Multi Expression Programming - an in-depth description

Mihai Oltean

Department of Computer Science, Babeş-Bolyai University
Kogalniceanu 1, Cluj-Napoca, 400084, Romania

mihai.oltean@gmail.com
https://mepx.org

June 4, 2006

Multi Expression Programming (MEP) [17, 18, 24, 25] is a Genetic Programming variant that uses a linear representation of chromosomes. MEP individuals are strings of genes encoding complex computer programs.

When MEP individuals encode expressions, their representation is similar to the way in which compilers translate *C* or *Pascal* expressions into machine code [2].

A unique 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 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).

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

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.

1 Chapter structure

The chapter is structured as follows:

MEP representation is described in section 2. The way in which a MEP chromosome is initialized is described in section 3.

Fitness assignment process is described in section 4.

Genetic operators used in conjunction with MEP are introduced in section 7.

MEP representation is compared to other GP representations in section 6.

The way in which MEP handles exceptions is described in section 8.

The complexity of the fitness assignment process is computed in section 9.

MEP algorithm is given in section 10.

Automatically Defined Function in MEP are introduced in section 11.

Tips for a good implementation of the MEP technique are suggested in section 15.

Further research directions and open problems are indicated in section 15.

2 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.

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

Example

Consider a representation where the numbers on the left 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 C 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, 5
8: + 2, 6

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

$$Number_of_Symbols = (n+1) * (Number_of_Genes - 1) + 1,$$

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

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

3 Initialization

There are some restrictions for generating a valid MEP chromosome:

- First gene of the chromosome must contain a terminal. If we have a function in the first position we also need some pointer to some positions with lower index. But, there are no other genes above the first gene.

- For all other genes which encodes functions we have to generate pointers toward function arguments. All these pointers must indicate toward genes which have a lower index than the current gene.

4 Fitness assignment

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 7 encodes the expression:

$E_6 = c + d.$

Gene 7 indicates the operation * on the operands located at position 3 and 5. Therefore this gene encodes the expression:

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

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

$E_8 = b * (c + d)$.

There is neither practical nor theoretical evidence that one of these expressions is better than the others. Moreover, Wolpert and McReady [34, 35] 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:

$E_1 = a$,
$E_2 = b$,
$E_3 = a + b$,
$E_4 = c$,
$E_5 = d$,
$E_6 = c + d$,
$E_7 = (a + b) * d$,
$E_8 = b * (c + d)$.

The value of these expressions may be computed by reading the chromosome top down. Partial results are computed by Dynamic Programming [4] 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. Figure 4 shows the forest of expressions encoded by the previously presented MEP chromosome.

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:

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

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.

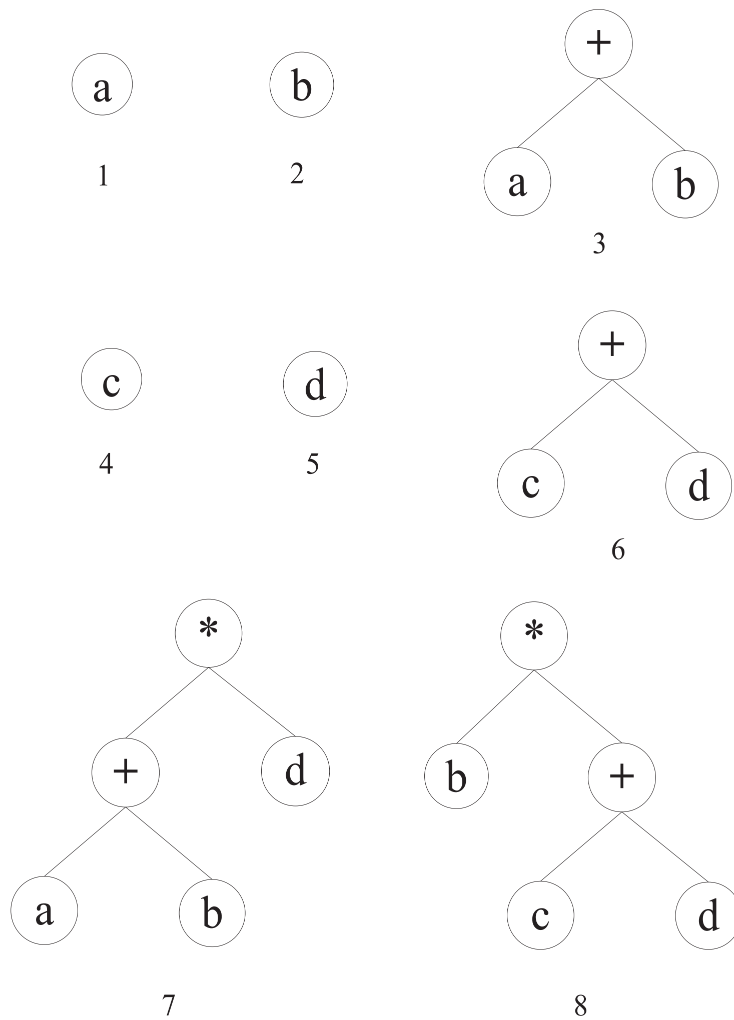

Figure 1: Expressions encoded by a MEP chromosome represented as trees

Table 1: An example for the MEP fitness assignment process. The first column contains the expression encoded into the MEP chromosome. The second column contains the value of these expressions. These values are computed by replacing the variables with their values. The last column represents the fitness of each expression which is the difference in absolute value between value of each expression and the expected output for the current fitness case (which has the value 10 in our example).

Expressions	Value	Fitness
$E_1 = a$	7	3
$E_2 = b$	2	8
$E_3 = a + b$	9	1
$E_4 = c$	1	9
$E_5 = d$	5	5
$E_6 = c + d$	6	4
$E_7 = (a+b)*d$	45	35
$E_8 = b*(c+d)$	12	2

The fitness of an individual is set to be equal to the lowest fitness of the expressions encoded in the chromosome:

$$f(C) = \min_i f(E_i). \tag{2}$$

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. An example on how to compute the fitness of a MEP chromosome is given in Table 1. Suppose that we have to solve a regression problem with a single example for training. Numerical values for inputs are as follows: $a = 7, b = 2, c = 1, d = 5$ and the expected output is 10.

Table 1 provides a simple example on how to compute the fitness of a MEP chromosome. In this example the expression $E_3 = a + b$ provides the lowest value for fitness and thus this expression will represent the chromosome.

Let us take another example that contains two fitness cases. The first one is described by the values $a = 7, b = 2, c = 1, d = 5$ and the expected output is 10. The second fitness case is given by the inputs $a = 12, b = 3, c = 9, d = 1$ and the expected output is 7. Numerical values for this example

Table 2: Another example for the MEP fitness assignment process. The first column contains the expression encoded into the MEP chromosome. The second column contains the value of these expressions for the first fitness case. The third column contains the value of these expressions for the second fitness case. The last column represents the fitness of each expression which is the sum of the fitnesses for each of the two training cases.

Expressions	Value (case 1)	Value (case 2)	Fitness
$E_1 = a$	7	12	3+5=8
$E_2 = b$	2	3	8+4=12
$E_3 = a + b$	9	15	1+8=9
$E_4 = c$	1	9	9+2=11
$E_5 = d$	5	1	5+6=11
$E_6 = c + d$	6	10	4+3=7
$E_7 = (a+b)*d$	45	15	35+8=43
$E_8 = b*(c+d)$	12	30	2+23=25

are given in Table 2.

The example shown in Table 2 is a little bit more complicated. It contains two training (fitness) cases. For computing the fitness of an expression we had to compute the fitness for each training case and then to add these 2 values. The best expression (that will be used to represent the chromosome) is $E_6 = c + d$ in this example. It provided the lowest fitness among all expressions encoded in that chromosome as can be seen from Table 2.

In is obvious (see Tables 1 and 2 that some parts of a MEP chromosome are not used. Some GP techniques, like Linear GP, remove non-coding sequences of chromosome during the search process. As already noted [5] 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.

5 MEP fitness in the case of multiple outputs problems

In this section we describe the way in which Multi Expression Programming may be efficiently used for handling problems with multiple outputs (such

as designing digital circuits [25]).

Each problem has one or more inputs (denoted by NI) and one or more outputs (denoted NO). In section 4 we have presented the way in which is the fitness of a chromosome with a single output is computed.

When multiple outputs are required for a problem, we have to choose NO genes which will provide the desired output (it is obvious that the genes must be distinct unless the outputs are redundant).

In Cartesian Genetic Programming (see chapter **CGP**), the genes providing the program's output are evolved just like all other genes. In MEP, the best genes in a chromosome are chosen to provide the program's outputs. When a single value is expected for output we simply choose the best gene (see section 4).

When multiple genes are required as outputs we have to select those genes which minimize the difference between the obtained result and the expected output.

We have to compute first the quality of a gene (sub-expression) for a given output (how good is that gene for providing the result for a given output):

$$f(E_i, q) = \sum_{k=1}^{n} |o_{k,i} - w_{k,q}|, \tag{3}$$

where $o_{k,i}$ is the obtained result by the expression (gene) E_i for the fitness case k and $w_{k,q}$ is the targeted result for the fitness case k and for the output q. The values $f(E_i,\ q)$ are stored in a matrix (by using dynamic programming [4] for latter use (see formula (4)).

Since the fitness needs to be minimized, the quality of a MEP chromosome may be computed by using the formula:

$$f(C) = \min_{i_1, i_2, \ldots, i_{NO}} \sum_{q=1}^{NO} f(E_{i_q}, q). \tag{4}$$

In equation (4) we have to choose numbers $i_1,\ i_2,\ \ldots,\ i_{NO}$ in such way to minimize the program's output. For this we shall use a simple heuristic which does not increase the complexity of the MEP decoding process: for each output q ($1 \leq q \leq NO$) we choose the gene i that minimize the quantity $f(E_i,\ q)$. Thus, to an output is assigned the best gene (which has not been assigned before to another output). The selected gene will provide the value of the qth output.

Table 3: An example on how to compute genes which will provides outputs. The problem has 3 outputs and the MEP chromosome has 5 genes. The value within cell (i, j) is the fitness provided by gene i for output j as computed by equation 3. Genes providing the outputs have been marked with *

Gene#	Output_1	Output_2	Output_3
0	5	3	9
1	7	6	7
2	1 *	0	2
3	4	1 *	5
4	2	3	4 *

Example

Let's consider a problem with 3 outputs and a MEP chromosome with 5 genes. We compute the fitness of each gene for each output as described by equation 3 and we store the results in a matrix. An example of matrix filled using equation 3 is given in Table 3.

The way in which genes providing the outputs have been selected is describes in what follows: first we need to compute the gene which will provide the result for the first output. We check the 2nd column in Table 3 and we see that the minimal value is 1 which correspond to the gene #2. Thus the result for the first output of the problem will be provided by gene #2. For the second output we see that gene #2 is generating the lowest fitness. We cannot choose this gene again because it already provides the result for the first output. Thus we choose the gene #3 for providing the result for the second output. Acording to the same algorithm (described above) the result for the third output will be provided by the gene #4.

Remark

Formulas (3) and (4) are the generalization of formulas (1) and (2) for the case of multiple outputs of a MEP chromosome.

The complexity of the heuristic used for assigning outputs to genes is:

$\mathrm{O}(NG \cdot NO)$,

where NG is the number of genes and NO is the number of outputs.

We may use another procedure for selecting the genes that will provide the problem's outputs. This procedure selects, at each step, the minimal value in the matrix $f(E_i, q)$ and assign the corresponding gene i to its paired output q. Again, the genes already used will be excluded from the search. This procedure will be repeated until all outputs have been assigned to a gene. However, we did not used this procedure because it has a higher complexity – $O(NO \cdot log_2(NO) \cdot NG)$ - than the previously described procedure which has the complexity $O(NO \cdot NG)$.

6 MEP representation vs. GP, LGP, GEP and CGP representations

Generally a GP chromosome encodes a single expression (computer program). This is also the case for GEP, GE and LGP 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).

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. A related work could be considered the Handley's idea [7] which stored the entire population of GP trees in a single graph. In this way a lot of memory is saved.

6.1 Multi Expression Programming versus Linear Genetic Programming

There are many similar aspects of MEP and LGP representations. Both chromosomes can be interpreted as a sequence of instructions that can be translated into programs of an imperative language.

However, there are also several important aspects that makes these two methods different:

- LGP modifies an array of variables. Each variable can be modified multiple times by different instructions. MEP computes the outputs generated by each gene and stores them into an array, but the obtained values are not overwritten by other values.

- The variable providing the output in the case of LGP is chosen at the beginning of the search process, whereas, in the case of MEP, the gene providing the output could dynamically change at each generation.

Linear GP [5] 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. This idea is later exploited in **Multi Solution Chapter**.

6.2 Multi Expression Programming vs. Gene Expression Programming

There are several major differences between GEP and MEP:

- The pointers toward function arguments are encoded explicitly in the MEP chromosomes whereas in GEP these pointers are computed when the chromosome is parsed. MEP also needs to evolve the pointers toward function arguments. This leads to a more compact representation of GEP when compared to MEP.
- MEP representation is suitable for code reuse, whereas GEP representation is not.
- The noncoding regions of GEP are always at the end of the chromosome, whereas in MEP these regions can appear anywhere in the chromosome.

It can be seen that the effective length of the expression may increases exponentially with the length of the MEP 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.

6.3 Multi Expression Programming versus Cartesian Genetic Programming

MEP representation is similar to GP and CGP, in the sense that each function symbol provides pointers towards its parameters.

There are 3 main differences between MEP and CGP:

- Within CGP the inputs are not stored in the chromosome. In MEP, the inputs are stored in any position of the chromosome. This fact is a consequence of the motivations that have lead to the proposal of these methods: CGP was proposed for designing digital circuits whereas MEP was motivated by the way in which C and Pascal compilers translate mathematical expressions into machine code [2].

- Within CGP the outputs are evolved along the other genes of the chromosome. In MEP all outputs are checked and the best of them is chosen to represent the chromosome.

- CGP stores a single solution / chromosome, whereas MEP stores multiple solutions in the same chromosome. Note, that both procedures have the same complexity which means the same running time.

7 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.

7.1 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.

7.1.1 One-point recombination

One-point recombination operator in MEP representation is similar to the corresponding binary representation operator. 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 given in Table 7.1.1.

Table 4: MEP one-point recombination.

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**

7.1.2 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 as given in Table 7.1.2.

Table 5: MEP two-point recombination.

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

7.1.3 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 given in Table 6.

Table 6: MEP uniform recombination.

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

7.2 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 given in Table 7.

Table 7: MEP mutation. Third and sixth genes have been affected by mutation

C	O
1: a	1: a
2: * 1, 1	2: * 1, 1
3: $\boldsymbol{b}$	3: **+ 1, 2**
4: * 2, 2	4: * 2, 2
5: b	5: b
6: + **3**, 5	6: **+ 1**, 5
7: a	7: a

8 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 [8]. For instance, if a division by zero exception is encountered, a predefined value (for instance 1 or the numerator) is returned.

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 may appear in a population.

There are some special issues that must be taken into account when implementing MEP for problems that might generate exceptions (such as division by 0). Because MEP chromosomes might be changed during the evaluation process it might be necessarily to recompute the some values for the previously (before the exception occurred) computed fitness cases.

Example

Let's consider a problem with 3 fitness cases and the current evaluated chromosome encodes the expression $a/b + b$. If the 3rd fitness case is $a = 1$, $b = 0$, an exception will occur and the corresponding gene (which contains the function symbol /) will be mutated into a terminal symbol (either a or b). In this case we need to recompute the value of the newly obtained expression for the fitness cases 1 and 2 and, of course, we have to compute its value for the remaining fitness case 3.

For avoiding extra computations we may store partial results for all MEP genes and all fitness cases. Thus, we will need a $NumberOfGenes$ x $NumberOfFitnessCases$ matrix whose elements will be computed row by row. First of all we will compute the value of the expression encoded by the first gene for all fitness cases. Next, we move to the second gene and we compute the its value for all fitness cases. When an exception is encountered we need to recompute only the values of the current row. All other values (previously computed) remain unchanged. An example is depicted in Figure 8.

The complexity of the storage space is:

$$O(NumberOfGenes * NumberOfFitnessCases).$$

Remark

For problems where exceptions are not encountered (e.g. digital circuits design) there is no need for storing this matrix. In this case we can compute

	$a = 6$ $b = 2$	$a = 24$ $b = 6$	$a = 1$ $b = 0$
0: a	6	24	1
1: b	2	6	0
2: / 0,1	3	4	DivBy Zero!!!
3: +2,1	-	-	-

	$a = 6$ $b = 2$	$a = 24$ $b = 6$	$a = 1$ $b = 0$
0: a	6	24	1
1: b	2	6	0
2: a	6	24	1
3: +2,1	8	30	1

Figure 2: An example on how to compute the value of the expressions encoded into a MEP chromosome. First column contains them MEP chromosome. First line contains three fitness cases. All other cells contains the value of the current expression for the current fitness case. An exception occurred in the 3rd gene for the 3rd fitness case (see table in the top of the figure). We need to recompute only the values for the gene which generated the exception (4rd row in our case). The values in all other rows (2 and 3) are preserved. At the end we may compute the values in the 4th gene

the value of the expressions, encoded by a MEP chromosome, fitness case by fitness case. Only linear memory storage is required in this case.

9 Complexity of the fitness computing process

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(*CodeLength*) steps by dynamic programming [4]. 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. Roughly speaking this means the same running time for MEP and all other GP methods that parse the chromosome once in order to compute the fitness.

10 MEP algorithm

Standard MEP algorithm uses steady-state evolutionary model [33] 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

S_1. Randomly create the initial population $P(0)$
S_2. **for** $t = 1$ **to** *Max_Generations* **do**
S_3. **for** $k = 1$ **to** $|P(t)|$ / 2 **do**
S_4. p_1 = *Select*($P(t)$); // select one individual from the current population
S_5. p_2 = *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
S15. endfor
```

11 Automatically Defined Functions in MEP

In this section we describe the way in which the Automatically Defined Functions [9] are implemented within the context of Multi Expression Programming.

The necessity of using reusable subroutines is a day-by-day demand of the software industry. Writing reusable subroutines proved to reduce:

(i) the size of the programs.

(ii) the number of errors in the source code.

(iii) the cost associated with the maintenance of the existing software.

(iv) the cost and the time spent for upgrading the existing software.

As noted by Koza [9] function definitions exploit the underlying regularities and symmetries of a problem by obviating the need to tediously rewrite lines of essentially similar code. Also, the process of defining and calling a function, in effect, decomposes the problem into a hierarchy of subproblems.

A function definition is especially efficient when it is repeatedly called with different instantiations of its arguments. GP with ADFs have shown significant improvements over the standard GP for most of the considered test problems [8, 9].

An ADF in MEP has the same structure as a MEP chromosome (i.e. a string of genes). The ADF is also evolved in the same way as a standard MEP chromosome. The function symbols used by an ADF are the same

as those used by the standard MEP chromosomes. The terminal symbols used by an ADF are restricted to the function (ADF) parameters (formal parameters). For instance, if we define an ADF with two formal parameters p_0 and p_1 we may use only these two parameters as terminal symbols within the ADF structure, even if in the standard MEP chromosome (i.e. the main evolvable structure) we may use, let say, 20 terminal symbols only.

The set of function symbols of the main MEP structure is enriched with the Automatically Defined Functions considered in the system.

Example

Let us suppose that we want to evolve a program using 2 ADFs, denoted ADF_0 and ADF_1 having 2 (p_0 and p_1) respectively 3 (p_0 and p_1 and p_2) arguments. Let us also suppose that the terminal set for the main MEP chromosome is $T = \{a, b\}$ and the function set $F = \{+, -, *, /\}$. The terminal and function symbols that may appear in ADFs and main MEP chromosome are given in Table 8.

Table 8: Parameters, terminal set and the function set for the ADFs and for the main MEP chromosome

	Parameters	**Terminal set**	**Function set**
ADF_0	p_0, p_1	T={p_0, p_1}	F={+,–,*,/}
ADF_1	p_0, p_1, p_2	T={p_0, p_1, p_2}	F={+,–,*,/}
MEP chromosome	–	T={a, b}	F={+,–,*,/, ADF_0, ADF_1}

The ADF_0 could be defined as follows:

ADF_0 (p_0, p_1)
1: p_0
2: + 1, 1
3: p_1
4: / 3, 2
5: * 2, 4

The main MEP chromosome could be the following:

1: a
2: b
3: + 1, 2
4: ADF_0 3, 1
5: a
6: ADF_1 4, 5, 5
7: * 3, 6

The fitness of a MEP chromosome is computed as described in section 4.

The quality of an ADF is computed in a similar manner. The ADF is read once and the partial results are stored in an array (by the means of Dynamic Programming [4]). The best expression encoded in the ADF is chosen to represent the ADF.

The genetic operators (crossover and mutation) used in conjunction with the standard MEP chromosomes may be used for the ADFs too. The probabilities for applying genetic operators are the same for MEP chromosomes and for the Automatically Defined Functions. The crossover operator may be applied only between structures of the same type (that is ADFs having the same parameters or main MEP chromosomes) in order to preserve the chromosome consistency.

12 Applications

Multi Expression Programming has been applied, so far, to the following problems:

- Symbolic regression, classification [17]
- Data analysis for real world problems [17]
- Digital circuits design (adders, multipliers, even-parity) [19, 25, 26]
- Designing digital circuits for NP-Complete problems [21]
- Designing reversible digital circuits [27]
- Evolving play strategies for Nim game [23]
- Evolving evolutionary algorithms [18, 28]
- Evolving heuristics for TSP problems [20]
- Modeling of Electronic Hardware [1]

13 Online resources

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

- Multi Expression Programming, `https://mepx.org` or `https://mepx.github.io`

14 Tips for implementation

As MEP encodes multiple solutions within a chromosome it is important to set an optimal value for the chromosome length. The chromosome length directly affects the size of the search space and the number of solutions explored within this search space. Numerical experiments described in some of the next chapters show that it is a good idea to maintain chromosomes longer than the size of the actual solution for the problem being solved. If we set the chromosome length equal to the length of the (shortest) solution we will get a very low success rate. The success of MEP will increase (up to a point) as long as the length of the chromosome will be increased. However, the speed of the algorithm will decrease by increasing the length of the chromosome. Thus an optimal length of the chromosome must be a tradeoff between running time and the quality of the solutions.

15 Summary

Multi Expression Programming technique has been described in this chapter.

A distinct feature of MEP is its ability to encode multiple solutions in the same chromosome. It has been shown that the complexity of the decoding process is the same as in the case of other GP techniques encoding a single solution in the same chromosome. Several numerical examples have been presented for the MEP decoding process.

A set of genetic operators (crossover and mutation) has been described in the context of Multi Expression Programming.

Problems

- Improve the speed of MEP by using Sub Machine Code GP [30, 31].
- Propose new techniques for reducing the duplicate expressions in MEP. This will reduce the size of the search space and will help MEP to perform better when solving difficult problems.

- Implement and compare different mechanisms for handling exceptions in MEP. The current one (described in section 8) mutates the gene that has generated the exception. Compare this mechanism with that used by LGP [5] or standard GP [8]. Compare the running time and the evolution of the number of exceptions in these cases.

- Implement a MEP variant in which the chromosomes have variable length. Also propose some genetic operators that can deal with variable length MEP chromosomes. In this case the algorithm must be able to evolve the optimal length for the MEP chromosome. There are some difficulties related to this approach since MEP chromosomes will tend to increase their length in order to accommodate more and more solutions. And this could lead to bloat

- In our implementation all symbols (terminals and functions) have the same probability to appear into a chromosome. There could be some problems in this case. For instance, if our problem has many variables (let's say 100) and the function set has only 4 symbols we cannot get too complex trees because the function symbols have a reduced chance to appear in the chromosome. Change the implementation (the procedures for initialization and mutation) so that the set of terminals and the set of functions have the same probability of being chosen in the chromosome. In this case one must choose first the set (either terminals or functions) and then one will randomly choose a symbol from that set. Compare those two implementation for different test problems.

- A variant of MEP is to keep all terminals in the first positions. No other genes containing terminals are allowed in the chromosome. Try to compare the standard variant with this one for some benchmark problems.

References

[1] Abraham A, Groşan C, Genetic Programming Approach for Fault Modeling of Electronic Hardware, in Proceedings of the Congress on Evolutionary Computation, CEC'05, 2005.

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

[3] Banzhaf, W., Nordin, P., Keller, E. R., Francone, F. D., Genetic Programming - An Introduction, Morgan Kaufmann, San Francisco, CA, 1998.

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

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

[6] Ferreira C (2001) Gene Expression Programming: a New Adaptive Algorithm for Solving Problems, Complex Systems, Vol. 13, Nr. 1, pp. 87-129

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

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

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

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

[11] Langdon, W. B., Poli, R., 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, pp. 96-112, Springer-Verlag, Berlin, 1998.

[12] Lones, M.A., Tyrrell, A.M., Biomimetic Representation in Genetic Programming, Genetic Programming and Evolvable Machines, Vol. 3, Issue 2, pp. 193-217, 2002

[13] Miller, J.F., Thomson, P., Cartesian Genetic Programming. In Proceedings of the 3rd International Conference on Genetic Programming (EuroGP2000), R. Poli, J.F. Miller, W. Banzhaf, W.B. Langdon, J.F. Miller, P. Nordin, T.C. Fogarty (Editors), LNCS 1802, Springer-Verlag, Berlin, pp. 15-17, 2000.

[14] Miller, J. F., Job, D., Vassilev, V.K., Principles in the Evolutionary Design of Digital Circuits - Part I, Genetic Programming and Evolvable Machines, Vol. 1(1), pp. 7–35, Kluwer Academic Publishers, 2000.

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

[16] Nordin P, Banzhaf W, Brameier M (1998), Evolution of a World Model for a Miniature Robot using Genetic Programming, Robotics and Autonomous Systems, Vol. 25 pp. 105 - 116

[17] Oltean M, Groşan C, A Comparison of Several Linear Genetic Programming Techniques, Complex-Systems, Vol. 14, Nr. 4, pp. 282-311, 2003.

[18] Oltean M, Groşan C, Evolving Evolutionary Algorithms using Multi Expression Programming, The 7th European Conference on Artificial Life, Dortmund, 14-17 September, Banzhaf W, (editor), LNAI 2801, pp. 651-658, Springer-Verlag, Berlin, 2003.

[19] Oltean M, Solving Even-parity problems with Multi Expression Programming, Te 8th International Conference on Computation Sciences, 26-30 September 2003, North Carolina, K. Chen (editor), pp. 315-318, 2003.

[20] Oltean M, Dumitrescu D (2004) Evolving TSP Heuristics using Multi Expression Programming, International Conference on Computational Sciences, ICCS'04, M. Bubak, G. D. van Albada, P. Sloot, and J. Dongarra (editors), Vol II, pp. 670-673, 6-9 June, Krakow, Poland, 2004.

[21] Oltean, M., Groşan, C., Oltean, M., Evolving Digital Circuits for the Knapsack Problem, International Conference on Computational Sci-

ences, M. Bubak, G. D. van Albada, P. Sloot, and J. Dongarra (editors), Vol. III, pp. 1257-1264, 6-9 June, Krakow, Poland, 2004.

[22] Oltean, M., Groşan, C., Oltean, M., Encoding Multiple Solutions in a Linear GP Chromosome, International Conference on Computational Sciences, M. Bubak, G. D. van Albada, P. Sloot, and J. Dongarra (editors), Vol III, pp. 1281-1288, 6-9 June, Krakow, Poland, 2004.

[23] Oltean, M., Evolving Winning Strategies for Nim-like Games, World Computer Congress, Student Forum, 26-29 August, Toulouse, France, edited by Mohamed Kaaniche, pp. 353-364, Kluwer Academic Publisher, 2004.

[24] Oltean M., Improving the Search by Encoding Multiple Solutions in a Chromosome, contributed chapter 15, Evolutionary Machine Design, Nova Science Publisher, New-York, N. Nedjah, L. de Mourrelo (editors), 2004.

[25] Oltean M., Grosan C., Evolving Digital Circuits using Multi Expression Programming, NASA/DoD Conference on Evolvable Hardware, 24-25 June, Seatle, R. Zebulum, D. Gwaltney, G. Horbny, D. Keymeulen, J. Lohn, A. Stoica (editors), pages 87-90, IEEE Press, NJ,2004.

[26] Oltean M., Improving Multi Expression Programming: an Ascending Trail from Sea-Level Even-3-Parity to Alpine Even-18-Parity, chapter 10, Evolutionary Machine Design: Theory and Applications, Springer-Verlag, N. Nedjah and L. de Mourrelo (editors), pp. 270-306, 2004.

[27] Oltean M., Evolving reversible circuits for the even-parity problem, EvoHOT workshop, Rothlauf, F., Branke, J., Cagnoni, S., Corne, D.W., Drechsler, R., Jin, Y., Machado, P., Marchiori, E., Romero, J., Smith, G.D., Squillero, G. (Editors), LNCS 3449, Springer-Verlag, Berlin, 2005.

[28] Oltean, M., Evolving Evolutionary Algorithms with Patterns, Soft Computing, Springer-Verlag, 2006 (accepted)

[29] O'Neill M, Ryan C, Crossover in Grammatical Evolution: A Smooth Operator? Genetic Programming, Proceedings of EuroGP'2000, 2000, Poli R (et al.) (Editors), LNCS 1802, pp. 149-162, Springer-Verlag, 2000.

[30] Poli R, Langdon W B, Sub-machine Code Genetic Programming, in Advances in Genetic Programming 3, L. Spector, W. B. Langdon, U.-M. O'Reilly, P. J. Angeline, Eds. Cambridge:MA, MIT Press, 1999, chapter 13.

[31] Poli R, Page J, Solving High-Order Boolean Parity Problems with Smooth Uniform Crossover, Sub-Machine Code GP and Demes, Journal of Genetic Programming and Evolvable Machines, Kluwer, pp. 1-21, 2000.

[32] Prechelt, L., PROBEN1 – A Set of Neural Network Problems and Benchmarking Rules, Technical Report 21, University of Karlsruhe, 1994.

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

[34] Wolpert, D.H., McReady, W.G., No Free Lunch Theorems for Search, technical report SFI-TR-05-010, Santa Fe Institute, 1995.

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