Evolutionary Design of Evolutionary Algorithms

Laura Dioşan and Mihai Oltean
Department of Computer Science, Faculty of Mathematics and Computer Science,
Babeş-Bolyai University, Kogalniceanu 1,
Cluj-Napoca, 400084, Romania,
{lauras, moltean}@cs.ubbcluj.ro

Abstract

Manual design of Evolutionary Algorithms (EAs) capable of performing very well on a wide range of problems is a difficult task. Another possibility is to let the evolution to discover the optimal structure and the parameters of the algorithm. Since we cannot build an EA capable of solving all the problems in the best way, we have to find other manners to construct algorithms that perform very well for some particular problems. One possibility (which is explored in this paper) is to let the evolution discover the optimal structure and parameters for the EA used for solving a specific problem. To this end a new model for automatic generation of EAs by evolutionary means is proposed in this paper. The model is based on a simple Genetic Algorithm (GA). Every GA chromosome encodes an EA, which is used for solving a particular problem. Several Evolutionary Algorithms for function optimization are generated by using the considered model. Numerical experiments show that the evolved EAs perform similarly and sometimes even better than standard approaches for several well-known benchmarking problems.

1 Introduction

Evolutionary Algorithms (EAs) [1, 2] are heuristics that mimic the Darwinian principles of evolution in order to solve complex optimization problems. They perform well in solving real-world problems, especially for which the fitness landscape is complex. EAs have been applied in a variety of domains including numerical function optimization, combinatorial optimization, adaptive control and machine learning. Many evolutionary models, representations and search operators have also been proposed. There will always be questions related to the usefulness of a particular scheme for solving a wide range of problems.

A breakthrough arose in 1995 when Wolpert and McReady unveiled their work on the *No Free Lunch* (NFL) theorems for *Search* [3] and *Optimization* [4]. The No Free Lunch theorems state that all the black-box algorithms have the same average performance over the entire set of optimization problems (a black-box algorithm does not take into account any information about the problem or the particular instance which is being solved.). The NFL results undermine all the efforts for developing a universal black-box optimization algorithm capable of solving all the optimization problems in the best manner.

Our approach is to evolve a full-featured EA (i.e. the output of our main program will be an EA capable of performing a given task). Thus, we will work with EAs at two levels: the first (macro) level consists in a steady-state EA [5] 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. A solution represents an evolved sequence of genetic operations and their parameters, which are performed by an evolutionary algorithm for solving a particular problem.

The rules employed by the evolved EAs during a generation are not pre-programmed. These rules are automatically discovered by the evolution. The evolved EA could be a generational one (the generations do not overlap), a steady-state evolutionary algorithm, or a mixture of these two models. In this context, our research was motivated by the need to answer several important questions concerning Evolutionary Algorithms. The most important questions are:

- *Can Evolutionary Algorithms be automatically synthesized by using only the information about the problem which is being solved?* [6].
- *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 Evolutionary Algorithm for a particular problem. For instance, in a standard GA the sequence is the following: selection, recombination and mutation. However, how do we know that this scheme is the best for a specific problem (or problem instance)?*
- *Which are the optimal values for crossover and mutation probabilities or for other parameters involved by the genetic operators?*

We had better let the evolution find the answer for us.

The paper is organized as follows. An overview of the related work in the field of evolving EAs is made in section 2. The model used for evolving EAs is presented in section 3. Test problems are described in section 4. Various numerical experiments are performed in section 5. Several EAs for function optimization are also evolved in that section. Scalability issues are discussed in section 6. A numerical comparison with other methods for evolving EAs is performed in section 7. Generalization ability is tested in section 8. Differences among the methods used for evolving EAs are discussed in section 9. A brief analysis over the evolution of EAs is given in section 10. Further research directions are suggested in section 11. Finally, section 12 concludes our paper.

2 Related work

The ability of an evolutionary algorithm to adapt its search strategy during the optimisation process is a key concept in evolutionary computation [7]. Several approaches evolve genetic operators for solving difficult problems [8, 9, 10, 11]. In [8] were introduced two adaptive crossover operations, Selective Self-Adaptive Crossover and Self-Adaptive Multi-Crossover, both of which were designed to evolve crossover points for sub-trees. The results of the experiments indicate that a static weight system for crossover points in Genetic Programming is not as desirable as an adaptive system. In his paper on Meta-Genetic Programming, Edmonds [9] used two populations: a standard Genetic Programming population and a co-evolved population of operators that act on the main population. Note that all these approaches use a fixed EA, which is not changed during the search process.

Spector and Robinson [12] described a programming language designed for evolutionary computation (called Push), which supports a self-adaptive form called *auto constructive evolution*. In such system, the evolving programs construct their own children including the variation mechanisms and the evolutionary process itself. This means that each evolve program also contain code for reproduction and diversification. When used within a standard genetic programming system Push provides several forms of self-adaptation including the automatic evolution of modules and program architecture. The obtained system – called Push Genetic Programming – was used for symbolic regression problems [12].

Kantschik et al. have introduced the evolution of operators for Genetic Programming by means of Genetic Programming. Specifically, meta-evolution of recombination operators in graph-based Genetic Programming is applied and compared to other methods for the variation of recombination operators in graph-based Genetic Programming [13].

Several attempts at evolving EAs using similar techniques were performed in the past. A generational EA was evolved [14, 15] by using the Linear Genetic Programming (LGP) technique [16, 17]. A non-generational EA was evolved [18] by using the Multi Expression Programming (MEP) technique [18]. MEP was used again for evolving only the kernel of an EA [19]. A new model for automatic generation of Evolutionary Algorithms (EAs) by evolutionary means was proposed in [20]. The model is based on a simple Genetic Algorithm (GA). Numerical experiments have shown [14, 18, 19, 20] that the evolved EAs perform similarly and sometimes even better than the standard evolutionary approaches with which they are compared. In the next subsections we briefly described several previous approaches that are the most related to the current one.

2.1 Linear Genetic Programming based approach

Linear Genetic Programming (LGP) [16, 17, 21] uses a specific linear representation of computer programs. It evolves programs of an imperative language (like ***C***) instead of the tree-based Genetic Programming 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:

Algorithm 1 LGP_Program(double r[8])

```
Randomly_initialize_the_registers();
                                   // repeat for a number of generations
for k=0 to MaxGenerations do
   r[0] = r[5] * r[3];
   r[7] = r[3] - r[6];
   r[4] = sin(r[2]);
   r[2] = r[0] + r[2];
   r[6] = r[1] * 45;
   r[2] = r[4] - r[3];
   r[1] = sin(r[6]);
   r[3] = exp(r[5]);
end for
```

Instead of evolving such deterministic computer programs, a full-featured evolutionary algorithm (*i.e.* the output of the main program will be an EA capable of performing a given task) is evolved. Thus, the evolutionary principles are applied at two levels: the first (macro) level consists in a steady-state EA [22] 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.

For the first (macro) level EA an evolutionary model similar to LGP [16, 17, 21] is utilised. The structure of the standard LGP has been adapted for evolving EAs [15]. Instead of working with registers, the LGP program modifies an array of individuals (the population of the micro level EA). Suppose that the array of individuals (the population) which will be modified by an LGP program is denoted by *Pop*.

The set of function symbols involve in LGP algorithm consists of genetic operators that may appear in an evolutionary algorithm. There are usually three types of such genetic operators: *Selection*, *Crossover* and *Mutation*.

The *Initialization* operator is used to generate the micro population and it is not evolved like all the other operators.

The LGP statements are considered genetic operations executed during an EA generation:

- *Selection*(i_1, i_2) - selects the better solution among two already existing solutions i_1 and i_2
- *Crossover*(i_1, i_2) - recombines two already existing solutions i_1 and i_2,
- *Mutation*(i_1) - varies an already existing solution i_1.

Note that i_1 or i_2 refer to the potential solutions to be identified by the micro-level algorithm for the problem that must be solved. In this case, the micro algorithm performs a numerical optimisation of a given function. Therefore, a solution is represented as an array of real values, the length of this array being given by the dimension number of the problem (by the problem size).

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 first generation of individuals.

An example of an LGP chromosome encoding an EA is given below:

Algorithm 2 LGP-based_EA_Program(Chromosome Pop[8])

```
Randomly_initialize_the_population();
                                        // repeat for a number of generations
for k=0 to MaxGenerations do
                                        // starting point of the evolved part
  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]);
                                        // ending point of the evolved part
end for
```

This evolved part refers to the following operations performed in a generation of the evolved EA: first of all, the chromosome *Pop*[5] is mutated and the resulted offspring replaces the first individual (*Pop*[0]). Than, the best chromosome between *Pop*[3] and *Pop*[6] is selected and it replaces the last chromosome (*Pop*[7]) and so on. Because variable size populations are involved in such mechanism, not all the individuals are updated. For instance, in the previous example *Pop*[5] is never updated.

Note that only the operations inside the **for** instruction are evolved. Everything else is kept fixed.

In order to compute the quality of the LGP chromosome the EA encoded there is run on the particular problem which is being solved. Roughly speaking, the fitness of an LGP individual is equal to the fitness of the best solution generated by the evolutionary algorithms encoded in that LGP chromosome.

2.2 Multi Expression Programming based approach

Multi Expression Programming (MEP) [18, 23] is a Genetic Programming variant that uses linear chromosomes for solution encoding. A unique MEP feature is its ability of encoding multiple solutions of a problem in a single chromosome.

Substrings of a variable length represent MEP genes. 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. A tree can be constructed for each gene by

following the function pointers. The entire chromosome is seen as a forest of trees. This representation is similar to Cartesian Genetic Programming [24].

A unique feature of MEP is the ability to choose which sub-tree will provide the solution of the problem. Usually the best sub-tree encoded in an MEP chromosome is chosen to represent the chromosome.

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 the following sets of functions $F = \{+, *\}$ and terminals $T = \{a, b, c, d\}$ are utilised. An example of a MEP chromosome using the sets F and T is given below:

1: a
2: b
3: + 1, 2
4: c
5: d
6: + 4, 5
7: * 3, 5
8: + 2, 6

Phenotypic translation of a MEP chromosome is obtained by parsing the chromosome top-down. The previous chromosome encodes 8 potential expressions:

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

The chromosome fitness is usually defined as the fitness of the best expression encoded by that MEP chromosome.

There have been designed two ways for evolving EAs by using MEP. The first one (synthesized in Section 2.2.1) generates full non-generational EAs. The second one (described in Section 2.2.2) generates only a small sequence of instructions, which are repeatedly utilised in order to create new individuals.

2.2.1 Evolving full EAs

In order to use MEP for evolving EAs, the sets of terminal and function symbols have been redefined in [18].

As in the LGP-based model, the previously presented operators (*Selection*, *Crossover* and *Mutation*) are utilised, but the *Initialization* operator is evolved also. The above-mentioned operators along with their meaning are listed below:

- *Initialization* - randomly initializes a solution,
- *Selection*(i_1, i_2) - selects the better solution among two already existing solutions i_1 and i_2
- *Crossover*(i_1, i_2) - recombines two already existing solutions i_1 and i_2,
- *Mutation*(i_1) - varies an already existing solution i_1.

These operators will act as symbols that may appear in an MEP chromosome. The only operator that generates a solution independent of the already existing solutions is the *Initialization* operator. This operator composes the terminal set. The other operators are considered function symbols.

An MEP chromosome, storing an EA, can look like this:

Algorithm 3 MEP-based_EA_Chromosome

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 better 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 EAs (in fact 8 EAs). They are given in Table 1. 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 EA encoded in a chromosome represents that chromosome (it provides the fitness of that chromosome).

The complexity of the EAs encoded in an MEP chromosome varies from very simple (EAs made up of a single instruction – see EA_1 or EA_2) to very complex (sometimes using all the genes of the MEP chromosome – see EA_7 or EA_8). This is very useful because the complexity of the EA that is claimed in order to solve a problem is not known in advance. The required algorithm could be very simple (in this case, the simplest individuals encoded by MEP are very useful) or it could be very complex (in this case, the most complex EAs are taken into account).

The algorithm proposed in [18] is a non-generational one (there are no generations). There is only one sequence of genetic operations.

Table 1: Evolutionary Algorithms encoded in the MEP chromosome C

EA_1	EA_2	EA_3	EA_4
i_1= 1: *Initialization*	i_1= 2: *Initialization*	i_1= 1: *Initialization*	i_1 = 1: *Initialization*
		i_2= 3: *Mutation*(i_1)	i_2= 3: *Mutation*(i_1)
			i_3= 4: *Selection*(i_1,i_2)

EA_5	EA_6	EA_7	EA_8
i_1= 1: *Initialization*	i_1= 1: *Initialization*	i_1= 1: *Initialization*	i_1= 1: *Initialization*
i_2= 2: *Initialization*	i_2= 3: *Mutation*(i_1)	i_2= 2: *Initialization*	i_2= 2: *Initialization*
i_3= 3: *Mutation*(i_1)	i_3= 4: *Selection*(i_1, i_2)	i_3= 3: *Mutation*(i_1)	i_3= 3: *Mutation*(i_1)
i_4= 4: *Selection*(i_1, i_3)	i_4= 6: *Mutation*(i_3)	i_4= 4: *Selection*(i_1, i_3)	i_4= 4: *Selection*(i_1, i_3)
i_5= 5: *Crossover*(i_2, i_4)		i_5= 5: *Crossover*(i_2, i_4)	i_5= 6: *Mutation*(i_4)
		i_6= 7: *Mutation*(i_5)	i_6= 8: *Crossover*(i_2, i_5)

The quality of an EA encoded in an MEP chromosome is computed as in the case of the LGP approach (see Section 2.1).

2.2.2 Evolving EAs with patterns

Instead of evolving an entire EA it is possible to evolve only the heart (the kernel) of the algorithm represented by the sequence of instructions that is repeatedly applied in order to generate new offspring by taking information from the current population [19].

In the previously described models (MEP-based and LGP-based), the search space of the EAs 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, the model proposed in [19] evolves a small piece of code that will be used repeatedly 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(); {choose randomly two individuals and return the better of them in p_1 }
p_2 = Selection(); {choose randomly two individuals and return the better 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 main advantage of this approach is its reduced complexity: the size of the pattern is considerably smaller than the size of the entire EA which was evolved in [14, 18]. The patterns in an EA could be assimilated with the Automatically Defined Functions (ADFs) in Genetic Programming [25].

In the model proposed in [19] the pattern was represented as an MEP computer program whose instructions are executed during the EA evolution.

The *Initialization* operator was not involved, as the purpose was to evolve a small piece of code that would be used in order to generate a new population based on the old one. This operator is applied only once, at the moment of population initialization. The other three operators, *Selection, Crossover* and *Mutation*, are used within the evolved pattern:

- *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(i_1, i_2)* - recombines two already existing solutions i_1 and i_2,

- *Mutation(i_1)* - varies an already existing solution i_1.

This model involved the following sets:

- the function set $F = \{Crossover, Mutation\}$
- the terminal set is $T = \{Selection\}$.

The model proposed in [19] replaces the terminal set by the new terminal symbol $\{Selection\}$ which is specific to this purpose. Also, the function set is replaced by $\{Crossover, Mutation\}$. An example of the MEP pattern is given below:

1: *Selection()*;
2: *Selection()*;
3: *Crossover 1, 2*;
4: *Mutation 3*;

Please note that, unlike the previous model, the *Selection* operator has no argument in this case.
This MEP chromosome should be interpreted as follows:

- An individual (let us denote it by ind_1) is selected from the current population by using a binary tournament mechanism - instruction *Selection()*
- Another individual (let us denote it by ind_2) is selected from the current population by using a binary tournament mechanism - instruction *Selection()*
- Individuals ind_1 and ind_2 are recombined using a representation-dependent crossover. A new individual ind_3 is obtained - instruction *Crossover 1, 2*
- Individual ind_3 is mutated. The result of the mutation is a new individual denoted by ind_4 - instruction *Mutation 3*.

After this briefly review of other adaptive EAs described in literature, in the next section we will explain the details about our model for evolving EAs.

3 Genetic Algorithms for evolving Evolutionary Algorithms

The model proposed for evolving EAs is described in this section. We deal with EAs at two levels: a macro-level EA and a micro-level EA. The macro EA is a GA manipulating a sequence of genetic operators utilized by the micro EA during a generation. In what follows, we will denote the operations, the chromosomes and the parameters involved in the macro level EA by using the *macro* or M prefix notation and those involved in the micro level EA by using the *micro* or μ prefix notation:

Object	Notations for the macro level	Notations for the micro level
Chromosome	MChromosome	μChromosome
Generation	MGeneration	μGeneration
Crossover	MCrossover	μCrossover
Mutation	MMutation	μMutation
Selection	MSelection	μSelection
Population	MPop	μPop
Size of population	MPopSize	μPopSize
Number of generations	MNoOfGenerations	μNoOfGenerations
Crossover probability	Mp_c	μp_c
Mutation probability	Mp_m	μp_m

3.1 Individual representation for evolving EAs

In order to use GAs for evolving EAs we have to modify the structure of a GA chromosome. The macro level chromosome is an evolutionary program which manipulates an array of individuals (the population of the μEA). The macro-chromosome, used for evolving μEAs, consists in an array of values that specify the operations that are performed by the μEA.

There are 2 types of operations performed inside a μEA:

- Crossover and mutation that generate new individuals. These operations are called *Chromosome creation operations*.

- Append and replace that tells what to do with the newly created individuals. These operations are called *Population altering operations*.

Each gene of a macro chromosome has two values: the first value is used in order to store the *chromosome creation operations*, while the second value of a gene refers to the *population altering operation*.

3.1.1 Chromosome creation operations

As we already mentioned, three types of genetic operators may usually appear in an EA. These genetic operators are:

- *Selection* - selects a solution from several existing solutions,
- *Crossover* - recombines two existing solutions in order to generate one offspring and
- *Mutation* - varies an existing solution.

We will call these operations *micro Selection* (μSelection), *micro Crossover* ($\mu Crossover$) and *micro Mutation* ($\mu Mutation$) because they are performed inside the *micro EA* (μEA).

Crossover requires two input parameters (the parents) and mutation requires one input parameter (an individual). The parameters for these operators are usually provided by the selection procedure. This is why we embed in the calls of *Crossover* and *Mutation* a call of *Selection* function. More specifically, we have three major types of *chromosome creation operations* in a modified GA chromosome. These instructions are:

$off_1 = \mu Crossover\ (\mu Selection(), \mu Selection());$	{ Crossover two individuals obtained by selection } { procedure. The result will be a new individual. }
$off_2 = \mu Mutation\ (\mu Selection());$	{ Mutate the individual obtained by selection } { procedure. The result will be a new individual. }
$off_3 = \mu Mutation\ (\mu Crossover\ (\mu Selection(), \mu Selection()));$	{ Crossover two individuals (obtained by } { selection procedure) and mutate the offspring. } { The result will be a new individual. } { In what follows this operation is denoted by } { $\mu CrossMutation$. }

Remarks:

(i) The $\mu Selection$ operator acts as a binary tournament selection. The better of two individuals is always accepted as the result of the selection.

(ii) The μCrossover and the μMutation are representation-dependent. For instance:

- if we want to evolve a μEA with binary representation for function optimization we may use the set of genetic operators having the following functionality: $\mu Crossover$ – one cutting point crossover [2], $\mu Mutation$ – bit-wise mutation [2]
- if we want to evolve a μEA with real representation for function optimization we may use the set of genetic operators having the following functionality: $\mu Crossover$ – uniform arithmetical crossover [26], $\mu Mutation$ – Gaussian mutation [27], [28]
- if we want to evolve a μEA for solving the Travelling Salesmen Problem [29] we may use DPX as a μCrossover operator and 2-opt as a μMutation operator [30].

(iii) The $\mu Crossover$ operator in our model always generates a single offspring from two parents. Still, crossover operators generating two offspring may be designed in order to fit our evolutionary model.

(vi) The $\mu Crossover$ and $\mu Mutation$ operators are applied with specific probabilities. If the conditions for crossover/mutation are not met, the offspring will be equal to (one of) the parent(s).

3.1.2 Population altering operations

When a new μIndividual is obtained by performing one of previously presented operations, we have to decide what to do with it. In our model, we have three possibilities:

- append it to the current population ($\mu Append$),
- overwrite the worst individual in the current population ($\mu ReplaceWorst$) or

- replace the current individual (the parent) with the new μindividual ($\mu ReplaceCurrent$).

These possibilities have been chosen in order to match some well-known evolutionary schemes. Table 2 shows the operations performed in our evolved EA along with the source of inspiration for them. If one of the schemes is better than the other is, then our algorithm will evolve in that direction.

Table 2: The population altering operations along with an existing similar scheme. Note that a perfect matching between the operations performed in our evolved EA and the exact sequence of instructions performed in a particular evolutionary scheme: Steady-State GA (SSGA), Generational GA (GA) or Evolutionary Strategy (ES)) is not possible due to the constrains imposed by our model.

Operation	Explanation	Existing scheme
$\mu Append(off)$;	Append the new individual *off* (obtained by $\mu Crossover$ or $\mu Mutation$) to the population	GA, EP
$\mu ReplaceWorst(off)$;	Replace the worst individual from the population *worst* with the new individual *off* (obtained by $\mu Crossover$ or $\mu Mutation$) if *off* is better than *worst*	SSGA
$\mu ReplaceCurrent(off)$;	Replace the parent (or the best parent) with the new individual *off* (obtained by $\mu Crossover$ or $\mu Mutation$) if *off* is better than its parent	(1+1)ES

3.1.3 How to form the next generation

Because, after each $\mu Crossover$ or $\mu Mutation$, a new individual can be added to the population, we need to select the survivors of the current generation. They will form the next generation. The current population can increase in size, since the *append* operation could be involved in the schema of the evolved algorithm. After the creation of $\mu PopSize$ individuals we have to move to the next generation. We have inserted a flag (called *next gen flag*) in our evolved EA which tell us how to form the next generation. This is done in 2 ways (according to the flag value):

- 0 - the new population is formed from the newly created individuals. This is specific to generational GAs.
- 1 - the best $\mu PopSize$ individuals from the current population (which contains more than $\mu PopSize$ individuals due to the effect of $\mu Append$) oepration form the new generation.

This flag was evolved like all other genes in the MChromosome.

3.1.4 The structure of the evolved EA

A GA chromosome C, storing an evolutionary algorithm is the following:

Gene #	Gene values	
1	$off_1 = \mu Mutation(\mu Selection())$	$\mu Append(off_1)$
2	$off_2 = \mu Crossover(\mu Selection(), \mu Selection())$	$\mu Replace(off_2)$
3	$off_3 = \mu Crossover(\mu Selection(), \mu Selection())$	$\mu Append(off_3)$
4	$off_4 = \mu Mutation(\mu Crossover(\mu Selection(), \mu Selection()))$	$\mu Replace(off_4)$
5	$off_5 = \mu Mutation(\mu Selection())$	$\mu CrtReplace(off_5)$
6	$off_6 = \mu Crossover(\mu Selection(), \mu Selection())$	$\mu Append(off_6)$
7	next gen flag	

In our implementation we have used an integer representation for encoding the values of the genes inside a MChromosome as follows:

0 μCrossover
1 μMutation
2 μCrossMutation
0 μAppend
1 μReplaceWorst
2 μReplaceCurrent

By using this encoding, the previous chromosome C can be re-written as follows:

Table 3: Possible evolutionary schemes, which can be obtained by various combination of the parameters for the evolved EA.

Chromosome creation	Population altering	NextGen flag	Obtained scheme
CrossoverMutation	Append	0	GA
Mutation	Append	1	Evolutionary Programming (EP)
Mutation	ReplaceCurrent	1	- repeated (1+1) ES
CrossoverMutation	ReplaceWorst	1	SSGA

Gene #	Gene values	
1	1	0
2	0	1
3	0	0
4	2	1
5	1	2
6	0	0

Note that the number of genes from a macro chromosome is equal to the number of individuals ($\mu PopSize$) from the evolved EA population. The genes of the macro chromosome C encode different micro genetic operations. These statements are actually executed during a micro EA generation.

For this representation of a macro GA chromosome, we can apply some standard macro recombination and macro mutation operators. For instance, we can use the one-cutting point crossover and the weak mutation inspired from binary representation [1].

Since our purpose is to evolve an entire EA, we have to add a wrapper loop around the genetic operations that are executed during an EA generation. Moreover, each EA starts with a random population of individuals. Thus, the program must contain some instructions that randomly initialize the first generation of the micro EA.

Having these parameters we can simulate a wide range of existing evolutionary schemes (see Table 3) and new hybrid ones.

The μEA that corresponds to the instructions encoded in the macro chromosome C is given below:

Algorithm 4 Micro chromosome-program – a μpopulation with 6 individuals

```
RandomlyInitialization(μPop);          // This operation is not encoded in the MGA chromosome
Fitness(μPop);                         // This operation is not encoded in the MGA chromosome
for g=1 to μNoOfGenerations do
  off1 = μMutation(μSelection())[a]; μAppend(off1);                                  // gene 1
  off2 = μCrossover(μSelection(),μSelection()); μReplace(off2);                      // gene 2
  off3 = μCrossover(μSelection(),μSelection()); μAppend(off3);                       // gene 3
  off4 = μMutation(μCrossover(μSelection(),μSelection())); μReplace(off4);           // gene 4
  off5 = μMutation(μSelection()); μCrtReplace(off5);                                 // gene 5
  off5 = μCrossover(μSelection(),μSelection()); μAppend(off5);                       // gene 6
  ConstructTheNewPopulation(μPop);                 // based on the flag value
end for
```

[a] The μCrossover and μMutation are applied with μp_c and μp_m proabilities, respectively.

Remark: The initialization function, the **for** cycle, the sort and truncation functions will not be affected by the genetic operators. These parts are kept intact during the search process.

3.1.5 Finding μp_c and μp_m

We have observed that μp_m and μp_c depend on the problem being solved (this assumption was later confirmed by the values discovered for these parameters as shown in the experiments). In order to find which the values of these parameters should be (for a specific problem) we have several options:

- to encode them in the MChromosome structure and to evolve them as all other genes. Using this approach we can generally find local-optima values instead of global-optima ones.
- to check all possible values (with a given step) for μp_m and μp_c. For each such pair the algorithm is run and the results are stored. The best values for μp_m and μp_c are finally shown. There are several problems with this method too: using a too small step, the running time (for checking all combinations) would be too high. Using a too large step, some good values could be missed.

We have tested both methods and we have decided to stick with the second one. The step for searching μp_m and μp_c is 0.1. Thus 100 pairs, over the discrete space $\{0.1, 0.2, \ldots, 0.9, 1.0\} \times \{0.1, 0.2, \ldots, 0.9, 1.0\}$ must be searched. In order to reduce the computational cost generated by this approach we have implemented the following strategy for finding μp_m and μp_c:

- We explore the μp_m and μp_c in 2 loop instructions (written below in C language): **for** ($\mu p_c = 1$; $\mu p_c > 0$; μp_c -= 0.1) **for** ($\mu p_m = 0$; $\mu p_m <= 1$; μp_m += 0.1)
- Basic idea is that if the fitness of the individual having the μp_m and μp_c set in this loop are several times larger than the best fitness of the same individual obtained with some already explored values for μp_m and μp_c, it makes no sense to compute the fitness of that individual with the next values of μp_m, because more than sure we will not have a too big drop in the fitness value.
- Actually we have applied the following strategy, which we have deduced after some experiments:
  - skip the next 3 values for μp_m if the fitness (current_individual, μp_m, μp_c) is 20 times bigger than the best fitness of the same individual discovered so far.
  - skip the next 2 values for pm if the fitness (current_individual, μp_m, μp_c) is 10 times bigger than the best fitness of the same individual discovered so far.
  - skip the next value for pm if the fitness (current_individual, μp_m, μp_c) is 5 times bigger than the best fitness of the same individual discovered so far.

Less than 1/3 combinations were actually explored in this way. Thus, the running time has increased, in average, only 30 times and not 100 times. This schema was deduced experimentally.

3.2 Fitness assignment

We deal with EAs at two different levels: a micro level representing the evolutionary algorithm encoded in a GA chromosome and a macro level GA, which evolves program-individuals. Macro level GA execution is bounded by the known rules for GAs (see [2]).

In order to compute the fitness of a GA individual we have to compute the quality of the evolved μEA encoded in that chromosome. For this purpose the μEA is run on the particular problem which is being solved.

Roughly speaking, the fitness of a macro individual equals the fitness of the best solution generated by the EA encoded in that GA chromosome. It is very likely that successive runs of the same μEA should generate completely different solutions, since the μEA encoded in a Mchromosome uses pseudo-random numbers. This stability problem is handled in a standard manner: the μEA encoded in a Mchromosome is executed (run) more times (100 μruns are in fact executed in all the experiments performed in order to evolve μEAs for function optimization) and the fitness of a Mchromosome is the average of the fitness of the best μchromosome from the last generation of the μEA over all the runs.

The optimization type (minimization/maximization) of the macro level GA 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).

3.3 The model used for evolving EAs

We use the steady-state evolutionary model [5] as an underlying mechanism for our macro GA implementation. The GA starts by creating a random population of MIndividuals (programs). 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 an offspring; the offspring is considered for mutation; the offspring *off* replaces the worst individual *worst* in the current population if *off* is better than *worst* (see the Algorithm 5).

3.3.1 Macro Initialization

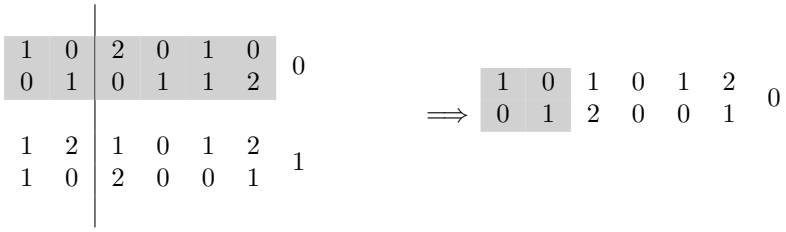
Each value of a gene from a macro chromosome will be initialize with a random value from $\{0, 1, 2\}$ set. The flag is randomly chosen from $\{0, 1\}$ set.

3.3.2 Macro Crossover

An example of a macro crossover taking two parents and generating an offspring is given below (the flag is randomly choosen from one of the parents):

Algorithm 5 The macro GA used for evolving μEAs

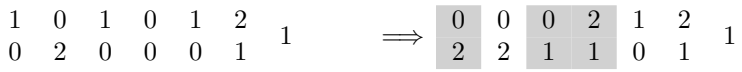
```
RandomlyInitialization(MPop);
Fitness(MPop);
for g=1 to MNoOfGenerations * MPopSize do
  p1 = MSelection(MPop);
  p2 = MSelection(MPop);
  MCrossover(p1, p2, off);
  MMutation(off);
  MFitness(off);                    // Run the EA encoded in the off in order to solve a particular problem
  if off is better than the worst individual (worst) from MPop then
    Replace worst with off
  end if
end for
```



The *MCrossover* operator is applied with a specific probability (Mp_c). If the conditions for crossover are not met, the offspring is the copy of the best parent.

3.3.3 Macro Mutation

Each gene of a macro chromosome is changed (with a given probability Mp_m) into another value (randomly generated) from $\{0, 1, 2\}$ set. The flag can also be mutated. A mutation example is given below:



3.4 Complexity

The complexity of the proposed method is bounded by the known rules for an EA.

The complexity of the evolved μEA can be described by the equation:

$$\mathcal{C}(\mu EA, \mu p_c, \mu p_m) = \mathcal{O}(NumberOf\mu Generations \times \mu PopSize \times \mathcal{C}_f) \tag{1}$$

where $\mathcal{C}_f$ is the function evaluation complexity.

The complexity of the GA used for evolving μEAs is given by the formula:

$$\begin{aligned}\mathcal{C}(GA) = \mathcal{O}(&NumberOfMGenerations \times MPopSize \times \mathcal{C}(\mu EA) \times NumberOf\mu Runs \times \\ &\times CardinalityOf\mu p_c ValueSet \times CardianlOf\mu p_m ValueSet)\end{aligned} \tag{2}$$

The $\mathcal{C}(\mu EA)$ factor was introduced here because we need to compute the fitness of each newly evolved micro program, which actually means that we have to repeatedly run the micro algorithm whose complexity is described by Equation (1).

The process of evolving algorithms is a complex task, which requires many computational resources. This is so because we need to assess the performance of each evolved EA by applying it to a particular problem (in our case, function optimization).

4 Test Functions

Ten test problems $f_1 - f_{10}$ (given in Table 4) 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 multi-modal (the number of the local minima increases exponentially with the problem dimension [28]). In all the experiments the definition domain of every function has ten dimensions ($n = 10$).

Table 4: Test functions used in our experimental study. The parameter n is the space dimension ($n = 10$ in our numerical experiments for evolving EAs) and f_{min} is the minimum value of the function.

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

5 Numerical Experiments

Several numerical experiments for evolving EAs are performed in this section. Several EAs for function optimization are evolved: some of them use the real encoding for the micro chromosomes and the other ones use the binary representation for the micro individuals. The evolved EAs are compared with standard EAs for several well-known optimization problems.

The section is structured as follows: EAs with real encoding are evolved in Section 5.1. Next Section, 5.2, contains a comparison between the evolved EA and three standard EAs (a generational, a steady state GA and an EP-like scheme). Next sections (5.3 and 5.4) are dedicated to the binary encoding: a set of experiments (similar to those from real encoding) are performed for binary representation.

5.1 Evolving EAs with real encoding

There is a wide range of Evolutionary Algorithms that can be evolved by using the proposed technique. Since the evolved μEA has to be compared with another algorithm (such as standard GA [2], steady state GA [22] or EP [28]), the chromosome representation and the parameters of the evolved μEA should be similar to those of the algorithm used for comparison. In the next experiment, we will compare the performance of the evolved μEA with the performance of several classical EAs and 50 individuals and 50 generations will be used for this purpose.

The solutions generated by the μEA are represented using real values [2]. Thus, each chromosome of the evolved μEA is a fixed-length array of real values. In what follows, we will denote the μEA evolved in this experiment by *Real-based evolved EA* or shortly *rEvoEA*.

A short description of real encoding and the corresponding genetic operators is given in Table 5.

The parameters of macro GA are given in Table 6. The macro chromosome contains the genetic operations performed in the μEA and their order. Note again that the length of a macro chromosome is equal to the number of individuals from the μpopulation (since each operation is a MChromosome creates a new μ individual).

This experiment serves our purpose of studying the performance of the evolved EA along a number of macro generations.

Each function from Table 4 was used as training problem. Consequently 10 different EAs have been evolved (see Table 7).

By looking at the evolved EAs we can deduce the followings:

- p_m and p_c generally depend on the problem being solved. This is not good and reduces the generalization ability. The evolved EAs could depend strictly on the problem being solved.
- Crossover followed by mutation is the predominant operation. In some cases (function f_7) no single mutation (the operation which has code 1 in our notation) is performed. This suggests that $CrossMutation$ is better in

Table 5: A short description of the real encoding.

Function to be optimized	f:[$MinX$, $MaxX$]$^n \rightarrow \Re$
Individual representation	$x = (x_1, x_2, \ldots, x_n)$, where $x_i \in [MinX, MaxX]$, for i=1, 2, ..., n
Initialization	Randomly
Selection	Binary tournament
Arithmetical Recombination with $\alpha = 0.5$	parent$_1$ – $x = (x_1, x_2, \ldots, x_n)$ parent$_2$ – $y = (y_1, y_2, \ldots, y_n)$ offspring – $o = (\frac{x_1+y_1}{2}, \frac{x_2+y_2}{2}, \ldots, \frac{x_n+y_n}{2})$
Gaussian Mutation with $\sigma = 0.01$	the parent – $x = (x_1, x_2, \ldots, x_n)$ the offspring – $o = (x_1 + N_1(0,\sigma), x_2 + N_2(0,\sigma), \ldots, x_n + N_n(0,\sigma))$[a]

[a] where $N_i(\mu, \sigma)$ is a function that generates a normally distributed one-dimensional random number with mean μ and standard deviation σ. The index i indicates that the random number is generated for each value of i

Table 6: Parameters of the macro GA algorithm used for Experiment 1.

Parameter	**Value**
Number of generations	100
Population size	100
Crossover type	One cutting point crossover
Crossover probability	0.8
Mutation type	Strong Mutation
Mutation probability	0.1
Selection	Binary tournament

most cases than single *Cross* or single *Mutation*. This also suggests that the algorithms driven by fixed step mutation are not as good as those performing both crossover and mutation are.

- If the μp_m is high (see functions f_5, f_6) more single mutations appear in the evolved EAs. This suggests a relationship between number of performed operations and the variation probabilities.
- *ReplaceWorst* is the major population altering operation for 8 problems out of 10. This suggests that a Steady-State like algorithm is better for most cases.
- *ReplaceCurrent* also appear in many of the evolved algorithms, which suggests that the strategy which implies offspring replacing the parent could lead to good results too.

Figure 1 presents the evolution of the performance of the best chromosome from the macro-GA population along with the number of macro generations. The average fitness was also given.

Figure 1 shows that the macro GA is able to evolve an EA in order to solve the optimization problems. The quality of the evolved EA improves as the search process advances (Figure 1 shows the improvements of the best *rEvoEA* along with the number of generations).

5.2 Comparing EAs with real encoding

This experiment serves our purpose of comparing the best evolved μEA obtained in the previous experiment with several well-known schemes (steady-state GA , a standard GA and an Evolutionary Programming-like scheme – see Algorithms 6, 7 and 8).

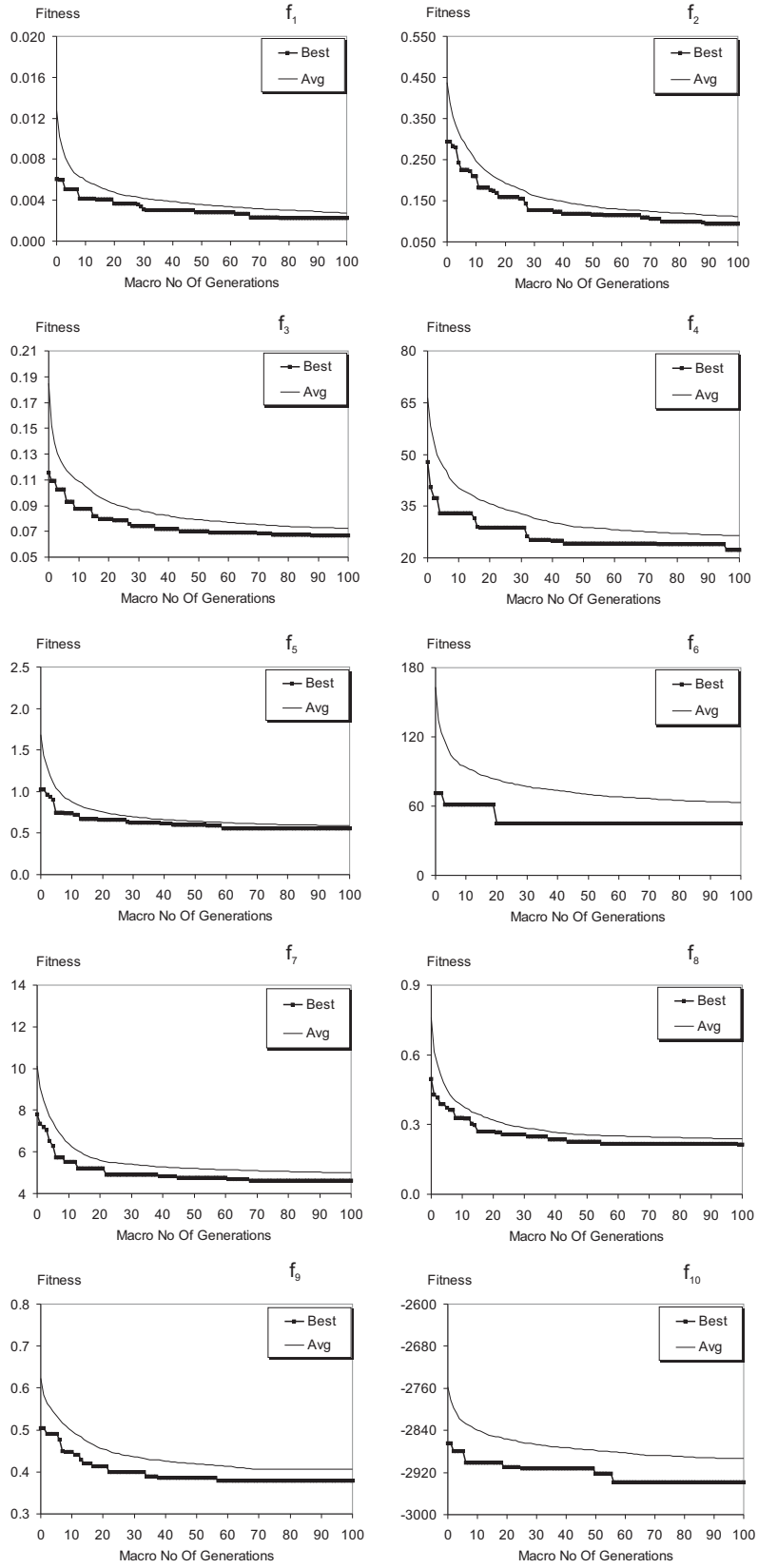


Figure 1: Relationship between the fitness of the best and average individual in each generation (of the macro GA) and the number of macro generations.

Table 7: Evolved EAs with real representation. For each chromosome we have given 2 lines: first line contains the code for chromosome creation operations (see Section 3.1.1) and the second line represents the population altering operations (see Section 3.1.2). The *next gen flag* was 1 in all experiments and we have not given it here anymore.

Fc	Operations	p_c	p_m
f_1	121220222022222222022222220222222222222222222012202 111111121112021212111122100011111111122211100111101	1.0	0.2
f_2	222222220022221222112222222222220222222212222222222 111111111021111211111101111001011111001001110200 10	1.0	0.2
f_3	220222220220222202022020222222022222202222222222222 111112101210220100010101222111110102000111110101 00	1.0	0.2
f_4	222222022222222222222222222222221222122222221222121 122102112112122221111101101111010111010010002001 00	1.0	1.0
f_5	202222022221220220212212212222222112222112122222122 111121011111101000112110201101012120011200011101 11	1.0	0.5
f_6	122121222222221202102222200011021112221222220111121 111200220220222000210120102111010201110000201021 00	0.8	0.4
f_7	022222222220002002222222220002000220222000222220202 202202202222120222012002202222220212020202002000 02	1.0	0.2
f_8	222222212222220122022022022222102222222220022222222 111111111112110111111210121111100011121111100010 0	1.0	0.3
f_9	022020022222220001022222222002002022220222020121 21 111111121111221011011111111010101002111010001111 10	1.0	0.2
f_{10}	122101221222122110120201122221011022010221002011 21 100000202200022222200210020212221222220222220022011	0.4	0.1

Algorithm 6 Steady-state GA - *ssGA*

```
Randomly_initialize_the_population μPop;
Evaluate_the_population μPop;
for g=1 to NoOfμGenerations × μPopSize do
  p1 = μSelection();
  p2 = μSelection();
  off = μCrossover(p1, p2, pc);
  off* = μMutation(off, pm);
  μFitness(off*);
  if off* is better then the worst individual worst from μPop then
    μReplace(off*);
  end if
end for
```

Algorithm 7 standard GA - *GA*

```
Randomly_initialize_the_population μPop;
Evaluate_the_population μPop;
for g=1 to NoOfμGenerations do
  Copy best to NextPop;
  for i=2 to μPopSize do
    p1 = μSelection();
    p2 = μSelection();
    off = μCrossover(p1, p2, pc);
    off* = μMutation(off, pm);
    μFitness(off*);
    μAppend(off*, NextPop);
  end for
  μPop = NextPop
end for
```

Algorithm 8 Evolutionary Programming - *EP*

```
Randomly_initialize_the_population μPop;
Evaluate_population μPop;
for g=1 to NoOfμGenerations do
  offspringPop = {};
  for i=1 to μPopSize do
    off* = μMutation(μPop[i], p_m);
    μFitness(off*);
    Add off to offspringPop;
  end for
  μPop = Choose best μPopSize individuals from μPop U offspringPop;
end for
```

In this experiment all involved algorithms use a real representation. Therefore, we will denote these algorithms by: *rSSGA*, *rGA*, *rEP* and *rEvoEA*.

In order to make a fair comparison we have to perform the same number of function evaluations in all compared algorithms. The μEA uses a population of 50 individuals which are evolved during 50 generations. Thus, for *rSSGA*, *rGA* and *rEP* we use the same number of function evaluations ($\mu NoGenerations \times \mu PopSize$). Both algorithms *rSSGA* and *rGA* have been run with all possible values for p_m and p_c optimized as described in section 3.1.5 and the best values are shown. *rEP* uses only mutation, thus only p_m is searched. For *rEvoEA* the values for μp_m and μp_c have been discovered in the previous experiment.

The values of real-encoding parameters are presented in Table 8.

Table 8: Parameters of a real-encoding EA.

Parameter	Value
Population size[a]	50
Individual encoding	fixed-length array of real values[b]
Number of generations	50
Selection	Binary Tournament
Crossover type	Uniform arithmetical crossover with $\alpha = 0.5$
Mutation	Gaussian mutation with $\sigma = 0.01$

[a]Note that the population size also represents the number of genes from the first part of a macro GA chromosome used in order to evolve the real-based μEA

[b]in fact, 10 values are used for each chromosome (the problem size was fixed to $n = 10$)

The results of these comparisons are presented in Table 9.

Table 9: Results of applying the Steady-state GA (*rSSGA*), the generational GA (*rGA*), the Evolutionary Programming (*rEP*) and the Evolved EA (*rEvoEA*) for the considered test problems. All the algorithms use real representation for their chromosomes. *Avg* stands for the mean best solution over 100 runs and *StdDev* stands for the standard deviation over 100 runs. Best results are written with bold font.

Fc	rSSGA					rGA					rEP				rEvoEA				
	p_c	p_m	Avg	±	StdDev	p_c	p_m	Avg	±	StdDev	p_m	Avg	±	StdDev	p_c	p_m	Avg	±	StdDev
f_1	1.0	0.2	**0.001**	±	0.003	1.0	0.2	0.018	±	0.022	1.0	25.844	±	13.793	1.0	0.2	0.002	±	0.013
f_2	0.9	0.2	0.103	±	0.097	1.0	0.2	0.864	±	0.666	1.0	2,316.550	±	1,056.630	1.0	0.2	**0.094**	±	0.126
f_3	0.9	0.2	0.073	±	0.209	1.0	0.1	0.210	±	0.220	1.0	17.844	±	6.763	1.0	0.2	**0.067**	±	0.043
f_4	1.0	1.0	23.680	±	12.083	1.0	0.9	63.474	±	42.915	1.0	3,206.240	±	1,592.620	1.0	1.0	**22.225**	±	16.700
f_5	1.0	0.4	0.573	±	0.125	1.0	0.3	0.963	±	0.787	0.9	23.547	±	7.593	1.0	0.5	**0.556**	±	0.370
f_6	0.8	0.3	89.728	±	4.807	1.0	0.6	103.483	±	108.714	1.0	621388.000	±	718962.000	0.8	0.4	**45.278**	±	365.192
f_7	1.0	0.1	7.450	±	4.807	1.0	0.2	4.812	±	1.977	1.0	40.782	±	9.513	1.0	0.2	**4.611**	±	2.824
f_8	1.0	0.2	0.243	±	0.140	1.0	0.2	0.832	±	0.445	1.0	14.512	±	2.678	1.0	0.3	**0.214**	±	0.275
f_9	0.8	0.1	0.384	±	0.227	1.0	0.2	0.739	±	0.108	1.0	20.031	±	11.123	1.0	0.2	**0.340**	±	0.177
f_{10}	0.1	0.7	-2767.1	±	396.731	0.1	0.8	-2744.9	±	323.010	0.9	-2857.1	±	287.955	0.4	0.1	**-2998.7**	±	367.357

Taking into account the average values, we can see that the evolved μEA performs significantly better than the steady-state GA in 9 cases out of 10 (see Table 9). We relate the results of *rEvoEA* only to the results of *rSSGA* because for all the problems the SSGA outperforms the other classic EAs (GA and EP).

In order to determine whether the differences between the evolved μEA and the steady-state GA are statistically significant, we use a *t*-test with 95% confidence. Only the average solution in each run has 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 the two-tailed t-test and of the F-test are given in Table 10.

Table 10: P-values (in scientific notation) of a t-test with 99 degrees of freedom in order to compare the *rEvoEA* with the classical GAs. All the algorithms use the real representation.

Function	*rEvoEA vs. rSSGA*	
	F-Test	t-Test
f_1	8.26E-20	5.58E-02
f_2	6.85E-02	1.44E-01
f_3	2.93E-21	2.49E-01
f_4	3.24E-02	2.66E-01
f_5	3.28E-12	1.70E-02
f_6	4.46E-79	2.40E-05
f_7	2.92E-04	1.23E-03
f_8	6.51E-06	3.02E-01
f_9	8.49E-02	4.85E-01
f_{10}	5.92E-01	1.58E-01

Table 10 shows that the difference between the *rEvoEA* and the *rSSGA* is statistically significant ($P < 0.05$) for the first 9 problems.

5.3 Evolving EAs with binary encoding

Several Evolutionary Algorithms using binary encoding for function optimization are evolved in this experiment. For training purposes we use again all the test problems described in Table 4. Each micro chromosome of the evolved EA is a fixed-length array of binary strings. For each dimension, we have used 30 bits for representation. The values of crossover and mutation probabilities (μp_c and μp_m) are optimised as described in section 3.1.5. A short description of binary encoding and the corresponding genetic operators is given in Table 11.

The parameters of the macro GA are the same as those used in Experiment 5.1 (the values for these parameters are presented in Table 6). The others micro parameters are the same as those used in Experiment 5.1: 50 μindividuals are evolved during 50 μgenerations.

We have obtained 10 different algorithms listed in Table 12. Based on the structure of the evolved EAs we can remark that:

- p_m and p_c are more stable (at least at this precision) compared with those from real encoding. We have noticed (results not shown here) that increasing the precision of p_m can lead to better results. However, searching p_m with a better precision will also increase the running time for evolving EAs (because all values for p_c and p_m are searched).
- Major variation operations are $Mutation$ and $CrossMutation$. This is different from real encoding where the major operation was $CrossMutation$. Mutation is more powerfully in binary encoding and this is why it appears more times than the real encoding case.
- Major population altering operations are $ReplaceWorst$ and $ReplaceCurrent$. This suggest again that Steady-State strategies are generally better than the others.

In this experiment we are again interested in observing the improvement trend of the evolved EA during the search process. The effectiveness of our approach can be seen in Figure 2. We depicted the fitness evolution for the best macro individual in a run and the average fitness. We can see in Figure 2 that the macro GA is able to evolve an EA for solving optimization problems. The quality of the evolved EA improves as the search process advances.

5.4 Comparing EAs with binary encoding

This experiment serves our purpose of comparing the Evolved EA (obtained in Experiment 5.3) with a steady state GA, a standard GA and an EP-like scheme. For a fair comparison, we must perform the same number of function evaluations.

Because the compared algorithms use the same binary representation of individuals, we will denote them by using the terms *bEvoEA*, *bSSGA*, *bGA* and *bEP*.

Both algorithms *bSSGA* and *bGA* have been run with all possible values for p_m and p_c optimized as described in section 3.1.5 and the best values are shown. *bEP* uses only mutation, thus only p_m is searched. For *bEvoEA* the values for μp_m and μp_c have been discovered in the previous experiment.

The results of this experiment are presented in Tables 14.

Table 11: A short description of binary encoding.

Function to be optimized	f:$[MinX, MaxX]^n$ $\rightarrow \Re$				
Individual representation	$x =$	$(x_{11}, x_{12}, \ldots, x_{1L})$ $(x_{21}, x_{22}, \ldots, x_{2L})$ $\ldots$ $(x_{n1}, x_{n2}, \ldots, x_{nL})$			
	where x_{ij} is a binary value				
Uniform Crossover		parent$_1$		parent$_2$	offspring
	$x =$	$(x_{11}, x_{12}, \ldots, x_{1L})$ $(x_{21}, x_{22}, \ldots, x_{2L})$ $\ldots$ $(x_{n1}, x_{n2}, \ldots, x_{nL})$	$y =$	$(y_{11}, y_{12}, \ldots, y_{1L})$ $(y_{21}, y_{22}, \ldots, y_{2L})$ $\ldots$ $(y_{n1}, y_{n2}, \ldots, y_{nL})$	$off =$ $(o_{11}, o_{12}, \ldots, o_{1L})$ $(o_{21}, o_{22}, \ldots, o_{2L})$ $\ldots$ $(o_{n1}, o_{n2}, \ldots, o_{nL})$
	where $P(o_{ij} = x_{ij}) = 0.5$ and $P(o_{ij} = y_{ij}) = 0.5$, $i = \overline{1,n}$ and $j = \overline{1,L}$				
Bit-wise Mutation		parent		offspring	
	$x =$	$(x_{11}, x_{12}, \ldots, x_{1L})$ $(x_{21}, x_{22}, \ldots, x_{2L})$ $\ldots$ $(x_{n1}, x_{n2}, \ldots, x_{nL})$	$off =$	$(o_{11}, o_{12}, \ldots, o_{1L})$ $(o_{21}, o_{22}, \ldots, o_{2L})$ $\ldots$ $(o_{n1}, o_{n2}, \ldots, o_{nL})$	
	where $o_{ij} = 1 - x_{ij}$, $i = \overline{1,n}$ and $j = \overline{1,L}$				

Table 12: Evolved EAs with binary representation. For each chromosome we have given 2 lines: first line contains the code for chromosome creation operations (see section 3.1.1) and the second line represents the population altering operations (see section 3.1.2). The *nextgen* flag was 1 in all experiments and we have not given it here anymore.

Function	Operations	p_c	p_m
f_1	12202010121111010121110101012002101022002121012020 22112112201122211011202222011010121002012220111000	1.0	0.1
f_2	12100110212021110200002200011221222012011201210202 21112101112212122112020102011211022212221122200021	1.0	0.1
f_3	22020220002202122020020112200001022001222002100202 00021121222101001221201202100212001001100012200020	1.0	0.1
f_4	20202101100010112000100112002220210122110212102120 20012011101220222112102121222210222101200101022112	0.9	0.1
f_5	11001212120220110100201120201010012010012000112200 12211222210222210021202011211010122121112221010101	1.0	0.1
f_6	01201010012020021011002221210110122020210022021110 11222012122212121222220211121122212112020000011002	1.0	0.1
f_7	02120012022102220020110220000010122020021210102021 22222120211222121100210211001110001202100212210222	1.0	0.1
f_8	11111202000111200112120101011020200201021210121020 21121101222112120101212120010112210112202000020210	1.0	0.1
f_9	12220012122112002112001200110010000202100220001101 12112121112112221212211122201022100001002210002122	1.0	0.1
f_{10}	02002122111221211102000012020212220022120022200022 01011010122211011002111001010222200212111121102122	1.0	0.1

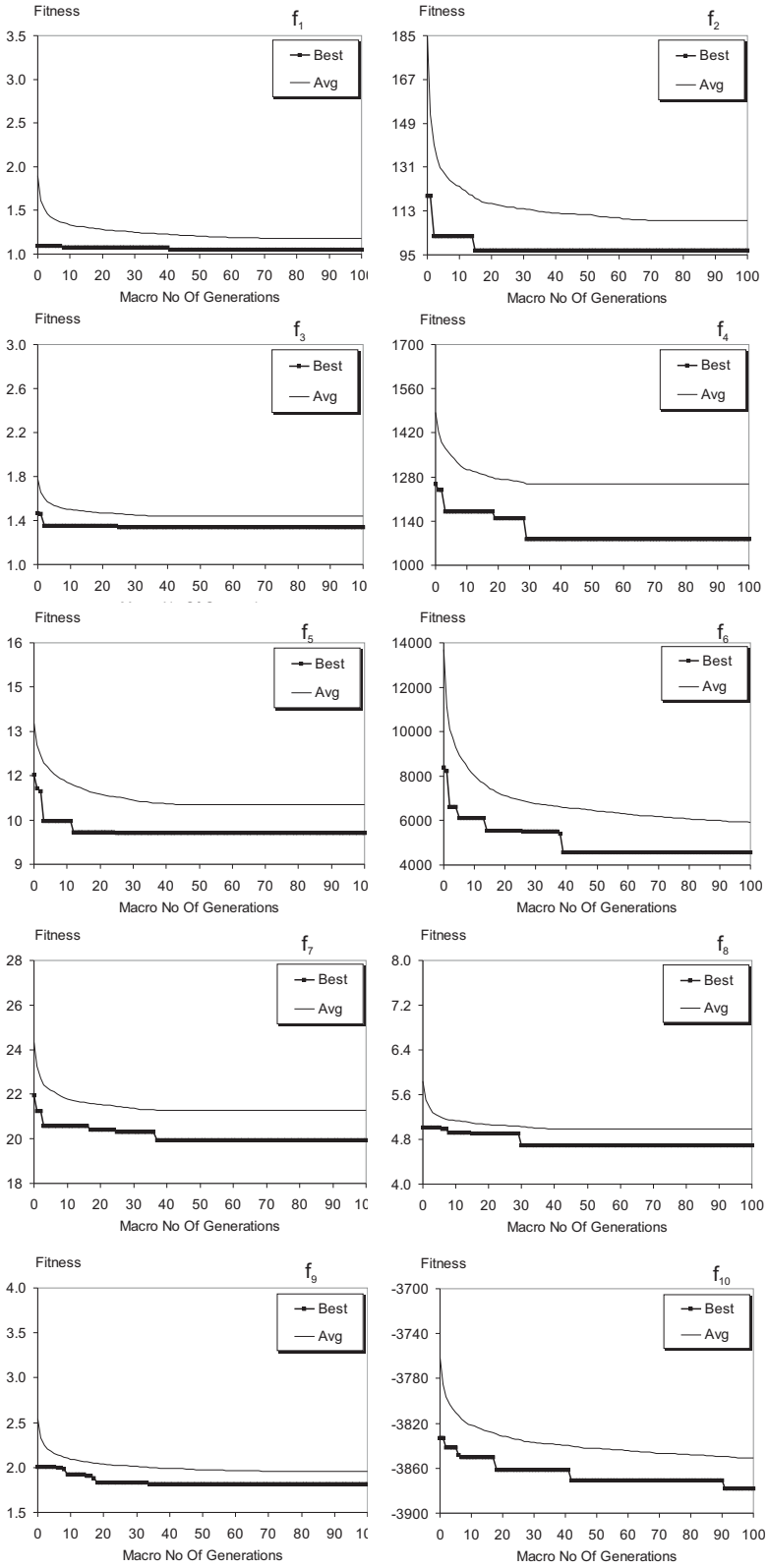


Figure 2: Relationship between the fitness of the best and average individual in each generation (of the macro GA) and the number of macro generations.

Table 13: Parameters of a binary-encoding EA.

Parameter	Value
Population size[a]	50
Individual encoding	fixed-length array of binary string
Number of generations	50
Selection	Binary Tournament
Crossover type	Uniform crossover
Mutation	Probabilistic bit-wise mutation

[a]Please note that the population size also represents the number of genes from the first part of a macro GA chromosome used in order to evolve binary-based EA

Table 14: Results of applying the Steady-state GA (*bSSGA*), the generational GA (*bGA*), the Evolutionary Programming (*bEP*) and the Evolved EA (*bEvoEA*) for the considered test problems. All the algorithms use a binary representation for their chromosomes. *Avg* stands for the mean best solution over 100 runs and *StdDev* stands for the standard deviation over 100 runs. Best results are written with bold font.

Fc	bSSGA					bGA					bEP				bEvoEA				
	p_c	p_m	Avg	±	StdDev	p_c	p_m	Avg	±	StdDev	p_m	Avg	±	StdDev	p_c	p_m	Avg	±	StdDev
f_1	0.1	0.9	11.52	±	4.32	0.1	0.1	12.16	±	5.85	0.1	6.94	±	2.27	1.0	0.1	**1.05**	±	1.10
f_2	0.1	0.9	1038.9	±	402	0.2	0.1	1329.4	±	693.5	0.1	635.3	±	244.3	1.0	0.1	**97**	±	75.3
f_3	1.0	1.0	4.87	±	2.67	0.1	0.1	6.81	±	2.11	0.1	5.18	±	0.98	1.0	0.1	**1.34**	±	0.80
f_4	0.1	0.1	1963.6	±	654.5	0.1	0.1	2824.2	±	984.0	0.1	1809.0	±	749.9	0.9	0.1	**1082.6**	±	855.2
f_5	0.1	0.9	19.40	±	4.03	0.1	0.1	23.34	±	5.66	0.1	17.66	±	4.95	1.0	0.1	**9.98**	±	4.51
f_6	0.4	0.1	7,516	±	6,519	0.1	0.1	225	±	198	0.1	75374	±	51226	1.0	0.1	**4556**	±	18230
f_7	1.0	1.0	35.01	±	12.20	0.1	0.1	43.08	±	8.92	0.1	29.59	±	5.35	1.0	0.1	**19.94**	±	7.64
f_8	1.0	1.0	10.49	±	2.32	0.2	0.1	11.29	±	1.77	0.1	9.47	±	1.16	1.0	0.1	**4.69**	±	1.39
f_9	0.2	0.1	10.38	±	3.28	0.1	0.1	11.78	±	4.55	0.1	6.46	±	2.01	1.0	0.1	**1.82**	±	0.90
f_{10}	0.8	0.1	-3392	±	191	0.3	0.1	-3235	±	235	0.1	-3384	±	165	1.0	0.1	**-3878**	±	192

Taking into account the average values, we can see that the evolved μEA performs significantly better than the steady-state GA in all cases (see Table 14). We relate the results of *bEvoEA* only to the results of *bSSGA* because for all the problems the SSGA outperforms the other classic EAs (GA and EP).

In order to determine whether the differences between the evolved EA and the other EAs are statistically significant, we use a t-test with 95% confidence. Only the average solution in each run has been taken into account for these tests. Before applying the t-test, an F-test has been used in order to determine whether the compared data have the same variance. The P-values of the two-tailed t-test and of the F-test are given in Table 15.

Table 15 shows that the difference between the evolved EA (*bEvoEA*) and the steady-state GA (*bSSGA*) is statistically significant ($P < 0.05$) for 7 test problems.

6 Scalability of the proposed approach

We are interested to find if our approach still performs well if we modify some of its parameters. Of great interest would be to see the outcome of the evolution for more generations and (very important) to see how the evolved algorithm performs on more difficult test functions (with more dimensions). One important parameter cannot be modified: the number of individuals in the evolved EA. This parameter has been fixed when we performed the evolution.

We start by running the algorithms on test problems with 30 dimensions. For the evolved EAs we keep both the structure and parameters (p_m and p_c) unchanged. For the other algorithms, we have tested 2 versions: first we kept fixed the p_m and p_c (as they were obtained in section 5.2 and 5.4) and secondly we have searched for specific values for 30 dimensions. Results are given in tables 16 - 19.

We can see the following:

- the scalability of real-encoded evolved EAs is not very good. In almost all cases the SS is better than the evolved one.
- the scalability of binary-encoded evolved EAs is a good one. In almost all cases the evolved EAs are better than the SS.

Next we focus our attention on increasing the number of generations. We run all algorithms for 100 generations (instead of only 50 as it was used during training purposes). The number of dimensions for the test functions is kept to 10. Results are show in tables 20 and 21. We can see that:

Table 15: P-values (in scientific notation) of a t-test with 99 degrees of freedom in order to compare the *rEvoEA* with the classical GAs. All the algorithms use the binary representation.

Function	*bEvoEA vs. bSSGA*	
	F-Test	t-Test
f_1	2.8E-75	3.7E-37
f_2	1.0E-68	7.0E-38
f_3	0.0000	0.0000
f_4	0.5129	1.0E-10
f_5	0.3653	0.0000
f_6	4.9E-40	3.5E-11
f_7	4.5E-05	1.5E-07
f_8	1.5E-13	5.0E-42
f_9	4.0E-49	1.4E-39
f_{10}	0.6792	5.0E-19

Table 16: Results obtained by applying the algorithms for 30 dimensions. The evolved EAs (including p_m and p_c) are kept unchanged. For the other algorithms the p_m and p_c are those discovered for 10 dimensions. Real encoding is used. Results are averaged over 100 runs. Best results are written with bold font.

Fc	rEvoEA			rSSGA			rGA			rEP		
	Avg	±	StdDev	Avg	±	StdDev	Avg	±	StdDev	Avg	±	StdDev
f_1	14.99	±	11.99	**6.59**	±	3.71	13.01	±	8.48	1086.43	±	180.52
f_2	389.68	±	388.69	**141.30**	±	151.74	337.54	±	200.60	34836.62	±	3495.30
f_3	8.33	±	3.19	**7.86**	±	3.22	10.61	±	2.80	2538.49	±	11096.59
f_4	2925.86	±	980.59	2485.63	±	686.81	**2441.67**	±	865.29	42413.58	±	12250.50
f_5	14.81	±	3.38	13.24	±	2.98	**11.51**	±	2.93	59.85	±	4.21
f_6	34914.21	±	35263.34	9397.77	±	14892.95	**5354.06**	±	4523.12	58804460.00	±	13703559.45
f_7	47.06	±	15.09	**32.07**	±	7.04	40.74	±	13.56	259.04	±	20.74
f_8	**5.67**	±	1.72	6.74	±	2.89	5.89	±	1.23	19.85	±	0.34
f_9	7.46	±	3.32	8.55	±	9.57	**3.52**	±	1.45	309.74	±	41.53
f_{10}	-6101.75	±	503.10	**-6843.04**	±	619.35	-6553.54	±	597.96	-6509.22	±	446.39

Table 17: Results obtained by applying the algorithms for 30 dimensions. The evolved EAs (including p_m and p_c) are kept unchanged. For the other algorithms the p_m and p_c are those discovered for 10 dimensions. Binary encoding is used. Results are averaged over 100 runs. Best results are written with bold font.

Fc	bEvoEA			bSSGA			bGA			bEP		
	Avg	±	StdDev	Avg	±	StdDev	Avg	±	StdDev	Avg	±	StdDev
f_1	**322.25**	±	106.67	984.54	±	136.04	863.47	±	143.00	669.90	±	78.67
f_2	**11041.06**	±	3021.06	31242.27	±	3073.32	26230.18	±	4285.16	19862.04	±	2313.98
f_3	**26.19**	±	6.13	85.28	±	25.45	92.17	±	41.03	63.53	±	6.37
f_4	36822.69	±	8173.60	58049.31	±	7984.54	42632.10	±	7860.73	**35264.27**	±	5549.15
f_5	**55.83**	±	4.29	72.80	±	3.46	64.50	±	4.36	58.78	±	3.60
f_6	**13468551**	±	4934092	84253970	±	15906015	47689732	±	12510278	29936396	±	6077607
f_7	**153.06**	±	29.43	242.41	±	19.83	282.94	±	23.83	248.00	±	13.95
f_8	**15.06**	±	1.09	19.06	±	0.31	18.69	±	0.46	17.80	±	0.50
f_9	**93.37**	±	25.08	143.58	±	28.79	229.12	±	43.75	182.38	±	24.13
f_{10}	-6113.42	±	367.47	-4153.17	±	374.73	-5794.54	±	399.63	**-6478.69**	±	354.87

Table 18: Results obtained by applying the algorithms for 30 dimensions. The evolved EAs (including p_m and p_c) are kept unchanged. For the other algorithms the p_m and p_c are are searched again (over 30 dimensions). Real encoding is used. Results are averaged over 100 runs. Best results are written with bold font.

Fc	rEvoEA			rSSGA			rGA			rEP		
	Avg	±	StdDev	Avg	±	StdDev	Avg	±	StdDev	Avg	±	StdDev
f_1	14.99	±	11.99	**2.89**	±	1.49	6.30	±	2.84	1055.56	±	193.69
f_2	389.68	±	388.69	**36.40**	±	13.74	106.59	±	35.57	33720.94	±	4597.59
f_3	8.33	±	3.19	**4.72**	±	1.22	6.34	±	1.21	384.21	±	1250.23
f_4	2925.86	±	980.59	2659.31	±	883.29	**2249.41**	±	805.31	42654.56	±	9039.93
f_5	14.81	±	3.38	10.50	±	2.97	**9.80**	±	2.43	59.48	±	3.84
f_6	34914.21	±	35263.34	**1624.82**	±	597.37	4340.87	±	3595.83	138793848.00	±	31283454.66
f_7	47.06	±	15.09	**31.31**	±	6.28	39.18	±	9.24	242.66	±	19.14
f_8	5.67	±	1.72	**3.25**	±	0.27	4.13	±	0.52	19.49	±	0.22
f_9	7.46	±	3.32	**1.36**	±	0.09	2.08	±	0.66	299.69	±	40.65
f_{10}	-6101.75	±	503.10	**-6917.50**	±	624.77	-6586.37	±	617.82	-6582.71	±	415.56

Table 19: Results obtained by applying the algorithms for 30 dimensions. The evolved EAs (including p_m and p_c) are kept unchanged. For the other algorithms the p_m and p_c are searched again (for 30 dimensions). Binary encoding is used. Results are averaged over 100 runs. Best results are written with bold font.

Fc	bEvoEA			bSSGA			bGA			bEP		
	Avg	±	StdDev	Avg	±	StdDev	Avg	±	StdDev	Avg	±	StdDev
f_1	**322.25**	±	106.67	466.88	±	84.59	840.94	±	169.53	649.81	±	89.05
f_2	**11041.06**	±	3021.06	15409.20	±	2346.90	25653.43	±	3819.48	20486.54	±	2692.91
f_3	**26.19**	±	6.13	42.88	±	4.85	101.87	±	82.52	64.26	±	5.77
f_4	36822.69	±	8173.60	**31756.59**	±	4916.20	43192.23	±	6975.87	36625.20	±	5498.46
f_5	55.83	±	4.29	**54.41**	±	3.53	65.36	±	3.20	56.94	±	4.01
f_6	**13468551**	±	4934092	18737664	±	4741336	44010717	±	16175841	29151356	±	6437101
f_7	**153.06**	±	29.43	225.60	±	15.85	285.71	±	23.80	248.07	±	13.89
f_8	**15.06**	±	1.09	16.99	±	0.64	18.67	±	0.53	17.88	±	0.51
f_9	**93.37**	±	25.08	141.41	±	22.71	235.88	±	39.94	190.00	±	19.24
f_{10}	-6113.42	±	367.47	**-7769.00**	±	407.01	-5733.18	±	500.40	-6463.09	±	325.94

- real-encoded evolved EAs scale reasonably well. In half of the cases, the evolved one is better than SS.
- binary-encoded evolved EAs scale better. In 7 cases (out of 10) the evolved EAs are better than the SS.

7 Comparing the techniques for evolving EAs

In this section we compare the results obtained with the current evolved EA and with several other evolved EAs which were reviewed in section 2 or proposed in [20].

Unfortunately, we cannot numerically compare our results with all previous attempts because the experimental conditions are too different. For instance the EA designed using the approach reviewed in section 2.1 contains 43 individuals (whereas the current approach contains 50).

The method that we can compare with is one based on MEP from section 2.2. The full-EAs (see section 2.2.1) were never too complex to compete with a human-designed one. The most complex design contains less than 100 function evaluations. The current approach performs 2500 function evaluations.

What was left was the kernel-generation with MEP (see section 2.2.2). Table 22 shows the results generated with the current method and that generated with the method from section 2.2.2. Both EAs use real representation with other parameters shown in Table 8. We can see that in all the cases the average values of EAs evolved with the current method are better than those generated with the method proposed in [19]. In all the cases the current method is better than the one proposed in [20].

Note that the comparison is still not a perfect one: in [19] only one EA was evolved (by using f_1 as training problem). In the current approach, we used each function as training, thus generating 10 different EAs.

8 Generalisation ability

The algorithms obtained in section 5 were trained on a single problem. Here we investigate the generalisation ability of these algorithms by testing them against all other test problems. The parameters of algorithms and test problems were set in section 5.

The results are presented in Tables 23 and 24 for real and binary representation, respectively.

Table 20: Results obtained by applying the algorithms for 100 generations. The evolved EAs (including p_m and p_c) are kept unchanged. For the other algorithms the p_m and p_c are those from section 5.2. Test problems with 10 dimensions are used. Real encoding is employed for all algorithms. Results are averaged over 100 runs. Best results are written with bold font.

Fc	rEvoEA			rSSGA			rGA			rEP		
	Avg	±	StdDev	Avg	±	StdDev	Avg	±	StdDev	Avg	±	StdDev
f_1	0.0003	±	0.0002	**0.0002**	±	0.0001	0.0042	±	0.0019	1.0679	±	1.0375
f_2	**0.0170**	±	0.0093	0.0199	±	0.0107	0.3347	±	0.1181	26.3593	±	20.1064
f_3	0.0230	±	0.0082	**0.0199**	±	0.0075	0.0524	±	0.0110	6.6526	±	5.0684
f_4	**6.7329**	±	2.6164	8.6817	±	2.2539	18.2645	±	8.5138	646.3822	±	477.3250
f_5	**0.4951**	±	0.0934	0.4960	±	0.1101	0.5640	±	0.0853	6.4036	±	4.6080
f_6	65.0598	±	165.4201	**29.5757**	±	61.9379	65.5823	±	71.4260	3933.4123	±	8064.9362
f_7	4.9385	±	1.7837	7.7634	±	3.2533	**4.6172**	±	1.7471	41.2092	±	9.4799
f_8	0.0911	±	0.0319	**0.0615**	±	0.0205	0.3800	±	0.0853	4.7604	±	2.8902
f_9	0.4254	±	0.1150	**0.0997**	±	0.0487	0.6392	±	0.1021	1.2266	±	0.2315
f_{10}	-2549.5004	±	348.9728	-2706.6062	±	299.8896	-2647.9632	±	324.4656	**-2854.1804**	±	282.0964

Table 21: Results obtained by applying the algorithms for 100 generations. The evolved EAs (including p_m and p_c) are kept unchanged. For the other algorithms the p_m and p_c are those from section 5.4. Test problems with 10 dimensions are used. Binary encoding is employed for all algorithms. Results are averaged over 100 runs. Best results are written with bold font.

Fc	bEvoEA			bSSGA			bGA			bEP		
	Avg	±	StdDev	Avg	±	StdDev	Avg	±	StdDev	Avg	±	StdDev
f_1	**0.49**	±	0.44	59.95	±	13.29	5.74	±	2.96	1.62	±	0.51
f_2	**42.79**	±	31.64	5429.71	±	1264.15	592.90	±	247.16	154.31	±	52.98
f_3	**1.10**	±	0.53	21.26	±	2.85	4.61	±	1.05	2.41	±	0.50
f_4	792.62	±	708.48	**649.25**	±	653.99	1708.31	±	710.35	900.25	±	591.38
f_5	**7.32**	±	3.15	36.52	±	4.81	17.43	±	5.06	8.86	±	2.58
f_6	1616.89	±	3201.15	**890.31**	±	860.07	60561.02	±	78123.60	6338.45	±	5589.27
f_7	**20.02**	±	5.68	72.73	±	6.14	32.42	±	7.36	20.78	±	3.64
f_8	**4.00**	±	0.84	16.99	±	1.01	9.31	±	1.64	6.01	±	0.84
f_9	1.43	±	0.45	**1.32**	±	0.15	5.48	±	2.80	2.35	±	0.56
f_{10}	-3678.03	±	140.03	**-3904.68**	±	147.47	-3467.97	±	216.72	-3794.73	±	144.09

Table 22: Comparing three different evolved EAs with real encoding. Best results are written with bold font.

Fc	rEvoEA			EvolvedMEP [19]			Evolved EA [20]		
	Avg	±	StdDev	Avg	±	StdDev	Avg	±	StdDev
f_1	**0.00002**	±	0.00002	0.12800	±	0.40100	0.11800	±	0.11300
f_2	**0.00212**	±	0.00214	27.10000	±	39.00000	9.11000	±	8.58000
f_3	**0.00448**	±	0.00338	0.25000	±	0.36400	0.30800	±	0.31300
f_4	**0.51300**	±	0.33700	38.30000	±	58.20000	35.40000	±	26.50000
f_5	**0.42000**	±	0.12000	3.00000	±	2.31000	5.01000	±	1.38000
f_6	**39.80000**	±	133.00000	340.00000	±	993.00000	394.00000	±	663.00000
f_7	**1.67000**	±	1.21000	1.86000	±	1.58000	4.42000	±	3.01000
f_8	**0.02490**	±	0.01350	2.78000	±	1.76000	2.89000	±	1.32000
f_9	**0.23000**	±	0.09290	0.50900	±	0.34000	0.75300	±	0.28800
f_{10}	**-1510.00000**	±	210.00000	-1010.00000	±	172.00000	-1480.00000	±	508.00000

Table 23: Generalisation ability of the evolved EAs with real representation. The results are average over 50 runs. Best results are written with bold font.

Train *vs.* test	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
f_1	0.00030	0.14443	0.00145	0.13874	61.29386	0.01037	0.00035	19.67479	125.05401	-1423.41734
f_2	**0.00026**	**0.13066**	**0.00090**	**0.12025**	62.10773	**0.00975**	**0.00026**	19.72137	134.33953	-1410.41334
f_3	0.00027	0.15211	0.00122	0.12681	60.92056	0.01169	0.00037	19.68650	128.44055	**-1430.91641**
f_4	0.01618	6.60418	0.05961	6.19439	62.05084	0.58681	0.01593	19.70636	124.68388	-1345.73950
f_5	0.00199	0.87686	0.00893	0.77453	60.88235	0.06880	0.00211	19.64946	125.86749	-1364.99698
f_6	0.00141	0.72836	0.00571	0.55545	61.40751	0.05490	0.00146	19.71340	127.47548	-1321.11750
f_7	0.00126	0.61725	0.00509	0.61667	62.72897	0.05378	0.00137	19.75456	131.64630	-1363.81673
f_8	0.00050	0.24229	0.00238	0.20781	61.38994	0.01881	0.00054	19.68233	125.34925	-1405.54314
f_9	0.00031	0.17288	0.00115	0.15795	**59.76257**	0.01150	0.00033	**19.52938**	**124.30715**	-1380.77104
f_{10}	0.00860	56.01909	0.02596	16.35453	61.90936	0.69741	0.01526	19.67687	126.92789	-1360.43448

Table 24: Generalisation ability of the evolved EAs with binary representation. The results are average over 50 runs. Best results are written with bold font.

Train *vs.* test	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
f_1	0.4144	**1512.4206**	1.6963	221.9874	**59.0461**	19.2511	0.5194	**19.4762**	120.3164	-1343.6610
f_2	0.4040	4893.9964	**1.4591**	257.8150	61.1152	18.1578	0.5801	19.6270	127.4783	-1376.2629
f_3	0.5910	3766.7472	1.7395	259.0765	62.6041	18.2763	0.6168	19.6613	125.5378	-1366.9854
f_4	0.4473	1705.4259	1.9056	239.7223	60.7780	20.3249	0.5312	19.4822	**118.9940**	**-1381.2192**
f_5	0.5059	2056.3808	1.9089	259.7646	61.4554	16.7482	0.4530	19.5351	123.0909	-1379.2464
f_6	0.4193	2785.8952	1.9151	250.1698	61.4332	15.7238	0.6394	19.6449	126.6345	-1337.9972
f_7	0.5716	3854.3255	1.9175	287.1736	61.6140	21.0453	0.6135	19.5125	124.3114	-1283.2350
f_8	**0.3592**	1968.5952	1.5953	**171.8165**	61.9482	**14.5021**	**0.3792**	19.6418	129.6790	-1312.2322
f_9	0.5723	2963.2558	1.5722	212.1393	62.6898	21.8680	0.6128	19.6566	130.0736	-1269.6699
f_{10}	0.5762	1968.6867	1.9035	258.9360	61.5949	19.4772	0.5026	19.6981	129.8182	-1325.6952

The results form Tables 23 and 24 show that the evolved EAs can compete with standard schemes whose results were already given in section 5. None of the evolved algorithms has seriously failed on any test functions. However, none of the evolved algorithm is better than the standard schemes for all problems. For real encoding the algorithm trained on f_2 is better than Steady State on 6 functions (out of 10).

9 Non-numerical comparison between evolved algorithms

Here we make a non-numerical comparison among various techniques used in order to generate EAs by evolutionary means (the proposed one and those described in section 2).

Several criteria could be considered in order to compare the mentioned techniques:

- Taking into account the number of EAs encoded in a macro chromosome we can observe that each LGP or GA chromosome encodes a single EA. This is different from the MEP approach where each chromosome encodes multiple EAs. Even in that case, the mechanism of encoding multiple solutions in a single MEP chromosome is only used for evolving full EAs. When evolving the EA pattern, only the best solution encoded in the MEP chromosome is taken into account.

- The complexity of the evolved algorithms is quite different. In the MEP-based approach used for evolving EAs [18] an entire EA is evolved. It is a simple sequence of genetic instructions (no generations, no loops for filling a population). In the LGP-based approach [15], as well as in the MEP-based approach (but only for evolving some pattern involved in an EA) [19] and the GA-based model (the current one) most of the EA parts are kept intact: we have generations and we have a loop which fills the new population etc.

- Another important observation regards the genetic operators employed by the evolved evolutionary algorithms. In the GA-based model, the genetic operators only manipulate individuals from the current population. The same mechanism is involved in the models for evolving EAs by using LGP [15] and for evolving some EA patterns by using MEP [19], but it is different from the methods proposed in [18] whose operators manipulate individuals taken from the set of all individuals created since the beginning of the search process. In other words, the nature-inspired mechanism of "generations" is present in the current model and in those proposed in [15] and [19].

- Initialization – in the MEP-based approach proposed in [18] we could use the initialization operator anywhere in the evolved algorithm. In the models proposed in [15], [19] and in the current approach this operator has been removed from the set of operators that could appear in an evolved EA. 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 [18] and in the LGP-based approach used in [15] the selection operator has 2 fixed parameters (e.g. *Selection 5, 2*). In the current approach (and in the model used for evolving EA patterns), a more general type of selection operator, which has no parameter, is used. This operator also selects two random individuals and the output is the better of them, but is not bounded anymore by some fixed positions in the population. Moreover, the model for evolving EA patterns is the following: the first gene or the first two genes from a MEP chromosome must encode a selection operator (this restriction is asked for the MEP chromosome representation and the terminal set composition - if the second operation is a mutation, than the first one must be a selection and if the third operation is a crossover, then the first two operations must be either two selections, or a selection and a mutation). These limitations were removed from the GA-based model: the crossover and the mutation operators have embedded the selection operators/operator as parameters.
- Crossover – the parents that participate to the recombination process are selected *on-fly*, and not at some previous steps (as in the case of the LGP-based model or the MEP-based approaches).
- Mutation – as with crossover, this operator perturbs an *on-fly* selected individual.
- Probabilities – the previous models have used fixed values for p_c and for p_m. The current model searches the best values of these parameters.
- Approximate running times were given. A perfect measure cannot be given since the algorithms have been run on different architectures ranging from 0.8GHz up to 2 GHz.

The similarities and the differences of the previously presented techniques are briefly presented in Table 25.

Table 25: A brief comparison of different models for evolving EAs.

Criteria	LGP	full MEP	pattern MEP	GA
How many EAs are encoded into a macro chromosome?	one	many	one	one
The micro population from which are manipulated the individuals considered for genetic operations	current	initial	current	current
Place of the *Initialization* process into the μEA	at the beginning	anywhere	at the beginning	at the beginning
The position of the μindividuals considered for *Selection*	fixed	fixed	random	random
The position of the μindividuals considered for *Crossover* and *Mutation*	fixed	fixed	fixed	random
μp_c and μp_m	fixed	fixed	fixed	optimsied
Running time	1 day	few hours	few minutes	1 week

10 Discussion over the evolution of EAs

Our primary purpose was to check if the evolution could help us to design some complex EAs. It is already well-known that evolution can generate new and unconventional designs. The antenna example is one of the best [31]. Another good example are digital circuits [32]. We tried to do a similar thing for EAs. We put many of the ingredients of the standard evolutionary schemes in the same pool and we tried to see if we can obtain something as good as the standard schemes or even better.

The running time of the macro EA is huge (several days for each training problem). Note that this huge running time was mainly due to the parameter sweep method used for finding good values for p_m and p_c from micro level. Without parameter sweep the running time would have been only few hours.

However, we did these experiments only to discover a good structure for an EA or to find some general guidelines for designing EAs for problems. After discovering that structure we do not need this power hunger algorithm anymore. The meta Genetic Programming should not be used a problem solver method. It should be only used for testing hypotheses or for generating better design for small cases from where we can easily generalize. From the results of our experiments we can establish a set of rules that can be used when implementing an Ea for a particular problem:

- Multiple values for p_m and p_c must be tried, because these values might depend on the actual problem being solved.
- Perform both crossover and mutation. Try to work with mutation only if the obtained results are not good.
- Use a steady-state scheme first. Try other schemes only if this one has failed in generating good results.

No Free Lunch theorems [3] say that is impossible to construct an algorithm which performs better than all others for all the problems. This is why we cannot hope to use our method for designing the best algorithm ever. From our numerical experiments, we have already seen that the generalization ability of the evolved algorithms is not perfect. There are some cases where the steady-state GA performed better than the evolved EA. Thus, our work does not contradict the implications of the No Free Lunch theorems.

Another natural question is whether the order of performing genetic operations is important. What if only the number of crossover and mutation is important and the order could be any? In order to prove that the order in which genetic operations are performed is very important we have made a new experiment. We have kept fixed the number of operators (the number of crossover, the number of mutations and so on) and we have generated 50 EAs having random orders for executing the operations. More than that, none of the randomly generated orders can compete with the evolved ones. This becomes obvious for binary encoding.

Other parameters were set similar to those from section 5. This experiment was performed for both chromosome representations: real and binary. The results obtained in this experiment are presented in Tables 26 and 27.

Table 26: Random order for real representation. *Best* and *Worst* stands for the best and worst solution, respectively, over 50 runs. *Avg* stands for the mean best solution over all the runs and *StdDev* stands for the corresponding standard deviation. Problem dimension was $n = 10$ for all test functions.

Fc	Random order - real			
	Best	Worst	Avg	StdDev
f_1	0.00030	0.00037	0.00033	0.00001
f_2	0.01888	0.02223	0.02083	0.00068
f_3	0.02418	0.02733	0.02571	0.00074
f_4	7.38097	8.11577	7.70165	0.18925
f_5	0.47830	0.49258	0.48597	0.00337
f_6	66.27220	103.97700	85.46476	7.00653
f_7	5.01692	5.51122	5.29459	0.09005
f_8	0.09352	0.10162	0.09801	0.00177
f_9	0.12537	0.13946	0.13127	0.00344
f_{10}	-2656.53000	-2605.21000	-2638.46940	11.14979

Table 27: Random order for binary representation. *Best* and *Worst* stands for the best and worst solution, respectively, over 50 runs. *Avg* stands for the mean best solution over all the runs and *StdDev* stands for the corresponding standard deviation. Problem dimension was $n = 10$ for all test functions.

Fc	Random order			
	Best	Worst	Avg	StdDev
f_1	0.3977	0.5351	0.4749	0.0273
f_2	39.6482	48.6247	44.0638	2.2151
f_3	0.8989	1.0794	1.0126	0.0427
f_4	798.3820	922.3890	869.5380	31.4641
f_5	6.7164	7.2041	6.9713	0.1162
f_6	1922.7800	2426.9100	2159.5730	116.4130
f_7	17.6990	20.6056	19.1313	0.5796
f_8	3.4865	3.9906	3.7493	0.1061
f_9	1.2956	1.4260	1.3716	0.0329
f_{10}	-3909.9800	-3838.1800	-3877.7140	14.2982

What we can see in Tables 26 and 27 is that order is important. There are big differences between the best and worst order.

11 Further work

There are some other questions about the Evolved Evolutionary Algorithms that should be answered. Some of them are:

- Which is the optimal selection procedure? In this paper we have used binary tournament, but other selection strategies could be of interest as well. For instance, q-tournament ($q > 2$) could be used instead of its binary counterpart. In this case, another parameter (q) should be evolved.

- Are all the genetic operators suitable for the particular problem that is being solved? A careful analysis regarding the genetic operators used should be performed in order to obtain the best results. The usefulness/uselessness of the genetic operators employed by the GA has already been subject to long debates. Due to the NFL theorems [4] we know that we cannot have "the best" genetic operator that performs best for all the problems. However, this is not our case, since our purpose is to find Evolutionary Algorithms for particular classes of problems.
- What is the optimal number of genetic instructions performed during a generation of the Evolved EA? In the experiments performed in this paper we have used fixed length GA chromosomes. In this way, we have forced a certain number of genetic operations to be performed during a generation of the Evolved EA. Further numerical experiments will be performed by using variable length GA chromosomes, with the expectation that this representation will find the optimal number of genetic instructions that have to be performed during a generation for a particular problem.

Our next experiments will be focused on:

- Using a network of computers for performing larger experiments. This could clarify some of the observations that we have made here.
- Using an extended set of training problems. This set should include problems from different fields, such as function optimization, symbolic regression, TSP, classification etc. Further efforts will be dedicated to the training of such algorithms, which could have increased generalization ability.
- Working with variable-size populations.
- Analyzing the relationship between the macro GA parameters (such as *Population Size*, *Chromosome Length*, *Mutation Probability* etc.) and the ability of the evolved EA to find optimal solutions.

12 Conclusions

In this paper, GAs have been used in order to design Evolutionary Algorithms. A detailed description of the proposed approach has been given, thus allowing researchers to apply the method in order to evolve Evolutionary Algorithms that could be used for solving problems in their fields of interest.

The proposed model has been used in order to evolve Evolutionary Algorithms for function optimization. The numerical experiments provided have emphasized 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.

References

[1] Holland, J.H.: Adaptation in natural artificial systems. University of Michigan Press, Ann Arbor (1975)

[2] Goldberg, D.E.: Genetic algorithms in search, optimization and machine learning. Addison Wesley (1989)

[3] Wolpert, D.H., Macready, W.G.: No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute (1995)

[4] Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation **1**(1) (1997) 67–82

[5] Syswerda, G.: A study of reproduction in generational and steady state genetic algorithms. In Rawlins, G.J.E., ed.: Proceedings of Foundations of Genetic Algorithms Conference, Morgan Kaufmann (1991) 94–101

[6] Ross, B.J.: Searching for search algorithms: Experiments in meta-search. Technical Report CS-02-23, Department of Computer Science, Brock University (2002)

[7] Angeline, P.J.: Adaptive and self-adaptive evolutionary computations. In Palaniswami, M., Attikiouzel, Y., Marks, R., Fogel, D., Fukuda, T., eds.: Computational Intelligence: A Dynamic Systems Perspective. IEEE Press, Piscataway, NJ (1995) 152–163

[8] Angeline, P.J.: Two self-adaptive crossover operators for genetic programming. In Angeline, P.J., Kinnear, Jr., K.E., eds.: Advances in Genetic Programming 2. MIT Press, Cambridge, MA, USA (1996) 89–110

[9] Edmonds, B.: Meta-genetic programming: Co-evolving the operators of variation. Elektrik **9**(1) (2001) 13–29

[10] Stephens, C.R., Olmedo, I.G., Vargas, J.M., Waelbroeck, H.: Self-adaptation in evolving systems. Artificial Life **4**(2) (1998) 183–201

[11] Teller, A.: Evolving programmers: The co-evolution of intelligent recombination operators. In Angeline, P., Kinnear, K., eds.: Advances in Genetic Programming II. MIT Press, Cambridge, MA (1996) 45–68

[12] Spector, L., Robinson, A.J.: Genetic programming and autoconstructive evolution with the push programming language. Genetic Programming and Evolvable Machines **3**(1) (2002) 7–40

[13] Kantschik, W., Dittrich, P., Brameier, M., Banzhaf, W.: Meta-evolution in graph Genetic Programming. In Poli, R., Nordin, P., Langdon, W.B., Fogarty, T.C., eds.: Genetic Programming, Proceedings of EuroGP'99. Volume 1598 of LNCS., Springer-Verlag (1999) 15–28

[14] Oltean, M.: Evolving evolutionary algorithms for function optimization. In (et al), K.C., ed.: The 7th Joint Conference on Information Sciences. Volume 1., Association for Intelligent Machinery (2003) 295–298

[15] Oltean, M.: Evolving evolutionary algorithms using linear genetic programming. Evolutionary Computation **13**(3) (2005) 387–410

[16] Brameier, M., Banzhaf, W.: A comparison of linear genetic programming and neural networks in medical data mining. IEEE-EC **5**(1) (2001) 17–26

[17] Nordin, P.: A compiling genetic programming system that directly manipulates the machine code. In Kinnear, Jr., K.E., ed.: Advances in Genetic Programming. MIT Press (1994) 311–332

[18] Oltean, M., Grosan, C.: Evolving evolutionary algorithms using multi expression programming. In Banzhaf, W., Christaller, T., Dittrich, P., Kim, J.T., Ziegler, J., eds.: Proceedings of European Conference on Artificial Life: Advances in Artificial Life. Volume 2801 of Lecture Notes in Artificial Intelligence., Springer (2003) 651–658

[19] Oltean, M.: Evolving evolutionary algorithms with patterns. Soft Computing **11**(6) (2007) 503–518

[20] Dioşan, L., Oltean, M.: Evolving evolutionary algorithms using evolutionary algorithms. In et al., D.T., ed.: GECCO 2007, Late breaking paper. (2007) 2442–2449

[21] Wolfgang, B.: Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications. Morgan Kaufmann (1998)

[22] Syswerda, G.: Uniform crossover in genetic algorithms. In: Proceedings of the third international conference on Genetic algorithms, San Francisco, CA, USA, Morgan Kaufmann (1989) 2–9

[23] Oltean, M., Grosan, C.: A comparison of several linear genetic programming techniques. Complex Systems **14**(4) (2004) 285–313

[24] Miller, J.F., Thomson, P.: Cartesian genetic programming. In Poli, R., Banzhaf, W., Langdon, W.B., Miller, J.F., Nordin, P., Fogarty, T.C., eds.: Proceedings of European Conference on Genetic Programming (EuroGP). Volume 1802 of Lecture Notes in Computer Science., Springer-Verlag (2000) 121–132

[25] Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press (1994)

[26] Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs. Springer (1992)

[27] Fogel, L.J., Owens, A.J., Walsh, M.J.: Artificial Intelligence through Simulated Evolution. John Wiley & Sons, New York (1966)

[28] Yao, X., Liu, Y., Lin, G.: Evolutionary Programming made faster. IEEE-EC **3**(2) (1999) 82

[29] Merz, P., Freisleben, B.: Genetic local search for the TSP: New results. In: Proceedings of the 1997 IEEE International Conference on Evolutionary Computation, IEEE Press (1997) 159–164

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

[31] Lohn, J.D., Linden, D.S., Hornby, G.S., Kraus, W.F., Rodriguez-Arroyo, A., Seufert, S.E.: Evolutionary design of an x-band antenna for nasa's space technology 5 mission. In: Evolvable Hardware, 2003. Proceedings. NASA/DoD Conference on. (2003) 155–163

[32] Miller, J.F., Job, D., Vassilev, V.K.: Principles in the evolutionary design of digital circuits—part I. Genetic Programming and Evolvable Machines **1**(1–2) (2000) 7–35