Evolving Evolutionary Algorithms with Patterns

Mihai Oltean
Department of Computer Science
Faculty of Mathematics and Computer Science
Babeş-Bolyai University, Kogălniceanu 1
Cluj-Napoca, 3400, Romania
Email: moltean@cs.ubbcluj.ro

Abstract

A new model for evolving Evolutionary Algorithms (EAs) is proposed in this paper. The model is based on the Multi Expression Programming (MEP) technique. Each MEP chromosome encodes an evolutionary pattern which is repeatedly used for generating the individuals of a new generation. The evolved pattern is embedded into a standard evolutionary scheme which is used for solving a particular problem. Several evolutionary algorithms for function optimization are evolved by using the considered model. The evolved evolutionary algorithms are compared with a human-designed Genetic Algorithm. Numerical experiments show that the evolved evolutionary algorithms can compete with standard approaches for several well-known benchmarking problems.

Keywords

Genetic Algorithms, Genetic Programming, Evolving Evolutionary Algorithms, Multi Expression Programming, Linear Genetic Programming.

I. INTRODUCTION

Solving problems defines the human history. Unfortunately, each problem requires its own method for solving it. Modern techniques and algorithms tried to overcome this drawback. For instance, Evolutionary Computation [16], [17] tries to shift the difficulty from algorithms to representation: the algorithm is intend to be a general one, whereas the solution representation and associated genetic operators should be problem dependent.

The questions related to the efficiency of a particular evolutionary scheme have led the birth of No Free Lunch theorems for Search and Optimization [37], [38]. The answer was negative: we cannot have the best Evolutionary Algorithm (EA) which could perfectly solve all the optimization problems. All we can do is designing optimal or near-optimal algorithms for some particular problems without expecting any guaranty related to their generalization ability on new and unseen test problems.

Many evolutionary algorithms have been proposed for dealing with optimization problems. However, the lack of mathematical proofs of convergence does exist in most of the cases. Only some experimental evidence has been employed for supporting the proposed theories.

Instead of involving sophisticated mathematical theories we propose a simple way of deriving good evolutionary algorithms for a particular problem or set of problems: by evolving them. Our purpose is to evolve an evolutionary algorithm capable of solving a particular optimization problem. Thus, we will work with EAs at two levels: the first (macro) level consists in a steady-state EA [34] which uses a fixed population size, a fixed mutation probability, a fixed crossover probability etc. The second (micro) level consists in the solutions encoded in a chromosome of the first level EA.

The macro level evolves a population of evolutionary algorithms in the micro level. The evolutionary algorithms in the micro level evolve solutions for a particular optimization problem (such as function optimization, TSP, etc.) Thus the output of the micro level EAs are solutions for a particular problem being solved, whereas the output of macro level EA is the best evolutionary algorithm in the micro level.

For the first (macro) level EA we use a standard evolutionary model, similar to Multi Expression Programming (MEP) [24], [25], [26], which is very suitable for evolving computer programs that may be easily translated into an imperative language (like *C* or *Pascal*). The basic idea of evolving evolutionary algorithms is depicted in Figure 1.

Initial population of Evolutionary Algorithms

Mutation	Selection	Selection	Crossover
Crossover	Mutation	Crossover	Mutation
Crossover	Crossover	Crossover	Crossover
Mutation	Mutation	Mutation	Crossover
Crossover	Crossover	Selection	Crossover
Selection	Selection	Selection	Mutation
Mutation	Crossover	Crossover	Mutation

Fitness

Apply each Evolutionary
Algorithm to the problem
being solved

Good solution found?

yes → **STOP**

no

Selection

Select individuals using
a standard selection
procedure

Crossover

Mutation	Selection	Mutation	Selection
Crossover	Mutation	Crossover	Mutation
Crossover	Crossover	Crossover	Crossover
Mutation	Mutation	Mutation	Mutation
Crossover	Crossover	Crossover	Crossover
Selection	Selection	Selection	Selection
Mutation	Crossover	Crossover	Mutation

X

Mutation

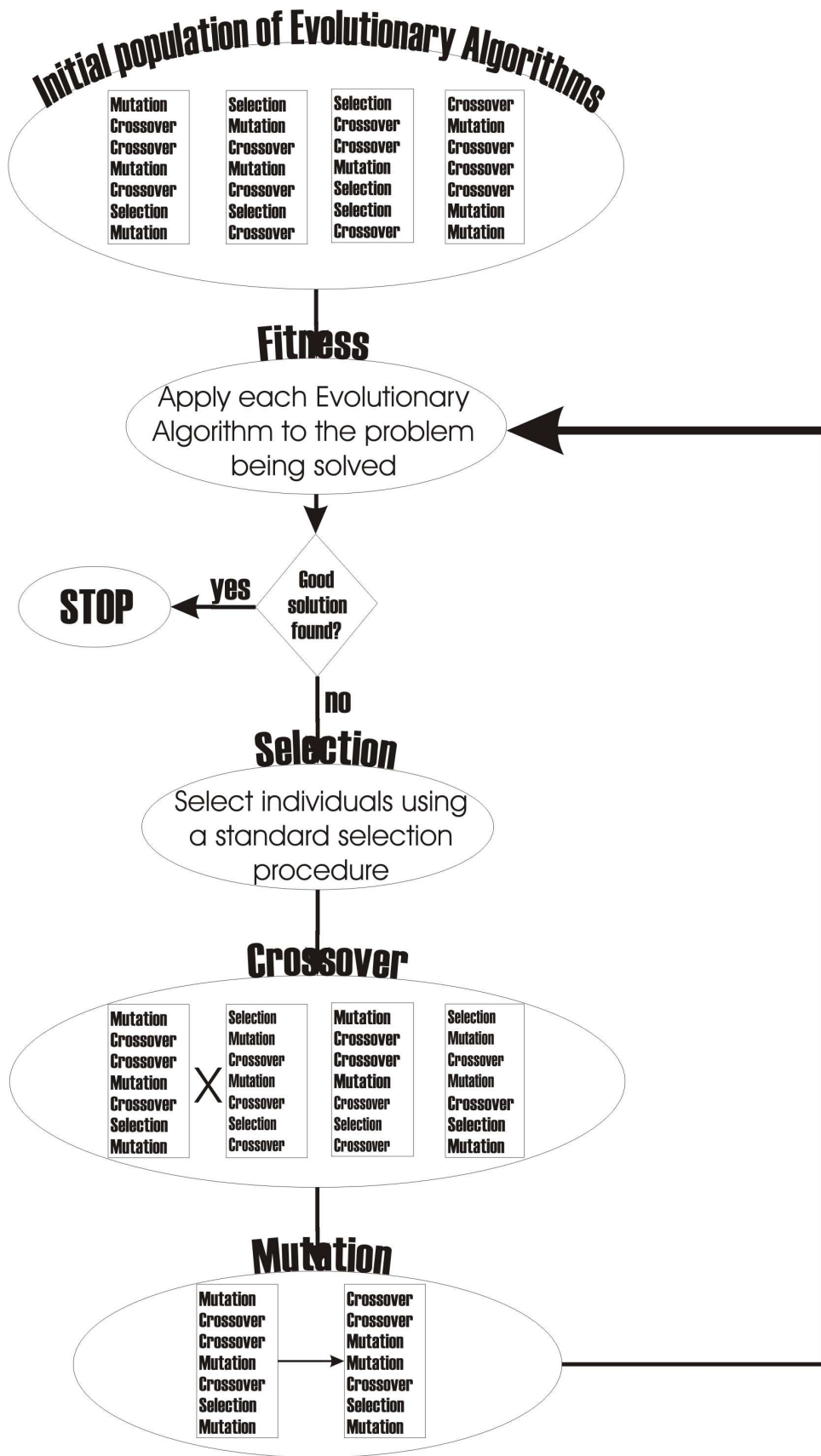|	
Mutation	Crossover
Crossover	Crossover
Crossover	Mutation
Mutation	Mutation
Crossover	Crossover
Selection	Selection
Mutation	Mutation

Fig. 1.   A schematic view of the process of evolving evolutionary algorithms. Macro level EA evolves evolutionary algorithms within micro level. The output of the macro-level EA is the best EA within micro-level.

Instead of evolving an entire EA we evolve only the heart of the algorithm, which is the sequence of instructions that generates a new offspring by taking information from the current population. The size of the search space is considerable smaller in this case. Numerical experiments performed with the proposed model reveal a short training time for obtaining a good evolutionary algorithm capable of competing a standard genetic algorithm for some particular test problems.

Note that the rules employed by the evolved EAs are not preprogrammed. These rules are automatically discovered by the evolution.

This research was motivated by the need of answering several important questions concerning evolutionary algorithms:

What is the optimal structure of an evolutionary algorithm?

Which are the genetic operators that have to be used in conjunction with an EA (for a given problem)?

Which is the optimal (or near-optimal) sequence of genetic operations (selections, crossovers and mutations) to be performed during a generation of an EA for a particular problem?

For instance, in a standard genetic algorithm the sequence is the following: selection, recombination and mutation. But, how do we know that this scheme is the best for a particular problem (or problem instance)? We had better let the evolution to find the answer for us. Note that evolving only the probabilities for applying specific genetic operators is not enough. Also, the structure of an EA and the order in which these genetic operators are applied is also important.

The paper is organized as follows. An overview of the related work in the field of evolving evolutionary algorithms is made in section II. The new model proposed for evolving EAs is presented in section III. Several numerical experiments are performed in section IV. These experiments proved that the model proposed in this paper is able to evolve good evolutionary algorithm capable of competing with human-written algorithms. Further research directions are suggested in section V.

II. RELATED WORK

Several attempts for evolving Evolutionary Algorithms using similar techniques were made in the past. A non-generational EA was evolved [25] by using the Multi Expression Programming (MEP) technique [25], [26]. A generational EA was evolved [27], [29] by using the Linear Genetic Programming (LGP) technique [7], [23]. Numerical experiments have shown [25], [27] that the evolved EAs perform similarly and sometimes even better than the standard evolutionary approaches with which they are compared.

There are also several approaches that evolve genetic operators for solving difficult problems [2], [3], [12], [33], [35]. In his paper on Meta-Genetic Programming, Edmonds [12] used two populations: a standard GP population and a co-evolved population of operators that act on the main population. Note that all these approaches use a fixed evolutionary algorithm which is not changed during the search.

Attempts for evolving complex heuristics for particular problems were made in the recent past. In [28] the authors evolve an heuristic for the Traveling Salesman Problem. The obtained heuristic is a mathematical expression that takes as input some information about the already constructed path and outputs the next node of the path. It was shown [28] that the evolved heuristic performs better than other well-known heuristics (Nearest Neighbor Heuristic, Minimum Spanning Tree Heuristic [11], [15]) for the considered test problems.

A recent paper of Spector and Robinson [32] describes a language called Push which supports a new, self-adaptive form of evolutionary computation called autoconstructive evolution. An experiment for symbolic regression problems was reported but no comparison with a standard evolutionary algorithm (like GP) was provided. The conclusion was that *Under most conditions the population quickly achieves reproductive competence and soon thereafter improves in fitness.* [32].

A. *Multi Expression Programming based Approach*

Multi Expression Programming (MEP) [25], [26], [24] uses a special representation that is similar to the way in which C and Pascal compilers translate mathematical expressions into machine code [1]. The MEP genes are (represented by) substrings of variable length. The number of genes in a chromosome is constant and it represents the chromosome length. Each gene encodes a terminal (an element in the terminal set T) or a function symbol (an element in the function set F). A gene encoding a function includes pointers towards the function arguments. Function parameters always have indices of lower values than the position of that function itself in the chromosome.

According to the proposed representation scheme, the first symbol in a chromosome must be a terminal symbol. In this way only syntactically correct programs are obtained.

Example

We use a representation where the numbers on the left stand for gene labels (or memory addresses). Labels do not belong to the chromosome, they are provided only for explanatory purposes. An example of a chromosome is given below (assuming that $T = \{a, b, c, d\}$ and $F = \{+, -, *, /\}$):

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

The MEP chromosomes are read top-down, starting with the first position. A terminal symbol specifies a simple expression. A function symbol specifies a complex expression (formed by linking the operands specified by the argument positions with the current function symbol).

The expressions encoded in this chromosome are:

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

We have to choose one of these expressions ($E_1, \ldots, E_7$) to represent the chromosome. There is neither theoretical nor practical evidence that one of them is better than the others. Thus, we choose to encode multiple solutions in a single chromosome. Each MEP chromosome encodes a number of expressions equal to the chromosome length (the number of genes). The expression associated to each chromosome position is obtained by reading the chromosome bottom-up from the current position, by following the links provided by the function pointers. The fitness of each expression encoded in a MEP chromosome is computed in a conventional manner (the fitness depends on the problem being solved). The best expression encoded in an MEP chromosome is chosen to represent the chromosome (the fitness of an MEP individual equals the fitness of the best expression encoded in that chromosome).

Genetic operators used in conjunction with MEP are crossover and mutation (see [24] for more information).

1) Evolving full EAs with MEP: In order to use MEP for evolving EAs the set of terminal symbols and a set of function symbols have been redefined [25]. We have to keep in mind that the value stored in a terminal symbol is independent of other symbols in the chromosome and a function symbol changes the solution stored in another gene.

We have decided to use 4 types of genetic operators in our evolved EA. The evolution will decide which operators are good and which are bad for our problem. The employed operators along with their meaning are given below:

- *Initialization* - randomly initializes a solution,
- *Selection* - selects the best solution among several already existing solutions
- *Crossover* - recombines two already existing solutions,
- *Mutation* - varies an already existing solution.

These operators will act as symbols that may appear into an MEP chromosome. The only operator that generates a solution independent of the already existing solutions is the *Initialization* operator. This operator will constitute the terminal set. The other operators will be considered function symbols. Thus, we have $T = \{Initialization\}$, $F = \{Selection, Crossover, Mutation\}$.

An MEP chromosome C, storing an evolutionary algorithm, can look like:

```
1:  Initialization     {Randomly generates a solution}
2:  Initialization     {Randomly generates another solution}
3:  Mutation 1         {Mutates the solution stored on position 1}
4:  Selection 1, 3     {Selects the best solution from those}
                       {stored on positions 1 and 3}
5:  Crossover 2, 4     {Recombines the solutions on positions 2 and 4}
6:  Mutation 4         {Mutates the solution stored on position 4}
7:  Mutation 5         {Mutates the solution stored on position 5}
8:  Crossover 2, 6     {Recombines the solutions on positions 2 and 6}
```

This MEP chromosome encodes multiple evolutionary algorithms (in fact 8 EAs as many). They are given in Table I. Each EA is obtained by reading the chromosome bottom-up, starting with the current gene and following the links provided by the function pointers. The best evolutionary algorithm encoded into a chromosome will represent that chromosome (will provide the fitness of that chromosome).

The complexity of the EAs encoded into a MEP chromosome varies from very simple (EAs made up of a single instruction) to very complex (sometimes using all genes of the MEP chromosome). This is very useful because we do not know, in advance, the complexity of the EA required to solve a problem. The required algorithm could be very simple (and, in this case, the simplest individuals encoded by MEP are very useful) or it could be very complex (and, in this case, the most complex EAs are taken into account).

Thus we deal with EAs at two different levels: a micro level representing the evolutionary algorithm encoded in an MEP chromosome and a macro level GA, which evolves MEP individuals. The number of genetic operators (initializations, crossovers, mutations, selections) is not fixed and it may vary between 1 and the MEP chromosome length. These values are automatically discovered by the evolution. The macro level GA execution is bound by the known rules for GAs ([16]). This process is depicted in Figure 1.

a) Remarks:

(i) In our model, the *Crossover* operator always generates a single offspring from two parents. The crossover operators generating two offspring may also be designed to fit our evolutionary model.

(ii) The *Selection* operator acts as a binary tournament selection. The better out of two individuals is always accepted as the selection result.

TABLE I

EVOLUTIONARY ALGORITHMS ENCODED IN THE MEP CHROMOSOME C

EA_1	EA_2	EA_3	EA_4
i_1=Initialization	i_1=Initialization	i_1=Initialization i_2=Mutation(i_1)	i_1 = Initialization i_2=Mutation(i_1) i_3=Selection(i_1,i_2)
EA_5	EA_6	EA_7	EA_8
i_1=Initialization i_2=Initialization i_3=Mutation(i_1) i_4=Selection(i_1,i_3) i_5=Crossover(i_1,i_4)	i_1=Initialization i_2=Mutation(i_1) i_3=Selection(i_1,i_2) i_4=Mutation(i_3)	i_1=Initialization i_2=Initialization i_3=Mutation(i_1) i_4=Selection(i_1,i_3) i_5=Crossover(i_2,i_4) i_6=Mutation(i_5)	i_1=Initialization i_2=Initialization i_3=Mutation(i_1) i_4=Selection(i_1,i_3) i_5=Mutation(i_4) i_6=Crossover(i_2,i_5)

(iii) The *Initialization, Crossover* and *Mutation* operators are problem dependent.

2) Fitness assignment: We have to compute the quality of each EA encoded in the chromosome in order to establish the fitness of an MEP individual. For this purpose each EA encoded in an MEP chromosome is run on the particular problem being solved.

Roughly speaking the fitness of an MEP individual is equal to the fitness of the best solution generated by one of the evolutionary algorithms encoded in that MEP chromosome. But, since the EAs encoded in an MEP chromosome use pseudo-random numbers it is likely that successive runs of the same EA generate completely different solutions. This stability problem is handled in the following manner: each EA encoded in an MEP chromosome was executed (run) more times and the fitness of an MEP chromosome is the average of the fitness of the best EA encoded in that chromosome over all runs. In all the experiments performed in [25] each EA encoded into an MEP chromosome was run 200 times.

B. Linear Genetic Programming based Approach

Linear Genetic Programming (LGP) [5], [7], [23] uses a specific linear representation of computer programs. Programs of an imperative language (like *C*) are evolved instead of the tree-based GP expressions of a functional programming language (like *LISP*).

An 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 an LGP program is the following:

```
void LGP_Program(double v[8])
{
...
   v[0] = v[5] + 73;
   v[7] = v[4] − 59;
   v[4] = v[2] * v[1];
   v[2] = v[5] + v[4];
   v[6] = v[1] * 25;
   v[6] = v[4] − 4;
   v[1] = sin(v[6]);
   v[3] = v[5] * v[5];
   v[7] = v[6] * 2;
   v[5] = [7] + 115;
   v[1] = sin(v[7]);
}
```

1) LGP for Evolving EAs: The structure of an LGP chromosome has been adapted for evolving EAs [27]. Instead of working with registers, our LGP program will modify an array of individuals (the population). We denote by *Pop* the array of individuals (the population) which will be modified by an LGP program.

The set of function symbols will consist in genetic operators that may appear into an evolutionary algorithm. There are usually 3 types of genetic operators that may appear into an EA (see section II-A): *Crossover, Mutation and Selection.* The *Initialization* operator (see section II-A) is used for initialization of the population and it is not evolved like all other operators.

The LGP statements are considered to be genetic operations executed during an EA generation. Since the purpose was to evolve a generational EA a wrapper loop has been added around the genetic operations that are executed during an EA generation. Even more, each EA starts with a random population of individuals. Thus, the LGP program must contain some instructions that initialize the initial population.

An LGP chromosome encoding an EA is given below:

```
void LGP_Program(Chromosome Pop[8]) // a population with of 8 individuals
{
    Randomly_initialize_the_population();
    // repeat for a number of generations
    for (int k = 0; k < MaxGenerations; k++){
        Pop[0] = Mutation(Pop[5]);
        Pop[7] = Selection(Pop[3], Pop[6]);
        Pop[4] = Mutation(Pop[2]);
        Pop[2] = Crossover(Pop[0], Pop[2]);
        Pop[6] = Mutation(Pop[1]);
        Pop[2] = Selection(Pop[4], Pop[3]);
        Pop[1] = Mutation(Pop[6]);
        Pop[3] = Crossover(Pop[5], Pop[1]);
    }
}
```

Remarks

(i) The initialization function and the **for** cycle will not be affected by the genetic operators. These parts are kept unchanged during the search process.

(ii) Each LGP chromosome encodes a single EA. This is different from the MEP approach where each chromosome encodes multiple EAs.

The quality of an EA encoded in an LGP chromosome is computed as in the case of the MEP approach (see section II-A.2). The EA encoded in an LGP chromosome is run on the particular problem being solved and the quality of the EA is equal to the quality of the best solution evolved by that EA.

III. PROPOSED MODEL

The model proposed for evolving evolutionary algorithms is described in this section.

A. Motivation

This section tries to answer an important question:

"Why evolving EAs with patterns?"

In the previously described models (MEP-based and LGP-based), the search space of the evolutionary algorithms was huge. The time needed to train a human-competitive evolutionary algorithm could take between several hours and several days.

Instead of evolving an entire EA we will try to evolve a small piece of code that will repeatedly be used in order to obtain new individuals. Most of the known evolutionary schemes use this form of evolution. For instance, in a standard GA, the following piece of code is successively applied until the new population is filled:

$p_1 = Selection$ (); // randomly choose two individuals from the current population and
                      //the best of them will be the result of the selection
$p_2 = Selection$ (); // choose another two randomly individuals and return the best of them in p_2
$c = Crossover(p_1, p_2)$;
$c = Mutation(c)$;
$Fitness(c)$; // compute the fitness of the individual c
$Copy$ c in the next generation;

The patern employed by a simple (1+1) Evolution Strategy [6] is:

$b = Mutation(a)$;
$Fitness(b)$ // compute the fitness of the individual b
if b is better than a
then $Replace$ a with b;
endif

The patern employed by a Steady-State Evolutionary Algorithm [34] is:

$p_1 = Selection$ (); // randomly choose two individuals from the current population and
                      //the best of them will be the result of the selection
$p_2 = Selection$ (); // choose another two randomly individuals and return the best of them in p_2
$c = Crossover(p_1, p_2)$;
$c = Mutation(c)$;
$Fitness(c)$; // compute the fitness of the individual c
if c is better than the worst individual in the population
then $Replace$ the worst individual in the population with c;
endif

The main advantage of this approach is its reduced complexity: the size of the pattern is considerably smaller than the size of the entire EA as evolved in [25], [27].

The patterns in an evolutionary algorithm can be assimilated with the Automatically Defined Functions (ADFs) in Genetic Programming [19].

B. Representation

As shown in the previous section, the use of patterns greatly reduces the size of the search space.

Our pattern will be represented as an MEP computer program whose instructions will be executed during the EA evolution. We have chosen Multi Expression Programming for representing the patterns because MEP provides an easy way to store a sequence of instructions. Note that, in our approach, MEP will store only one solution (pattern) per chromosome because, due to the generational algorithm where the pattern will be embedded, is difficult to handle multiple evolutionary patterns in the same chromosome. We will still use the MEP notation but the there will be only one solution per chromosome in all cases and experiments below.

Other GP methods (such as standard Genetic Programming [18] and Gene Expression Programming [13] may also be used for storing patterns. Linear Genetic Programming [7] is less suitable for this model because we are not very interested in storing (by using some destination variables) the results of the crossover, selection and mutation operators. However, in our implementation we use some destination variables, but these variables are fixed and they are not subject to evolution (as in the LGP model).

As previously described in section II-A, we have decided to use 4 types of genetic operation within an EA:

- *Initialization* - randomly initializes a solution,
- *Selection* - selects the best solution among several already existing solutions,
- *Crossover* - recombines two already existing solutions,
- *Mutation* - varies an already existing solution.

Since our purpose is to evolve a small piece of code that will be used in order to generate a new population based on the old population we will not use the *Initialization* operator. This operator is used once at the moment of population initialization. The other three operators *Selection, Crossover* and *Mutation* will be used within the evolved pattern.

The operators that appear in the evolved EA and their new meanings are shown below:

- *Selection* - selects a solution from the old population. This operation is implemented as a binary tournament selection: two individuals are randomly chosen and the best of them is the result of selection.
- *Crossover (a, b)* - recombines solutions a and b.
- *Mutation (a)* - varies solution a.

Function set is $F = \{Crossover, Mutation\}$ and the terminal set is $T = \{Selection\}$. As a restriction imposed by the MEP representation, the first instruction in a chromosome must be a $Selection$.

Let us recall the MEP representation as given in section II-A. A MEP chromosome can have the form (we took only the first 5 genes of the chromosome given in section II-A):

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

We will replace the terminal set $\{a, b, ...\}$ by the new terminal symbol $\{Selection\}$ which is specific to our purpose. Also the function set $\{+, *, -, ...\}$ will be replaced by $\{Crossover, Mutation\}$.

Example 1

An example of a MEP chromosome encoding a pattern is given below:

1: $Selection$
2: $Selection$
3: $Mutation$ 1
4: $Mutation$ 2
5: $Crossover$ 2, 4

This MEP chromosome should be interpreted as follows:

- An individual (let us denote it by a) is selected from the current population
- Another individual (let us denote it by b) is selected from the current population
- Individual a is mutated. The result of the mutation is a new individual denoted by c.
- Individual b is mutated. A new individual (denoted by d) is obtained.
- Individuals b and d are recombined using a problem-dependent crossover. A new individual e is obtained.

Example 2

The pattern employed by a standard GA rewritten as an MEP chromosome is given below:

1: *Selection*
2: *Selection*
3: *Crossover* 1, 2
4: *Mutation* 3

The obtained individual is added to the new population.

C. *What brings new the proposed approach*

There are several important differences between the MEP-based technique [25] presented in section II-A.1 and the one proposed in this paper. First of all, the complexity of the evolved algorithms is quite different. In the first approach [25] we have evolved an entire EA, which is a simple sequence of genetic instructions (no generations, no loops for filling a population). In the current approach we keep intact most of the GA parts: we have generations and we have a loop which fills the new population.

Another important modifications have been performed to the genetic operators employed by the evolved evolutionary algorithms. As a general modification all genetic operators manipulate only individuals from the current population. This is again different from the method proposed in [25] whose operators manipulate individuals taken from the set of all individuals created since the beginning of the search process.

Other particular modifications are described below:

- *Initialization*. In the MEP-based approach proposed in [25] we could use initialization operator anywhere in the evolved algorithm. Here, in the current approach, this operator has been removed from the set of possible operators that could appear into an evolved pattern. Initialization is used only at the beginning of the algorithm. However that part is not subject to evolution.
- *Selection*. In the MEP-based approach used in [25] we have used a selection operator which has 2 fixed parameters (e.g. *Selection* 5, 2). In the current approach we use a more general type of selection operator which has no parameter. This operator also selects two random individuals and outputs the best of them, but is not bound anymore to some fixed positions in the population.
- *Crossover* - no modifications.
- *Mutation* - no modifications.

D. *Decoding MEP individuals*

The individual generated by the last instruction of the MEP chromosome will be added to the new population. It can be easily seen that not all MEP instructions are effective. Thus, we have to see the sequence of instructions that has generated that individual. In order to extract all effective instructions we

proceed in a bottom-up manner. We begin by marking the last instruction in the chromosome and then we follow the function pointers recursively until we get some terminals. The marked MEP genes are the only useful genetic instructions for our purpose.

For instance in the example 1 only the instructions in positions 2, 4 and 5 are actually utilized. Thus the pattern will be reduced to 3 instructions only:

    1: *Selection*
    2: *Mutation* 1
    3: *Crossover* 1, 2

The other instructions (introns) have been removed from this final pattern and the function pointers have been relabeled in order to reflect changes. Note that the introns removal is performed only for the best chromosome (in the population) at the end of the search process.

Remark.

Other operators could be added to the function and terminal sets in order to define a more complex behavior for the evolved pattern. For instance we could add an operator that computes the worst individual in the current population. This operator is very useful in a steady-state evolutionary model [34].

E. Where the pattern is embedded?

The evolved pattern is a sequence of instructions which will repeatedly be used for obtaining new individuals. This process is depicted in Figure 2. Note that the proposed approach tells us only the way in which a new individual is generated based on the existing individuals (the current population) and it does not tell us what to do with the obtained individual. In this model we have chosen to add the obtained individual to the new population. The new population will replace the current population at the end of a generation and the search process continues.

The algorithm where the pattern will be embedded is a standard one: it starts with a random population of solutions; always copies the best-of-generation individual in the next generation and all the other individuals are obtained by applying the pattern.

The evolutionary algorithm embedding the pattern is given below:

The Evolutionary Algorithm embedding the Evolved Pattern

S_1. Randomly create the initial population P(0)
S_2. **for** t = 1 **to** *Max_Generations* **do**
S_3. $P'(t)$ = { The best individual in $P(t)$ };
       //The best individual in the current population is added to the new population
S_4.   **for** k = 2 **to** $|P(t)|$ **do**
S_5.   RUN THE PATTERN() // the evolved pattern is run here
               // an offspring *offspr* is obtained
S_6.   Add *offspf* to $P'(t)$; //add *offspr* in the new population
S_7.   **endfor**
S_8. $P(t+1)$ = $P'(t)$;
S_9. **endfor**

Note that this algorithm will be used (see section IV-A.2) for solving particular problems such as function optimization. After obtaining the pattern we will not be anymore interested in MEP and other techniques for evolving complex computer programs (such EAs). We will focus only in using the pattern for solving some particular problems.
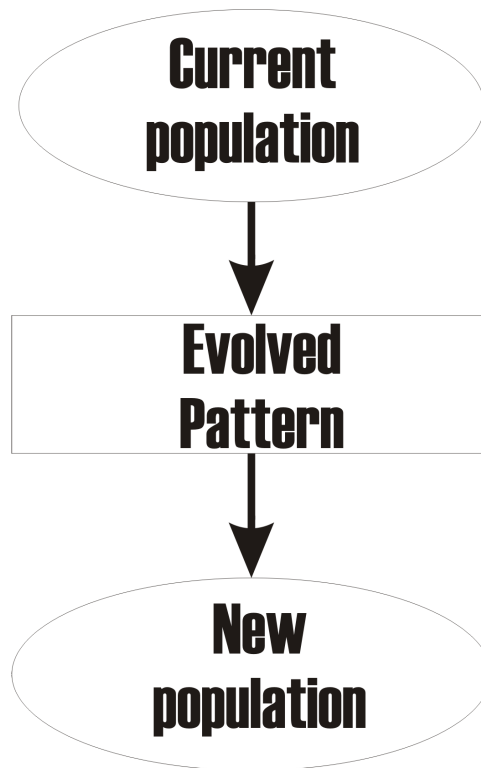
Fig. 2. The dataflow view of the process of obtaining new individuals using the evolved pattern. The input for the pattern is the current population. The output (the individual obtained by applying the pattern) is always added to the new population.

F. Fitness Assignment

We deal with EAs at two different levels: a micro level representing the pattern encoded into an MEP chromosome and a macro level GA, which evolves MEP individuals. Macro level GA execution is bounded by known rules for GAs (see [16]).

In order to compute the fitness of an MEP individual we have to compute the quality of the pattern encoded in that chromosome. For this purpose, the pattern encoded into a MEP chromosome is embedded into an evolutionary algorithm which is run on the particular problem being solved (as shown in the previous section). A solution (for the particular problem being solved) is obtained. For instance, this solution could be a real-valued array (in the case of function optimization problems) or could be a path in a graph in the case of the TSP problem.

Roughly speaking, the fitness of an MEP individual equals the fitness of the solution generated by the pattern encoded into that MEP chromosome. But, since the pattern encoded into an MEP chromosome uses pseudo-random numbers, it is very likely that successive runs of the same EA will generate completely different solutions. This stability problem is handled in a standard manner: the EA embedding the pattern encoded into an MEP chromosome is executed (run) more times (100 runs are, in fact, performed in all the experiments for evolving EAs for function optimization) and the fitness of an MEP chromosome is the average of the fitness of the EA encoded in that chromosome over all runs.

The optimization type (minimization/maximization) of the macro level EA is the same as the optimization type of the micro level EA. In our experiments we have employed a minimization relation (finding the minimum of a function).

G. The Model used for Evolving EAs

We use the steady state algorithm described in section III-H in order to evolve EAs. For increasing the generalization ability (i.e. the ability of the evolved EA to yield good results on new test problems), the

problem set is divided into three sets, suggestively called training set, validation set and test set (see [30]). In our experiments the training set consists in a difficult test problem. Validation is performed by using another difficult test problem. The test set consists in some other well-known benchmarking problems.

A method called *early stopping* [30] is used in order to avoid the overfitting of the population individuals to the particular training examples used. This method consists in computing the test set performance for the chromosome which had the minimum validation error during the search process. The use of early stopping technique will increase the generalization performance [30].

The test set consists in several well-known benchmarking problems [10], [31], [36] used for assessing the performances of the evolutionary algorithms.

H. The Algorithm for Evolving EAs

For evolving MEP individuals we use a steady-state [34] evolutionary algorithm. The sketch of this algorithm has been depicted in Figure 1.

The steady-state MEP algorithm starts with a randomly chosen population of individuals. The following steps are repeated until a termination condition is reached: Two parents are selected by using binary tournament and then they are recombined with a fixed crossover probability. Two offspring are obtained by the recombination of two parents. The offspring are mutated and the better of them replaces the worst individual in the current population (if the offspring is better than the worst individual in the current population).

The output of the MEP algorithm is the best evolutionary pattern encoded into a MEP chromosome. This pattern (sequence of intructions) will be embedded in the algorithm described in section III-E. The obtained algorithm will be used in section IV-A.2 for solving several well-known optimization problems.

I. Complexity and running time

The complexity of the proposed method is bounded by the known rules for evolutionary algorithm. In the case of the algorithm where evolved pattern is embedded (see section III-E) then complexity can be simply described by the classic equation:

$$CA = O(PopSize * NumberOfGenerations * NumberOfGenes * C), \tag{1}$$

where $NumberOfGenes$ is the number of genes in the MEP pattern, C is the complexity induced by the representation (real-valued array, TSP path etc) and the other parameters are self explaining.

The complexity of the algorithm used for evolving patterns is given by the formula:

$$O(PopSize * NumberOfGenerations * CA). \tag{2}$$

The CA factor was introduced here because we need to compute the fitness of each newly evolved pattern, which actually means that we have to run the algorithm given in section III-E whose complexity is described by equation 1.

Note that the values of parameters $PopSize$ and $NumberOfGenerations$ may be different in formulas 1 and 2.

The process of evolving algorithms is a complex task which requires many computational resources. This is because we need to assess the performance of each evolved pattern by applying it to a particular problem (function optimization in our case). For evolving an evolutionary pattern (see section IV) we need about 1 hour. This is a significant improvement compared to the technique proposed in [29] which requires about 1 day to evolve an algorithm.

IV. NUMERICAL EXPERIMENTS

In this section, several numerical experiments for evolving EAs are performed. An evolutionary algorithm for function optimization is evolved. Several numerical experiments, with a standard Genetic Algorithm [16] for function optimization, are also performed in order to assess the performance of the evolved EA. Finally the results are compared.

A. Evolving EAs for Function Optimization

In this section, an Evolutionary Algorithm for function optimization is evolved.

1) Test Functions: Ten test problems $f_1 - f_{10}$ (given in Table II) are used in order to asses the performance of the evolved EA. Functions $f_1 - f_6$ are unimodal test function. Functions $f_7 - f_{10}$ are highly multimodal (the number of the local minima increases exponentially with the problem dimension [36]).

TABLE II

TEST FUNCTIONS USED IN OUR EXPERIMENTAL STUDY. THE PARAMETER n IS THE SPACE DIMENSION ($n = 5$ IN OUR NUMERICAL EXPERIMENTS) AND f_{min} IS THE MINIMUM VALUE OF THE FUNCTION.

Test function	Domain	f_{min}
$f_1(x) = \sum\limits_{i=1}^{n} (i \cdot x_i^2).$	$[-10, 10]^n$	0
$f_2(x) = \sum\limits_{i=1}^{n} x_i^2.$	$[-100, 100]^n$	0
$f_3(x) = \sum\limits_{i=1}^{n} |x_i| + \prod\limits_{i=1}^{n} |x_i|.$	$[-10, 10]^n$	0
$f_4(x) = \sum\limits_{i=1}^{n} \left(\sum\limits_{j=1}^{i} x_j \right)^2.$	$[-100, 100]^n$	0
$f_5(x) = \max_i \{x_i, 1 \le i \le n\}.$	$[-100, 100]^n$	0
$f_6(x) = \sum\limits_{i=1}^{n-1} 100 \cdot (x_{i+1} - x_i^2)^2 + (1 - x_i)^2.$	$[-30, 30]^n$	0
$f_7(x) = 10 \cdot n + \sum\limits_{i=1}^{n} (x_i^2 - 10 \cdot \cos(2 \cdot \pi \cdot x_i))$	$[-5, 5]^n$	0
$f_8(x) = -a \cdot e^{-b\sqrt{\frac{\sum\limits_{i=1}^{n} x_i^2}{n}}} - e^{\frac{\sum \cos(c \cdot x_i)}{n}} + a + e.$	$[-32, 32]^n$ $a = 20,\ b = 0.2,\ c = 2\pi.$	0
$f_9(x) = \frac{1}{4000} \cdot \sum\limits_{i=1}^{n} x_i^2 - \prod\limits_{i=1}^{n} \cos(\frac{x_i}{\sqrt{i}}) + 1.$	$[-500, 500]^n$	0
$f_{10}(x) = \sum\limits_{i=1}^{n} (-x_i \cdot \sin(\sqrt{|x_i|}))$	$[-500, 500]^n$	$-n* 418.98$

2) Experimental Results: In this section we evolve an EA for function optimization and then we asses its performance. A comparison with standard GA is performed further in this section.

For evolving an EA we use f_1 as the training problem and f_2 as the validation problem. The number of dimensions was set to 5 for all numerical experiments.

An important issue concerns the solutions evolved by the EAs encoded into an MEP chromosome and the specific genetic operators used for this purpose. The solutions evolved by the EA encoded into MEP chromosomes are represented by using real values [16]. Thus, each chromosome of the evolved EA is a fixed-length array of real values. By initialization, a point within the definition domain is randomly generated. Convex crossover with $\alpha = {}^1\!/_2$ and Gaussian mutation with $\sigma = 0.01$ are used [16].

A short description of real encoding and the corresponding genetic operators is given in Table III.

Experiment 1

TABLE III

A SHORT DESCRIPTION OF REAL ENCODING.

Function to be optimized	$f : [MinX, MaxX]^n \rightarrow \Re$
Individual representation	$x = (x_1, x_2, \ldots, x_n)$.
Convex Recombination with $\alpha = 0.5$	parent $1 - x = (x_1, x_2, \ldots, x_n)$.
	parent $2 - y = (y_1, y_2, \ldots, y_n)$.
	the offspring $- o = \left(\frac{x_1 + y_1}{2}, \frac{x_2 + y_2}{2}, \ldots, \frac{x_n + y_n}{2} \right)$.
Gaussian Mutation	the parent $- x = (x_1, x_2, \ldots, x_n)$.
	the offspring $- o = (x_1 + G(0,\sigma), x_2 + G(0,\sigma), \ldots,$
	$x_n + G(0,\sigma))$,
	where G is a function that generates real values with
	Gaussian distribution.

An Evolutionary Algorithm for function optimization is evolved in this experiment. The parameters of the MEP algorithm are given in Table IV.

TABLE IV

THE PARAMETERS OF THE MEP ALGORITHM USED FOR EVOLVING EVOLUTIONARY ALGORITHMS.

Parameter	Value
Population size	100
Code Length	15 instructions
Number of generations	100
Crossover probability	0.8
Crossover type	Uniform Crossover
Mutation	1 mutation per chromosome
Function set	$F = \{Crossover, Mutation\}$
Terminal set	$F = \{Selection\}$

The parameters of the evolved EA are given in Table V.

TABLE V

THE PARAMETERS OF THE EVOLVED EA FOR FUNCTION OPTIMIZATION.

Parameter	Value
Individual representation	fixed-length array of real values.
Population size	50
Number of generations	50
Crossover probability	1
Crossover type	Convex Crossover with $\alpha = 0.5$
Mutation	Gaussian mutation with $\sigma = 0.01$
Mutation probability	1
Selection	Binary Tournament

The results of this experiment are depicted in Figure 3.

The effectiveness of our approach can be seen in Figure 3. The MEP technique is able to evolve an EA for solving optimization problems. The quality of the evolved EA improves as the search process advances.

One complete run (100 generations) took about 1 hour. Note that one run in the case of the LGP approach [27] took one day.

One of the best evolved evolutionary patterns is given below in MEP notation. Note that we have removed the introns [7] and we have kept only the effective code. This is why the pattern is shorter than the MEP chromosome used in evolution (see Table IV).
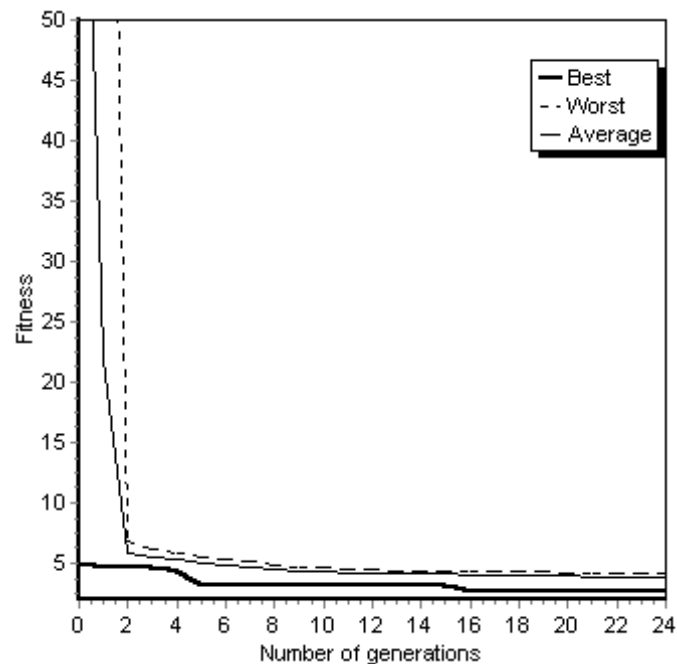
Fig. 3. The evolution of the fitness of the best individual, the worst individual and the average fitness in a particular run. We have depicted a window of 25 runs just in order to provide a better visualisation.

1: *Selection*
2: *Selection*
3: *Mutation* 2
4: *Mutation* 3
5: *Crossover* 1, 4
6: *Mutation* 5
7: *Mutation* 6
8: *Mutation* 7
9: *Mutation* 8

This chromosome selects two individuals (a and b), mutates twice the individual b and then performs a crossover between the result and a. Then the offspring is mutated four times. The structure of the chromosome suggests that the standard mutation (with $\sigma = 0.01$) is not sufficient for this test function. Additional mutations are performed in order to improve the algorithm's performance.

We have also analyzed the quality of other individuals involved in the search process. We have randomly chosen one of the runs and we have analyzed the fitness of the best and of the worst individual in the population (see Figure 3). The fitness of the best individual in the first generation is 4.92, while the average fitness (of all the individuals at generation 0) is 85.15. This means that we have some good individuals in the first generation, but we also have some very unfit individuals. The worst individual in the first generation has the fitness equal to 151.70. This individual is given below in MEP notation:

1: *Selection*
2: *Mutation* 1
3: *Selection*
4: *Crossover* 2, 3
5: *Selection*
6: *Crossover* 4, 5

7: *Selection*
8: *Crossover* 6, 7
9: *Mutation* 8

In the second generation the average fitness is 22.01 and the fitness of the worst individual is 121.02. In the third generation the worst individual in the population has the fitness 6.83, whereas the fitness of the best individual is 4.06. We see that in very few generations the average fitness quickly decreases, getting close to the value of the best fitness in the population. This means that the very unfit individuals are quickly replaced by some very fit individuals.

Experiment 2

This experiment serves our purpose of comparing the evolved EA with a standard Genetic Algorithm. The parameters used by the evolutionary algorithm embedding the evolved pattern are given in Table V. The parameters used by standard GA are given in Table VI. The results of the comparison are given in Table VII.

The standard GA algorithm, used for comparison, is given below:

Standard GA algorithm

S_1. Randomly create the initial population $P(0)$
S_2. **for** t = 1 **to** *Max_Generations* **do**
S_3. $P'(t)$ = { The best individual in $P(t)$ };
S_4.   **for** k = 2 **to** $|P(t)|$ **do**
S_5. p_1 = *Selection*($P(t)$); // select an individual from the population
S_6. p_2 = *Selection*($P(t)$); // select the second individual
S_7.   *Crossover* (p_1, p_2, *offsp*); // crossover the parents p$_1$ and p$_2$
            // an offspring *offspr* is obtained
S_8.   *Mutation* (*offspr*); // mutate the offspring *offspr*
S_9.   Add *offspf* to $P'(t)$; //move *offspr* in the new population
S_{10}.   **endfor**
S_{11}. $P(t+1)$ = $P'(t)$;
S_{12}. **endfor**

The best solution in the last generation is the output of the program.
The pattern (the sequence of code that is repeatedly used) is given by steps $S_5 - S_9$.
The parameters of the standard GA are given in Table VI. Results are given in Table VII.

TABLE VI

THE PARAMETERS OF A STANDARD GA FOR EXPERIMENT 2.

Parameter	Value
Population size	50
Individual encoding	fixed-length array of real values
Number of generations	50
Crossover probability	1
Crossover type	Convex Crossover with $\alpha = 0.5$
Mutation	Gaussian mutation with $\sigma = 0.01$
Mutation probability	1
Selection	Binary Tournament

TABLE VII

THE RESULTS OBTAINED BY APPLYING THE EVOLVED EA AND THE STANDARD GA TO THE CONSIDERED TEST FUNCTIONS. STDDEV

STANDS FOR THE STANDARD DEVIATION. THE RESULTS ARE AVERAGED OVER 500 RUNS.

Test function	Evolved EA				Standard GA			
	Best run	Worst run	Mean	StdDev	Best run	Worst run	Mean	StdDev
f_1	2.62E-5	4.306	0.128	0.401	1.424E-5	5.503	0.335	0.691
f_2	6.878E-5	366.563	27.115	38.991	3.990E-5	468.196	25.963	35.553
f_3	4.852E-3	2.297	0.250	0.364	3.676E-3	4.708	0.487	0.571
f_4	6.244E-5	458.838	38.277	58.218	4.899E-4	417.656	38.821	57.073
f_5	6.592E-3	14.279	2.996	2.306	3.283E-3	14.701	3.032	2.120
f_6	5.513E-2	9801.02	339.818	993.006	4.310E-3	27823.4	485.559	1935.170
f_7	3.073E-3	15.361	1.861	1.577	4.636E-3	19.304	3.243	3.811
f_8	1.036E-2	8.536	2.775	1.756	4.915E-3	9.507	2.905	1.599
f_9	2.163E-2	3.520	0.509	0.340	2.123E-2	3.387	0.533	0.323
f_{10}	-1641.83	-569.444	-1010.610	171.799	-1642.01	-610.59	-998.549	174.782

Taking into account the averaged values we can see in table VII that the evolved EA significantly performs better than the standard GA in 9 cases (out of 10). For the test function f_2 the latter performs better than the evolved pattern.

When taking into account the solution obtained in the best run, the standard GA performs better than the evolved EA in 8 cases (out of 10).

When taking into account the solution obtained in the worst run, the evolved EA performs better than the standard GA in 7 cases (out of 10).

In order to determine whether the differences between the Evolved EA and the standard GA are statistically significant, we use a t-test with 95% confidence. Only the best solutions in each run have been taken into account for these tests. Before applying the t-test, an F-test has been used for determining whether the compared data have the same variance. The P-values of a two-tailed t-test are given in Table VIII.

TABLE VIII

THE RESULTS OF THE t-TEST AND F-TEST WITH 499 DEGREES OF FREEDOM.

Function	F-Test	t-Test
f_1	9.22E-10	8.59E-10
f_2	3.94E-2	6.25E-1
f_3	5.08E-23	1.72E-14
f_4	6.57E-1	8.81E-1
f_5	5.97E-2	7.96E-1
f_6	5.65E-47	1.34E-1
f_7	2.75E-77	2.10E-13
f_8	3.69E-2	2.21E-1
f_9	2.33E-1	2.48E-1
f_{10}	7.00E-1	2.71E-1

Table VIII shows that the difference between the EA embedding the evolved pattern and the standard GA is statistically significant ($P < 0.05$) for 6 test problems.

Experiment 3

We are also interested in analyzing the relationship between the number of generations of the evolved EA and the quality of the solutions obtained by applying the evolved EA to the considered test functions. The parameters of the Evolved EA are given in Table V and the parameters of the standard GA are given

in Table VI.

The results of this experiment are depicted in Figure 4 (the unimodal test functions) and in Figure 5 (the multimodal test functions).

Figures 4 and 5 show that the Evolved EA is scalable regarding the number of generations. We can see a continuous improvement tendency during the search process for all test functions ($f_1 - f_{10}$).

Experiment 4

We are also interested in analyzing the relationship between the population size of the evolved EA and the quality of the solutions obtained by applying the evolved EA to the considered test functions. The parameters of the Evolved EA are given in Table V and the parameters of the standard GA are given in Table VI.

The results of this experiment are depicted in Figure 6 (the unimodal test functions) and in Figure 7 (the multimodal test functions).

Figures 6 and 7 show that the Evolved EA is scalable on what concerns the number of individuals in the population. We can see a continuous improvement tendency during the search process for all test functions ($f_1 - f_{10}$).

V. Conclusions and further work

A new model for evolving Evolutionary Algorithms has been proposed in this paper. A detailed description of the proposed approach has been given, thus allowing researchers to apply the method for evolving Evolutionary Algorithms that could be used for solving problems in their fields of interest.

The proposed model has been used for evolving Evolutionary Algorithms for function optimization. Numerical experiments emphasize the robustness and the efficacy of this approach. The evolved Evolutionary Algorithms perform similarly and sometimes even better than some standard approaches in the literature for the considered test functions.

In order to evolve high quality EAs and assess their performance, an extended set of training problems should be used. This set should include problems from different fields, such as function optimization, symbolic regression [18], the Traveling Salesman Problem [14], [15], the Quadratic Assignment Problem [20], [22] classification etc. Further efforts will be dedicated to the training of such an algorithm which should have an increased generalization ability.

An extended set of operators will be used in order to obtain more powerful Evolutionary Algorithms. This set will include operators that compute the fitness of the best/worst individual in the population. In this case, the evolved EA will have the "elitist" feature which will allow us to compare it with more complex evolutionary schemes like steady-state [34].

For obtaining better results we also plan to evolve the parameters involved by the pattern (e.g. mutation probability, crossover probability etc). In this case, the evolved structure will contain a MEP chromosome and some numerical values for the parameters whose optimal value we want to find.

Only fixed-size populations have been used in our experiments. Another extension of the proposed approach will take into account the scalability of the population size.

Further effort will be focused on evolving evolutionary algorithms for other difficult (NP-Complete problem) [10], [15], [31].
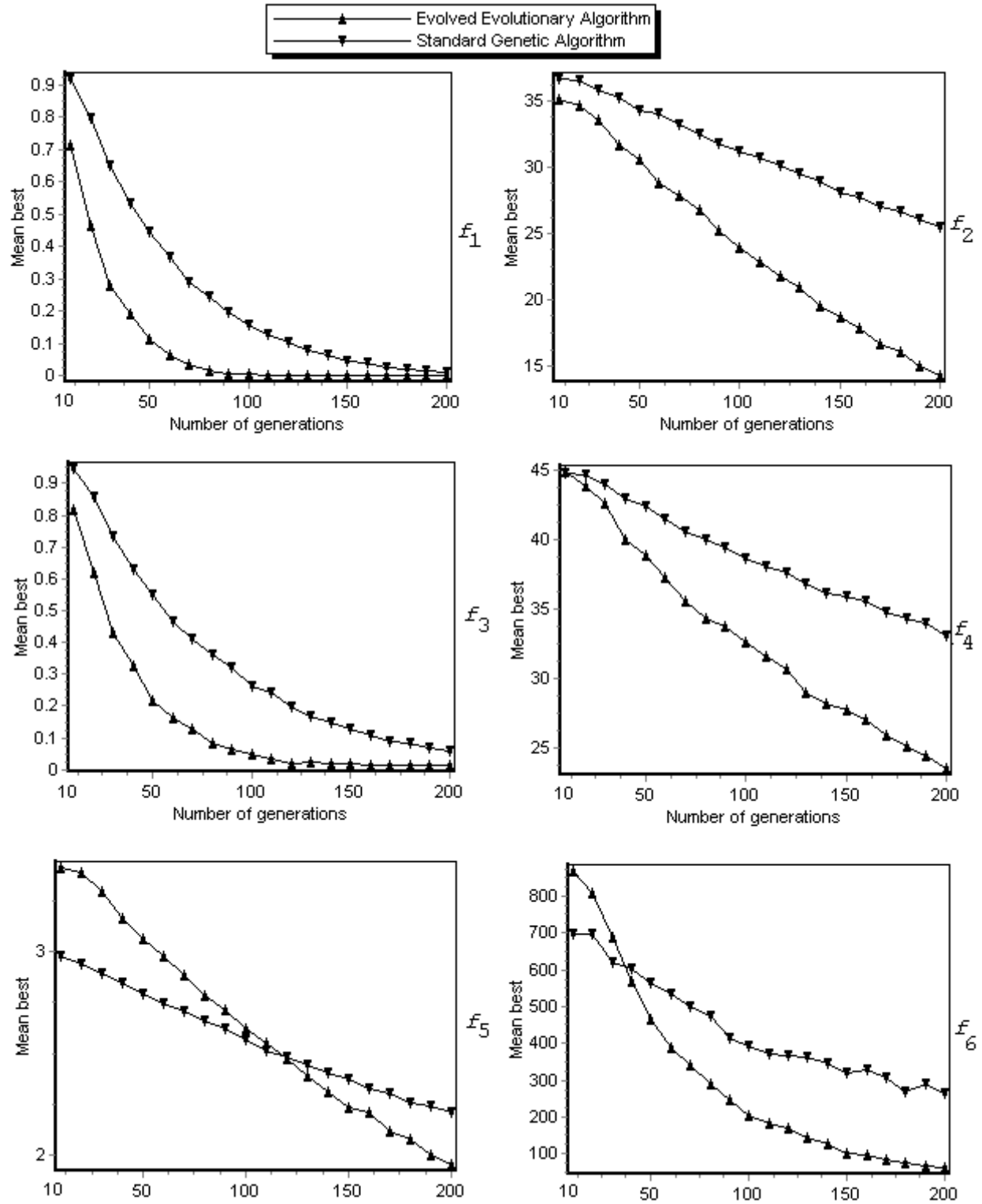
Fig. 4. The relationship between the number of generations and the quality of the solutions obtained by the Evolved EA and by the standard GA for the unimodal test functions $f_1 - f_6$. The number of generations varies between 10 and 200. Results are averaged over 100 runs.
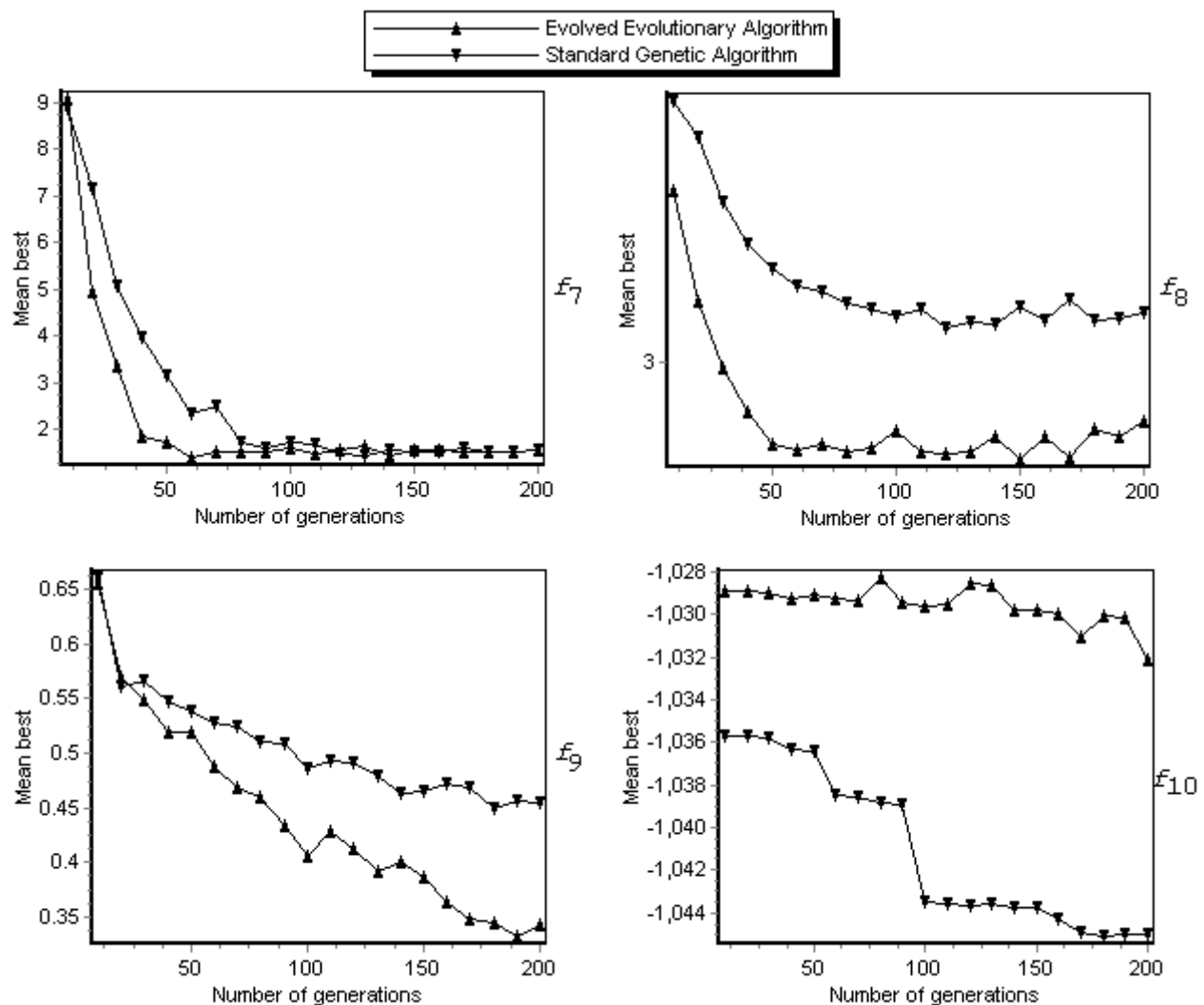
Fig. 5. The relationship between the number of generations and the quality of the solutions obtained by the Evolved EA and by the standard GA for the multimodal test functions $f_7 - f_{10}$. The number of generations varies between 10 and 200. Results are averaged over 100 runs.

REFERENCES

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

[2] Angeline, P. J., (1995), Adaptive and Self-Adaptive Evolutionary Computations, *Computational Intelligence: A Dynamic Systems Perspective*, pages 152-163, IEEE Press: New York, USA.

[3] Angeline, P. J., (1996), Two Self-Adaptive Crossover Operators for Genetic Programming. In Angeline, P. and Kinnear, K. E., editors, *Advances in Genetic Programming II*, pages 89-110, MIT Press, Cambridge, MA, USA.

[4] Back, T., (1992), Self-Adaptation in Genetic Algorithms, In *Toward a Practice of Autonomous Systems: Proceedings of the first European Conference on Artificial Life*, pages 263-271, MIT Press, Cambridge, MA, USA.

[5] Banzhaf, W., Nordin, P., Keller, R. E. and Francone, F. D., (1998), *Genetic Programming - An Introduction On the automatic evolution of computer programs and its applications*, dpunkt/Morgan Kaufmann, Heidelberg/San Francisco.

[6] Beyer, H.G, (1994), Toward a Theory of Evolution Stategies: the (μ, λ) Strategy, *Evolutionary Computation*, Vol. 2, Nr. 4, pp. 381-408, MIT Press.

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

[8] Brameier, M. and Banzhaf, W., (2002), Explicit Control of Diversity and Effective Variation Distance in Linear Genetic Programming, In E. Lutton, J. Foster, J. Miller, C. Ryan and A. Tettamanzi Editors, *European Conference on Genetic Programming IV*, Springer Verlag, Berlin, pages 38-50, 2002.

[9] Brameier, M. and Banzhaf, W., (2001), Evolving Teams of Predictors with Linear Genetic Programming, *Genetic Programming and Evolvable Machines*, 2:381-407, Kluwer.

[10] Burkard, R. E. and Rendl, F., (1991), QAPLIB-A Quadratic Assignment Problem Libray, *European Journal of Operational Research*, pages 115-119.
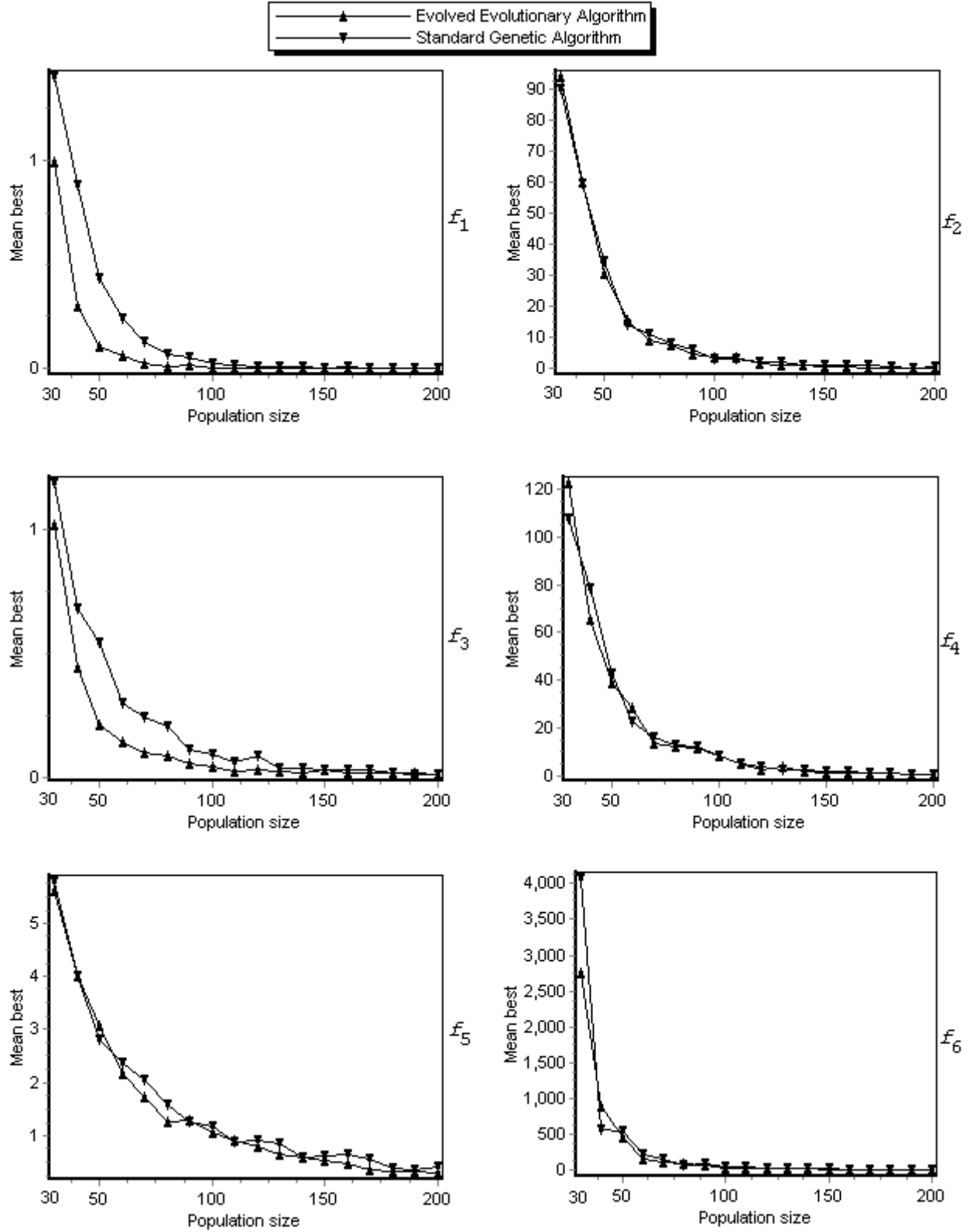
Fig. 6. The relationship between the population size and the quality of the solutions obtained by the Evolved EA and by the standard GA for the unimodal test functions $f_1 - f_6$. Population size varies between 30 and 200. Results are taken from the last generation and averaged over 100 independent runs.
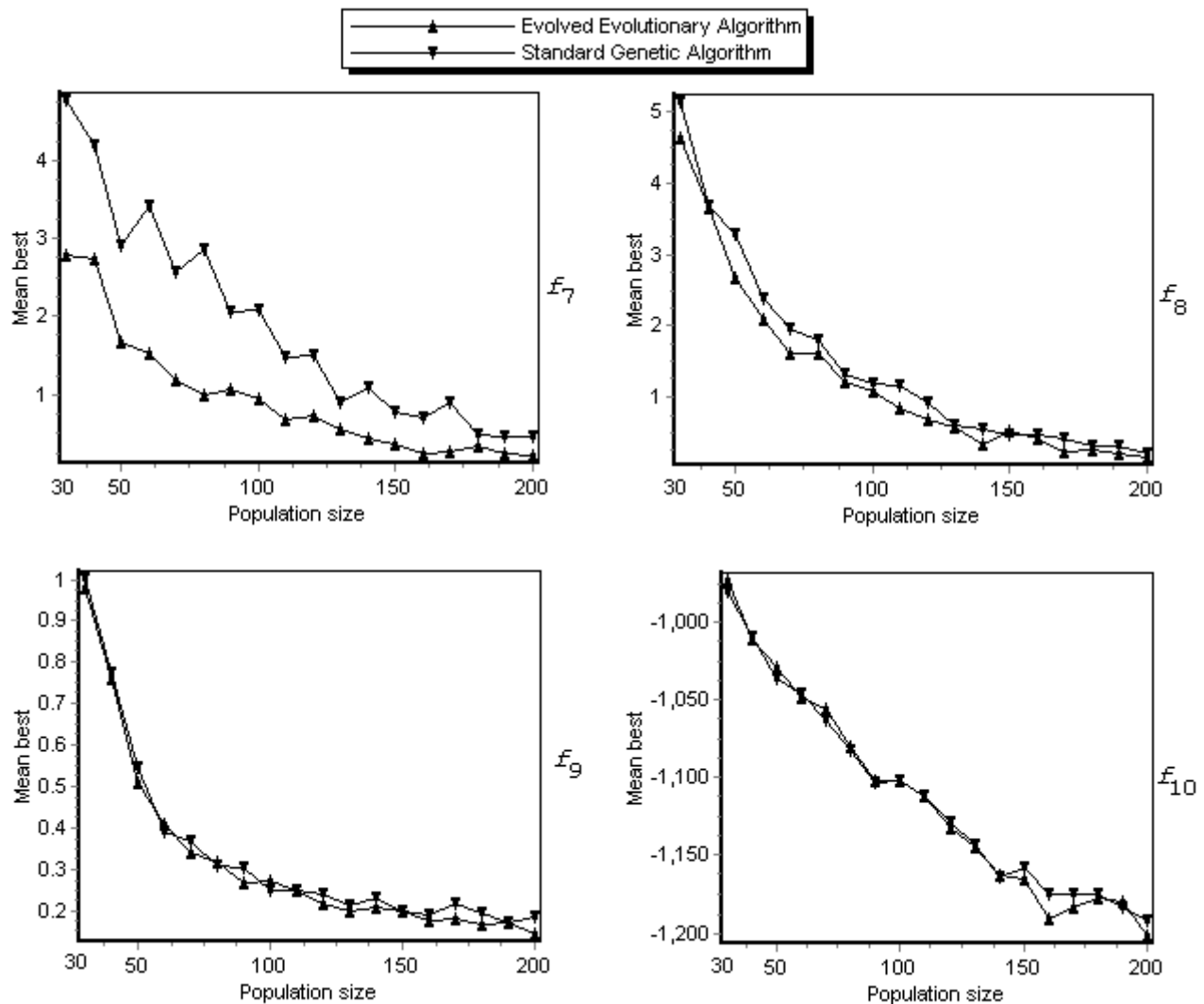
Fig. 7. The relationship between the population size and the quality of the solutions obtained by the Evolved EA and by the standard GA for the multimodal test functions $f_7 - f_{10}$. Population size varies between 30 and 200. Results are averaged over 100 runs.

[11] Cormen, T. H., Leiserson, C. E. and Rivest, R. R. (1990), *Introduction to Algorithms*, MIT Press.

[12] Edmonds, B., (2001), Meta-Genetic Programming: Co-evolving the Operators of Variation. *Electrik on AI*, Vol. 9, pages 13-29.

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

[14] Freisleben, B. and Merz, P., (1996), A Genetic Local Search Algorithm for Solving Symmetric and Asymmetric Traveling Salesman Problems, In *IEEE International Conference on Evolutionary Computation 1996*, pages 616-621, IEEE Press.

[15] Garey, M. R. and Johnson, D. S., (1979), *Computers and Intractability: A Guide to NP-Completeness*, Freeman & Co, San Francisco, CA.

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

[17] Holland, J. H., (1975), *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor.

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

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

[20] Krasnogor, N., (2002), Studies on the Theory and Design Space of Memetic Algorithms, PhD Thesis, University of the West of England, Bristol.

[21] Merz, P. and Freisleben, B., (1997), Genetic Local Search for the TSP: New Results, In *IEEE International Conference on Evolutionary Computation*, pages 616-621, IEEE Press.

[22] Merz, P. and Freisleben, B., (2000), Fitness Landscape Analysis and Memetic Algorithms for the Quadratic Assignment Problem, *IEEE Transaction On Evolutionary Computation*, 4:337-352, IEEE Press, NY, USA.

[23] Nordin, P., (1994), A Compiling Genetic Programming System that Directly Manipulates the Machine-Code. In Kinnear K.E. editors, *Advances in Genetic Programming I*, pages 311-331, MIT Press, Cambridge, MA, USA.

[24] Oltean, M. and Groşan, C., (2003), A Comparison of Several Linear Genetic Programming Techniques, *Complex-Systems*, Vol 14, Nr. 4, pages 282-311.

[25] Oltean, M. and Groşan, C., (2003), Evolving Evolutionary Algorithms using Multi Expression Programming, in Banzhaf, W. (et al) editors, *European Conference on Artificial Life VII*, LNAI 2801, pages 651-658, Springer-Verlag, Berlin, Germany.

[26] Oltean, M., (2003), Solving Even-parity problems with Multi Expression Programming, in K. Chen (et al) editors, *International Workshop on Frontiers in Evolutionary Algorithm V*, Triangle Research Park, pages 315-318.

[27] Oltean, M., (2003), Evolving Evolutionary Algorithms for Function Optimization, in K. Chen (et al) editors, *International Workshop on Frontiers in Evolutionary Algorithm V*, Triangle Research Park, pages 295-298.

[28] Oltean, M. and Dumitrescu, D., (2004), Evolving TSP heuristics with Multi Expression Programming, *International Conference on Computational Sciences ICCS'04*, Edited by M. Bubak, (et al), Vol II, pages 675-678, LNCS Springer-Verlag, Berlin, Germany.

[29] Oltean, M., (2005), Evolving Evolutionary Algorithms using Linear Genetic Programming, *Evolutionary Computation*, MIT Press, (to appear in No. 3).

[30] Prechelt, L., (1994), PROBEN1 - A Set of Neural Network Problems and Benchmarking Rules, Technical Report 21, University of Karlsruhe, Germany.

[31] Reinelt, G., (1991), TSPLIB - A Traveling Salesman Problem Library, ORSA, *Journal of Computing*, Nr. 4, pages 376-384.

[32] Spector, L. and Robinson, A., (2002), Genetic Programming and Autoconstructive Evolution with the Push Programming Language, *Genetic Programming and Evolvable Machines*, Nr. 1, pages 7-40, Kluwer.

[33] Stephens, C. R., Olmedo, I. G., Vargas, J. M. and Waelbroeck, H., (1998), Self-Adaptation in Evolving Systems, *Artificial Life* 4, pages 183-201.

[34] Syswerda, G., (1989), Uniform Crossover in Genetic Algorithms. In Schaffer, J. D. editors, *The 3rd International Conference on Genetic Algorithms*, pages 2-9, Morgan Kaufmann Publishers, San Mateo, CA.

[35] Teller, A., (1996), Evolving Programmers: the Co-evolution of Intelligent Recombination Operators. In Angeline, P. and Kinnear, K. E., editors, *Advances in Genetic Programming II*, pages 45-68, MIT Press, USA.

[36] Yao, X., Liu, Y. and Lin, G., (1999), Evolutionary Programming Made Faster. *IEEE Transaction on Evolutionary Computation*, Nr. 2, pages 82-102, IEEE Press, NY, USA.

[37] Wolpert, D. H. and McReady, W. G., (1995), No Free Lunch Theorems for Search, Technical Report SFI-TR-05-010, Santa Fe Institute, USA.

[38] Wolpert, D. H. and McReady, W. G., (1997), No Free Lunch Theorems for Optimization. *IEEE Transaction on Evolutionary Computation*, Nr. 1, pages 67-82, IEEE Press, NY, USA.