strategy in 11 generations. This process requires less than 10 seconds when an Intel Pentium 3 processor at 1GHz is used.

In Figure 10, the fitness of the best individual in the best run and average fitness of the best individuals over all runs are depicted.

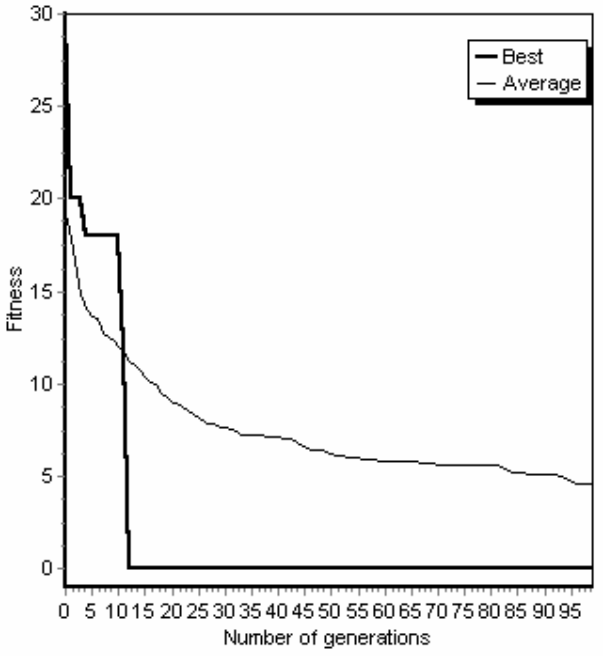

Figure 10. Fitness of the best individual in the best runs and the average fitness of the best individuals over all runs. The results are taken over 30 runs.

Form Figure 10 it can be seen that an individual representing a non-loosing strategy appears in the population at generation 14.

Some functions evolved by the MEP algorithm are given below:

$$F_1(P) = ((p_4-p_5-(p_6+p_5))*p_8+p_4*p_3)*(p_4-p_7), \quad (19)$$

$$F_2(P) = p_2-(p_8*p_7-p_4)-p_7-(p_2*p_5), \quad (20)$$

$$F_3(P) = (p_4*p_1+p_2)*p_7-(p_1-p_2+p_7*p_5)-(p_8-(p_3*p_5)). \quad (21)$$

These functions do not force the win when it is possible, but they never lose. This is a consequence of fitness assignment process. However, proposed technique can also generate a function that forces the win whenever it is possible.

It is not easy to compare this result with the result obtained by Chellapilla and Fogel [4] as the experiment conditions were not the same. In [4] evolved strategies play against a heuristic procedure, but here MEP formulas play against an all-moves procedure. Population size was the same (50 individuals). Individual sizes are difficult to be compared. All MEP individual have the same size: 148 symbols. Neural network's sizes used in [4] are variable since the hidden layer contains a variable number of nodes. If the number of nodes in the hidden layer is 9, then the size of the neural network (biases + connection weights) is 9 * 9 + 9 * 9 * 9 + 9 * 9 = 324.

MEP approach seems to be faster as MEP was able to discover a non-losing strategy in no more than 17 generations. As noted in [4] neural network approach requires 800 generations.

4.4.4 A good TTT heuristic vs. MEP

A good heuristic for Tic-Tac-Toe is described in what follows:

S1. *If a winning move is available make that move, else*
S2. *If a winning move is available for the opponent, move to block it, else*
S3. *If a move of the opponent that leads to two winning ways is available, block that move, else*
S4. *If the board center is available, move in the board center,*
S5. *If one ore more corners of the table are available, move in one of them, else*
S6. *Move randomly in an available square.*

This heuristic performs well on most of the game positions. However, by applying one of the formulas evolved by the MEP algorithm some benefits are obtained:

- easy implementation in programming languages,
- MEP evolved formula is a faster algorithm than the previously shown heuristic.

4.4.5 Applying MEP for generating complex game strategies

Using the "all-possibilities" technique (a backtracking procedure that plays all the moves for the second player) allows us to compute the absolute quality of a game position.

For complex games a different fitness assignment technique is needed since the moves for the second player can not be simulated by an "all-possibilities" procedure (as the number of moves that needs to be simulated is too large).

One fitness assignment possibility is to use a heuristic procedure that acts as the second player. But there are several difficulties related to this approach. If the heuristic is very good it is possible that none of evolved strategy could ever beat the heuristic. If the heuristic procedure plays as a novice then many evolved strategy could beat the heuristic from the earlier stages of the search process. In the last case fitness is not correctly assigned to population members and thus we can not perform a correct selection.

A good heuristic must play on several levels of complexity. At the beginning of the search process the heuristic procedure must play at easier levels. As the search process advances, the level of difficulty of the heuristic procedure must increases.

However, for complex games such a procedure is difficult to implement.

Another possibility is to search for a game strategy using a coevolutionary algorithm [4]. This approach seems to offer the most spectacular results. In this case, MEP population must develop intelligent behavior based only on internal competition.

5 Conclusions and further work

Multi Expression Programming is a new evolutionary technique that may be used for evolving computer programs. MEP technique uses a new solution representation and specific search operators.

MEP correctly identified several important trends in nowadays Evolutionary Algorithms:

- encoding multiple solutions in single chromosome,

- individual encoding is similar to nowadays machine code (MEP representation is similar to the way in which C and Pascal compilers evaluate expressions),

- low complexity of fitness assignment process (MEP individuals are read only once for computing the fitness).

It is documented that MEP technique can be used for solving various classes of difficult problems. Several numerical experiments have been performed with MEP. For the considered problems MEP algorithm performs the as well as - and sometimes even better than - other similar evolutionary techniques (such as GP, GEP and CGP).

Further efforts will be dedicated for applying MEP to other real world difficult problems namely prediction of complex phenomena, classification, discovering heuristics for NP-Complete [7] problems, discovering strategies for complex games, etc.

References

[1] A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, 1986.

[2] R. Bellman, *Dynamic Programming*, Princeton, Princeton University Press, New Jersey, 1957.

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

[4] K. Chellapilla, D. B. Fogel, *Evolution, Neural Networks, Games and Intelligence*, Proc. IEEE. 87, 1471-1496, 2000.

[5] D. Dumitrescu, B. Lazzerini, L. Jain, A. Dumitrescu, *Evolutionary Computation*. CRC Press, Boca Raton, FL, 2000.

[6] C. Ferreira, *Gene Expression Programming: a New Adaptive Algorithm for Solving Problems*, Complex Systems, 13**,** 87-129, 2001.

[7] N. F. McPhee and N. J. Hopper. *Analysis of Genetic Diversity Through Population History*. W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, (editors), Proceedings of the Genetic and Evolutionary Computation Conference, 2, 1112–1120, Morgan Kaufmann, 1999.

[8] M.R. Garey, D.S. Johnson, *Computers and Intractability: A Guide to NP-completeness*, Freeman & Co, San Francisco, 1979.

[9] D.E., Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA, 1989.

[10] S. Handley, *On the Use of a Directed Graph to Represent a Population of Computer Programs*. Proceeding of the IEEE World Congress on Computational Intelligence, 154-159, Orlando, Florida, 1994.

[11] W. Kantschik, W Banzhaf, *Linear-Tree GP and its comparison with other GP structures*, In J. Miller, M. Tomassini, P.-L. Lanzi, C. Ryan, A. Tettamanzi and W. B. Langdon editors, Proceedings of 4th EuroGP Conference, Springer, 302 – 312, 2001.

[12] M.A. Lones, A.M. Tyrrell, *Biomimetic Representation in Genetic Programming*, Proceedings of the Workshop on Computation in Gene Expression at the Genetic and Evolutionary Computation Conference 2001 (GECCO2001), July 2001.

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

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

[15] J. R. Koza, *Genetic Programming III: Darwinian Invention and Problem Solving*, Morgan Kaufmann, San Francisco, CA 1999.

[16] M. O'Neill, C. Ryan, *Grammatical Evolution*, IEEE Transaction on Evolutionary Computation, 5, 2001.

[17] J. Miller, P. Thomson, *Cartesian Genetic Programming*, Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin and Terence C. Fogarty (editors) Proceedings of EuroGP 2000, Springer, 121-132, 2000.

[18] P. Nordin, *A Compiling Genetic Programming System that Directly Manipulates the Machine Code*, K. E. Kinnear, Jr. (editor), Advances in Genetic Programming, 311-331, MIT Press, 1994.

[19] N.R. Patterson, Genetic programming with context-sensitive grammars, PhD thesis, University of St Andrews, Scotland, 2003.

[20] R. Poli, W. B. Langdon, *Sub-machine Code Genetic Programming*, Technical report CSRP -98-18, School of Computer Science, University of Birmingham, 1998.

[21] U.M., O'Reilly, F. Oppacher, *A Comparative Analysis of Genetic Programming*, Advances in Genetic Programming 2, Peter J. Angeline and Kenneth E. Kinnear (editors), MIT Press, 23-44, 1996.

[22] A. Singleton, *Genetic Programming with C++*, BYTE, 171-176, 1994.

[23] G. Syswerda, *Uniform Crossover in Genetic Algorithms*, Schaffer, J.D., (editor), Proceedings of the 3rd International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, San Mateo, CA, 2–9, 1989.

[24] W. B. Langdon and R. Poli. *Genetic Programming Bloat with Dynamic Fitness*. W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty (editors), Proceedings of the First European Workshop on Genetic Programming, Springer-Verlag, 96-112, 1998.

[25] Wolpert D.H., McReady W.G., *No Free Lunch Theorems for Optimisation*, IEEE Transaction on Evolutionary Computation, Vol. 1, 67-82, 1997.