Evolving Evolutionary Algorithms using Evolutionary Algorithms

Laura Diosan and Mihai Oltean
Department of Computer Science, Babes-Bolyai University
Kogalniceanu 1, Cluj-Napoca, 400084, Romania
lauras, moltean@cs.ubbcluj.ro

ABSTRACT

A new model for automatic generation of Evolutionary Algorithms (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 evolved by using the considered model. Numerical experiments show that the evolved Evolutionary Algorithms perform similarly and sometimes even better than standard approaches for several well-known benchmarking problems.

Categories and Subject Descriptors

I.2.6 [**Learning**]; I.2.8 [**Problem Solving, Control Methods and Search**]

General Terms

Algorithms

Keywords

Genetic Algorithms, Evolutionary Algorithms, Function Optimization.

1. INTRODUCTION

Evolutionary Algorithms (EAs) [5, 6] are new and powerful tools used for solving difficult real-world problems. They have been developed in order to solve some real-world problems that the classical (mathematical) methods failed to successfully tackle. Many of these unsolved problems are (or could be turned into) optimization problems. The solving of an optimization problem means finding solutions that maximize or minimize a criteria function [5, 6, 21].

Many Evolutionary Algorithms have been proposed for dealing with optimization problems. Many solution representations and search operators have also been proposed and tested within a wide range of evolutionary models. There are several natural questions to be answered in all these evolutionary models: *What is the optimal population size? What is the optimal individual representation? What are the optimal probabilities for applying specific genetic operators? What is the optimal number of generations before halting the evolution?*

A breakthrough arose in 1995 when Wolpert and McReady unveiled their work on *No Free Lunch* (NFL) theorems for *Search* [19] and *Optimization* [20]. 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 being solved.) The magnitude of the NFL results stroke all the efforts for developing a universal black-box optimization algorithm capable of solving all the optimization problems in the best manner. Since we cannot build an EA able to solve best all problems we have to find other ways to construct algorithms that perform very well for some particular problems. One possibility (explored in this paper) is to let the evolution to discover the optimal design and parameters for the EA used for solving a particular problem.

In their attempt for solving problems, men delegated computers to develop algorithms capable of performing certain tasks. The most prominent effort in this direction is Genetic Programming (GP) [7], an evolutionary technique used for breeding a population of computer programs. Instead of evolving solutions for a particular problem instance, GP is mainly intended for discovering computer programs capable of solving particular classes of optimization problems.

There are many such approaches in literature concerning GP. Noticeable effort has been dedicated for evolving deterministic computer programs capable of solving specific problems such as symbolic regression [7], classification [3] etc.

Instead of evolving such deterministic computer programs, we will evolve a full-featured evolutionary algorithm (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 GA [17] 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 EA for solving a particular problem.

The rules employed by the evolved EAs during of 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) or it could be a steady state EA or a mixture of these two models.

This research was motivated by the need of answering several important questions concerning EAs. The most important question is "*Can Evolutionary Algorithms be automatically synthesized by using only the information about the problem being solved?*" [14]. And, if yes, *which are the genetic operators that have to be used in conjunction with an EA (for a given problem)?* Moreover, we are also interested to find the optimal (or near-optimal) sequence of genetic operations (selections, crossovers and mutations) to be performed during a generation of an EA for a particular problem. For instance, in a standard GA the sequence is the following: selection, recombination and mutation. However, how do we know that this scheme is the best for a particular problem (or problem instance)? And if this scheme is so good, which are the optimal values for crossover and mutation probabilities or for other parameters involved by the genetic operators? We better let the evolution to find the answer for us.

The paper is organized as follows. An overview of the related work in the field of evolving EAs is briefly presented in Section 2. The model used for evolving EAs is described in Section 3. Various numerical experiments are performed in Section 4. Several EAs for function optimization are evolved in that section. Further research directions are suggested in Section 5.

2. RELATED WORK

Several approaches evolve genetic operators for solving difficult problems [2, 4, 16, 18]. For instance, in his paper on Meta-Genetic Programming, Edmonds [4] used two populations: a standard GP population and a co-evolved population of operators that act on the main population. Note that all these approaches use a fixed evolutionary algorithm, which is not changed during the search.

Spector and Robinson [15] describes a language called Push, which supports a new, self-adaptive form of evolutionary computation called *auto constructive evolution*. An experiment was reported for symbolic regression problems. The conclusion was that "*Under most conditions the population quickly achieves reproductive competence and soon thereafter improves in fitness.*" [15].

Several attempts for evolving Evolutionary Algorithms using similar techniques were performed in the past. A generational EA was evolved [11] by using the Linear Genetic Programming (LGP) technique [3]. A non-generational EA was evolved [13] by using the Multi Expression Programming (MEP) technique [13]. MEP was used again for evolving only the kernel of an EA [12]. Numerical experiments have shown [12, 13] that the evolved EAs perform similarly and sometimes even better than the standard evolutionary approaches with which they are compared.

3. GAS FOR EVOLVING EAS

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

We deal with evolutionary algorithms at two levels: a macro EA and a micro EA. The macro EA is actually a GA manipulating a sequence of genetic operators and parameters employed by the micro EA during a generation.

In what follows we will denote the operations and the chromosomes involved into the macro level EA by using the *macro* or M prefix notation: MChromosome, MGeneration, MCrossover, MMutation and those involved into the micro level EA by using the *micro* or μ prefix notation: μChromosome, μGeneration, μMutation, μCrossover.

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 represents an evolutionary program which affects an array of individuals (the population involved into the μEA).

A GA chromosome (a macro-chromosome) used for evolving μEAs consists in two main parts: a part is used for specifying the operations that are performed into the μEA (micro crossover, micro mutation, micro append and micro replacement); the other chromosome part encodes the parameters of the μEA (the crossover and mutation probabilities). Both parts of the GA chromosome are subject to evolution.

The first part of the macro-chromosome represents the modifications performed into the population of the micro EA. These modifications can regard the operations performed for obtaining new micro individuals (crossover and mutation - called in what follows *chromosome-based operations*) or the operations performed in order to change the population composition (append a new individual into the population or replace a chromosome with other new chromosome - called *population-based operations*). The operations about population are performed each time when a new micro individual is obtained (by a crossover or by a mutation operation).

Each gene from the first part of a macro chromosome has two values: the first value is used for storing the *chromosome-based operations* and the second value of a gene stores the *population-based operations*.

Usually, three types of genetic operators may appear into an EA. These genetic operators are: *Selection* - selects the best solution among several already existing solutions, *Crossover* - recombines two existing solutions and *Mutation* - varies an existing solution. We will call these operations *micro Selection* (μSelection), *micro Crossover* (μ*Crossover*) and *micro Mutation* (μ*Mutation*) because they are performed into the μEA.

Because mutation operator and crossover operator need one and, respectively, two argument(s), we must perform these operations into an EA after minimum one or two selection(s). For avoiding these constraints, the selection procedure will be embedded (as a parameter) into the perturbation operators. More specifically, we have two major types of *chromosome-level instructions* in a modified GA chromosome. These instructions are:

off$_1$ = μ*Crossover* (μ*Selection()*, μ*Selection()*);
off$_2$ = μ*Mutation* (μ*Selection()*).

Remarks:

(i) The μ*Crossover* operator always generates a single offspring from two parents in our model. Crossover operators generating two offspring may be designed to fit our evolutionary model as well.

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

(iii) The μCrossover and the μMutation are problem 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: *μCrossover* – recombines two parents using one cutting point crossover [5], *μMutation* – bit-wise mutation [5].
- if we want to evolve a μEA for solving the TSP problem [9] we may use DPX as a μCrossover operator and 2-opt as a μMutation operator [8].

When a new μindividual is obtained by performing one of previous presented operations, we have to decide what to do with it. In our model we have two possibilities: to append it to the current population or to overwrite the worst individual in the current population.

Therefore, the second value of each gene from a macro chromosome will encode what will happen with the offspring obtained by recombination or mutation procedure. More specifically, we have two major types of *population-level instructions* in a macro chromosome: *μAppend(off)* and *μReplace(off)*.

The number of genes from the first part of a macro chromosome is equal to the number of individuals (n) from the evolved EA population. In this way, after performing all the instructions encoded into the macro chromosome, it is possible to have into the current micro population no more than n new individuals. Note that in the current population there are already n individuals from the previous generation.

The second part of a GA chromosome contains the values of different parameters used by the μEA (whose instructions are encoded into the first part of the GA individual). These parameters could be the micro crossover probability (μp_c), the micro mutation probability (μp_m). Therefore, the second part of the macro chromosome will have more genes, each of them containing a value for a parameter of the μEA.

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

	Gene values	
g_1	$o_1 = \mu Mutation(\mu Select())$	$\mu Append(o_1)$
g_2	$o_2 = \mu Crossover(\mu Select(),\ \mu Select())$	$\mu Replace(o_2)$
g_3	$o_3 = \mu Crossover(\mu Select(),\ \mu Select())$	$\mu Append(o_3)$
g_4	$o_4 = \mu Crossover(\mu Select(),\ \mu Select())$	$\mu Replace(o_4)$
g_5	$o_5 = \mu Mutation(\mu Select())$	$\mu Replace(o_5)$
g_6	$o_6 = \mu Crossover(\mu Select(),\ \mu Select())$	$\mu Append(o_6)$
g_7	μp_c	
g_8	μp_m	

Because each value of a gene from the first part of a macro chromosome ($g_1 \rightarrow g_6$ from the previous example) encodes one of two possible of micro genetic operations (*μCrossover* or *μMutation* for the first gene value and *μAppend* or *μReplace* for the second gene value), we can use some binary codes for these operations: 0 – *μCrossover*, 1 – *μMutation*, 0 – *μAppend*, 1 – *μReplace*.

Using this representation for a GA chromosome, we can apply some standard macro recombination and macro mutation operators. For instance, we can use the one-cutting point crossover [6] and a bit-wise mutation [6] (or other binary representation-based crossover and mutation) for the first part of a macro chromosome and an arithmetical crossover [5, 10] and a Gaussian mutation [5, 21] (or other genetic operators specific for real encoding) for the second part of a macro chromosome.

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. More than that, 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.

More than that, because after each *μMutation* or *μCrossover* it is possible to add a new individual into the population, we need to select the survivors of the current generation. They will form the next generation. Therefore, after all the micro genetic operations (encoded into the GA chromosome) are performed, we will sort the entire micro population and the best n micro individuals will form the new generation (where n is the size of the evolved micro population).

The μEA that corresponds to instructions encoded into chromosome C is given by the Algorithm 1:

Algorithm 1 GA chromosome-program – a μpopulation with 6 individuals

```
Randomly_initialize_the_population();{This operation is not encoded into the GA chromosome}
FitnessPopulation()
for g=1 to MaxGenerations do
   off_1 = μMutation(μSelection()); μAppend(off_1);
   off_2 = μCrossover(μSelection(),μSelection());
   μReplace(off_2);
   off_3 = μCrossover(μSelection(),μSelection());
   μAppend(off_3);
   off_4 = μCrossover(μSelection(),μSelection());
   μReplace(off_4);
   off_5 = μMutation(μSelection());
   μReplace(off_5);
   off_5 = μCrossover(μSelection(),μSelection());
   μAppend(off_5);
   FitnessPopulation()
   SortPopulation();
   Truncation();
end for
```

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

3.2 Fitness assignment

We deal with EAs at two different levels: a micro level representing the EA encoded into a GA chromosome and a macro level GA, which evolves program-individuals. Macro level GA execution is bounded by known rules for GAs (see [5]).

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 being solved.

Roughly speaking the fitness of a macro individual equals the fitness of the best solution generated by the μEA encoded into that GA chromosome. But since the μEA encoded into a macro chromosome uses pseudo-random numbers it is very likely that successive runs of the same μEA will generate completely different solutions. This stability problem is handled in a standard manner: the μEA is executed (run) more times (100 μruns are in fact executed in all the experiments performed for evolving μEAs for function optimization) and the fitness of a GA chromosome is the average of the fitness of the best micro chromosome (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 steady-state evolutionary model [17] as underlying mechanism for our macro GA implementation. The GA starts by creating a random population of macro individuals (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 2).

Algorithm 2 The macro GA used for evolving μEAs

```
Randomly_initialize_the_macro_population $MPop$;
Evaluate_the_macro_population $MPop$;
while not stop_condition do
    $p_1$ = $MSelection(MPop)$;
    $p_2$ = $MSelection(MPop)$;
    $MCrossover(p_1, p_2, off)$;
    $MMutation(off)$;
    $MFitness(off)$; {Run the EA encoded into the off for solving a particular problem}
    if off is better than worst individual worst from $MPop$ then
        Replace worst with off
    end if
end while
```

Macro Initialization Each part of a macro GA individual is generated according to its type:

- each gene from the first part of a macro chromosome will be initialize with a random binary value
- each gene from the second part of the macro chromosome will be a random real value from $[0, 1]$ range.

Macro Crossover An example of a macro crossover taking two parents and generating an offspring is given below:

$$\begin{array}{cc|ccc}
10 & 0010 & \mu p_c^1 & \mu p_m^1 \\
01 & 0110 & & \\
11 & 1011 & \mu p_c^2 & \mu p_m^2 \\
10 & 0001 & &
\end{array}
\Longrightarrow
\begin{array}{cccc}
10 & 1011 & \mu p_c^* & \mu p_m^* \\
01 & 0001 & &
\end{array}$$

For the first part of the macro chromosome, the cutting point is randomly chosen after the second position and the chromosomes exchange genetic material after this point. The genes from the second part of a macro offspring will be computed as a convex combination of corresponding parent genes:

$\mu p_c^* = \alpha \times \mu p_c^1 + (1 - \alpha) \times \mu p_c^2$
and
$\mu p_m^* = \alpha \times \mu p_m^1 + (1 - \alpha) \times \mu p_m^2$.

Macro Mutation Each gene from the first part is changed (with a given probability) into its complement. The genes from the second part will be affected by a random Gaussian variable:

$$\begin{array}{c} 101011 \\ 010001 \end{array} \quad \mu p_c^* \quad \mu p_m^* \Longrightarrow \begin{array}{cccc} 0 & 0 & 01 & 11 \\ 1 & 1 & 11 & 01 \end{array} \quad \mu p_c^+ \quad \mu p_m^+$$

with: $\mu p_c^+ = \mu p_c^* + N(0, 0.01)$ and $\mu p_m^+ = \mu p_m^* + N(0, 0.01)$, where $N(\mu, \sigma)$ is a function that generates a normally distributed one-dimensional random number with mean μ and standard deviation σ.

Note that *macro Crossover* and *macro Mutation* operators are not problem dependent. Their functionality is to combine and mutate some information from the macro chromosome: the macro offspring will encode a new order for applying the genetic operations into the micro EA and the values of some parameters involved into the evolved EA.

4. NUMERICAL EXPERIMENTS

In this section several numerical experiments for evolving EAs are performed. Two evolutionary algorithms for function optimization are evolved: one uses real encoding for the micro chromosomes and another one uses the binary representation for the micro individuals. For assessing the performance of the evolved EAs, several numerical experiments with two classical GAs for function optimization are also performed and the results are compared.

4.1 Test Functions

Ten test problems $f_1 - f_{10}$ (given in Table 1) are used in order to asses the performance of the evolved EAs. 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 [21]). In all experiments the definition domain of every function has five dimensions ($n = 5$).

4.2 Experimental Results

In this section we evolve an EA for function optimization and then we asses its performance. A comparison with several classical EAs is performed further in this section.

An important issue concerns the representation of the solutions evolved by the μEA (algorithm which is encoded into a macro GA chromosome) and the specific genetic operators used for this purpose. The solutions evolved by the μEA are represented using either real values or binary values [5]. Thus, each chromosome of the evolved μEA is a fixed-length array of real values (see Experiments 1 and 2) or binary values (see Experiments 3 and 4).

4.2.1 Experiment 1

An Evolutionary Algorithm for function optimization with real-encoding is evolved in this experiment.

There is a wide range of Evolutionary Algorithms that can be evolved by using the technique described above. Since the evolved μEA has to be compared with another algorithm (such as standard GA [5] or a steady state GA [17]), the chromosome representation and the parameters of the

Table 1: Test functions used in our experimental study. The parameter n is the space dimension ($n = 5$ in our numerical experiments) and f_{min} is the minimum value of the function.

Test function	Domain	f_{min}
$f_1(x) = \sum_{i=1}^{n} (i \cdot x_i^2)$.	[-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

evolved μEA should be similar to those of the algorithm used for comparison. In Experiment 2 we will compare the performance of the evolved μEA with the performance of several classical EAs using 50 individuals and 50 generations.

The solutions generated by the μEA are represented using real values [5]. 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*.

The parameters of the macro GA are given in Table 2. The macro chromosome contains the genetic operations (and their order) performed into the μEA and the crossover and mutation probabilities for the μEA. Moreover, the first part length of a macro chromosome gives the number of individuals from the micro population.

Table 2: The parameters of the macro GA algorithm used for Experiment 1.

Parameter		**Value**
Number of generations		100
Population size		100
Crossover type	I part	One cutting point
	II part	Uniform arithmetical
Crossover probability		0.8
Mutation type	I part	Strong
	II part	Gaussian
Mutation probability		0.1
Selection		Binary tournament

This experiment serves our purpose for studying the performance of the evolved EA taking into account various macro chromosome sizes (in fact only the number of genes from the first part of a chromosome is changed). Actually, we evolved more real encoding EAs taking into account different sizes for the population involved into the μEA.

In all tests performed in this experiment the μEA is evolved by using the first function f_1 as training problem. Figure 1 presents the evolution of the performance of the best chromosome from the macro-GA population along with the number of macro generations and for different number of genes into the first part of a macro chromosome.

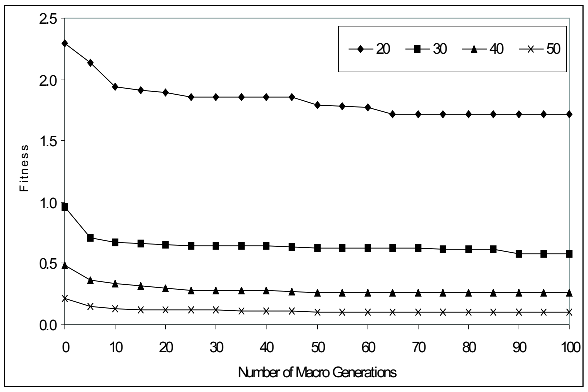

Figure 1: The relationship between the fitness of the best individual in each generation (of the macro GA) and the number of macro generations. Results are given on a logarithmic scale. Different μpopulation sizes are considered. Results are average over set of 10 runs, each set being performed for different population sizes (20, 30, 40 and 50 micro individuals).

Figure 1 shows 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. Moreover, we can observe in Figure 1 the improvements of the *RevoEA* along with the number of generations, but also along with the increasing of gene number from the first part of a macro chromosome.

4.2.2 Experiment 2

This experiment serves our purpose of comparing the best evolved μEA obtained in the previous experiment with two classical GAs: a standard generational GA with elitism *genGAe* [5] and with a steady-state GA [17] *ssGA*.

In this experiment both classical algorithms (*genGAe* and *ssGA*) and the evolved μEA (*evoEA*) use a real representation of the μChromosomes. Therefore we will denote these algorithms by: *RgenGAe*, *RssGA* and *RevoEA*.

For assessing the performance of the *RevoEA*, *RgenGAe* and *RssGA* we will use all the test functions given in Table 1. *RevoEA* uses the design evolved in the previous experiment. In order to make a fair comparison we have to perform the same number of function evaluations in both evolved EA and standard GAs. The μEA uses a population with 50 individuals, which are evolved during 50 generations. Therefore, for the *RgenGAe* we use the same number of μgenerations and the same μpopulation size and for the *RssGA* we used the same number of function evaluations ($\mu NoGenerations \times \mu PopSize$). Moreover, all the algorithms (*RevoEA*, *RgenGAe*, *RssGA*) perform the same crossover type and the same mutation type using the same probabilities. These probabilities are taken from the best macro chromosome obtained during evolution in the last generation of the macro GA (from the best GA-program evolved into the Experiment 1). The values of these parameters are more exactly presented in Table 4.

The results of these comparisons are presented in Table 3.

Table 4: The parameters of a real-encoding EA.

Parameter	Value
Population size[a]	50
Individual encoding	real fixed-length array
Number of generations	50
Crossover type	Uniform, with $\alpha = 0.5$
Crossover probability	0.9984
Mutation	Gaussian, with $\sigma = 0.01$
Mutation probability	0.7674
Selection	Binary Tournament

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

Taking into account the average values we can see in Table 3 that the evolved μEA significantly performs better than the standard generational GA in all cases.

Regarding the performance comparison of *RevoEA* vs. *RssGA* and taking into account the average values, we can see in Table 3 that the *RevoEA* significantly performs better than the *RssGA* in all the cases with only one exception (problem f_7).

Analyzing the differences between the performance of evolved μEA and those of the other GAs (*RgenGAe* and *RssGA*) we can see that the *RevoEA* is closer to the *RssGA* than to the *RgenGAe*.

4.2.3 Experiment 3

An Evolutionary Algorithm using binary encoding for function optimization is evolved in this experiment. For training purposes we use again the first test problem f_1 whose definition domain is $[-10, 10]$.

Each micro chromosome of the evolved EA is a fixed-length array of binary strings. By initialization, a point within the definition domain is randomly generated. Uniform crossover [1] and strong (probabilistic bit-wise) mutation [5] are used. In uniform crossover, every allele is exchanged between the two random individual pairs with a certain probability, known as the swapping probability (in general, this probability p_s is 0.5). For performing a bit-wise mutation we alter each gene independently with a probability p_m. In our model, the crossover probability (μp_c) and the mutation probability (μp_m) are stored into the second part of the macro chromosome.

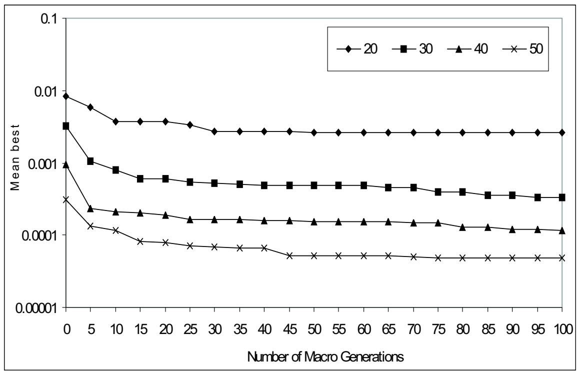

Figure 2: The relationship between the fitness of the best individual (a μEA with binary encoding) in each generation (of the macro GA) and the number of macro generations. Results are given on a logarithmic scale. Different μpopulation sizes are considered. Results are average over set of 10 runs, each set being performed for different population sizes (20, 30, 40 and 50 micro individuals).

The parameters of the macro GA are the same with those used in Experiment 1 (the values for these parameters are presented in Table 2). The micro EA representation is the major difference between the current experiment and the previous experiments: unlike the first experiment where the evolved μEA has used a real representation, now, the evolved micro algorithm is using a binary encoding of its chromosomes. Therefore, each micro chromosome is a binary string with 20 elements. The others micro parameters are the same with those used in Experiment 1: more μindividuals are evolved during 50 μgenerations.

This experiment serves our purpose of studying the performance of the evolved μEA based on a binary representation taking into account various macro chromosome sizes. Actually, we evolved more μEAs which use the binary representation for their chromosomes taking into account different sizes for the μpopulation. The evolved μEA is trained on the first function f_1 in all tests performed in this experiment.

We tested more values for the number of genes from the first part of a macro chromosome (or, with other words, more sizes for the micro population). In Figure 2 we present

Table 3: Results of applying the evolved μEA (*RevoEA*), the Standard generational GA with elitism (*RgenGAe*) and the Steady state GA (*RssGA*) for the considered test problems. The algorithms use the real representation for their chromosomes. *Avg* stands for the mean best solution over 100 runs and *StdDev* stands for standard deviation over 100 runs.

	RevoEA		*RgenGAe*		*RssGAe*	
	Avg	StdDev	Avg	StdDev	Avg	StdDev
f_1	1.18E-01	1.13E-01	2.32E-01	2.42E-01	1.41E-01	1.61E-01
f_2	9.11E+00	8.58E+00	2.61E+01	1.96E-01	2.20E+01	8.36E+00
f_3	3.08E-01	3.13E-01	1.13E+00	3.94E-01	8.89E-01	2.16E-01
f_4	3.54E+01	2.65E+01	6.30E+01	1.99E-01	4.45E+01	2.89E+01
f_5	5.01E+00	1.38E+00	7.70E+00	1.90E-02	5.50E+00	6.54E-01
f_6	3.94E+02	6.63E+02	4.55E+02	1.62E+01	4.33E+02	3.78E+02
f_7	4.42E+00	3.01E+00	7.57E+00	5.05E+00	2.11E+00	1.81E+00
f_8	2.89E+00	1.32E+00	6.20E+00	4.96E-01	6.31E+00	1.13E+00
f_9	7.53E-01	2.88E-01	1.13E+00	2.08E-01	1.58E+00	6.56E-01
f_{10}	-1.48E+03	5.08E+02	-8.81E+02	1.31E+02	-1.03E+03	2.04E+02

the fitness evolution for the best individual from the macro GA population along with the number of generations for different numbers of genes into the first part of the macro chromosome (20, 30, 40 and 50 genes or μindividuals). We can observe the improvements of the *BevoEA* along with the number of generations, but also along with the increasing of genes number from the first part of a macro chromosome. Note that we have modified just the length of the first part of a GA chromosome (the second part, reserved for μp_m and μp_c, rests the same).

4.2.4 Experiment 4

The next experiment serves our purpose of comparing the Evolved EA (obtained in Experiment 3) with a generational GA with elitism and with a steady state GA. Again, for a fair comparison, we must perform the same number of function evaluations. Having this in view we count how many new individuals are created during a generation of the evolved EA. Thus, *genGAe* and *ssGA* will use a main population of 50 individuals which are evolved during 50 generations (as the evolved μEA already used during the training stage on the first problem).

Moreover, because the compared algorithms are using the same binary representation of genetic individuals, we will denote them as *BevoEA, BgenGAe, BssGA*.

For assessing the performance of the evolved μEA we have use the test functions given in Table 1. Moreover, all the algorithms (*BevoEA*, *BgenGAe*, *BssGA*) involve the uniform crossover [1] and the bit-wise mutation [6] using the same probabilities. These probabilities are taken from the best macro chromosome obtained during the evolution in the last generation of the macro GA (the best GA-program evolved into the Experiment 3) - see Table 6.

The results of this experiment are presented in Table 5.

Taking into account the average values we can see in Table 5 that the evolved μEA significantly performs better than the standard generational GA in 8 cases.

Moreover, the solution from Table 5 indicate that the binary evolved μEA significantly performs better than the steady-state GA only for the third problem f_3, but if we considered the best solution found in one of these 100 runs, the *BevoEA* determines a better solution than the *BssGA* for six problems.

Table 6: The parameters of a binary-encoding EA.

Parameter	Value
Population size	50
Individual encoding	Binary string
Number of generations	50
Crossover type	Uniform
Crossover probability	0.812412
Mutation	Bit-wise
Mutation probability	0.0297436
Selection	Binary Tournament

5. CONCLUSIONS AND FURTHER WORK

In this paper, GAs have been used for designing EAs. A detailed description of the proposed approach has been given allowing researchers to apply the method for evolving EAs that could be used for solving problems in their fields of interest.

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

Some other questions should be answered about the Evolved Evolutionary Algorithms. Some of them are: *Which is the optimal selection procedure? Are all the instructions effective? Are all the genetic operators suitable for the particular problem being solved? What is the optimal number of genetic instructions performed during a generation of the Evolved EA?*

In order to evolve high quality EAs and assess their performance an extended set of training problems should be used. This set should include problems from different fields such as: function optimization, symbolic regression, TSP, classification etc. Further efforts will be dedicated to the training of such algorithm which should to have increased generalization ability.

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

Table 5: Results of applying the evolved μEA (*RevoEA*), the Standard generational GA with elitism (*RgenGAe*) and the Steady state GA (*RssGA*) for the considered test problems. The algorithms use the binary representation for their chromosomes. *Avg* stands for the mean best solution over 100 runs and *StdDev* stands for standard deviation over 100 runs.

	BevoEA		*BgenGAe*		*BssGAe*	
	Avg	StdDev	Avg	StdDev	Avg	StdDev
f_1	9.66E-05	1.70E-04	6.98E-02	7.20E-02	6.24E-05	6.79E-05
f_2	2.67E-02	9.12E-02	1.19E+01	1.23E+01	9.61E-03	1.23E-02
f_3	9.13E-03	7.20E-03	3.18E-01	1.71E-01	9.45E-03	5.11E-03
f_4	2.54E+02	4.33E+02	4.31E+01	5.14E+01	7.41E+01	2.10E+02
f_5	8.41E-01	9.44E-01	3.75E+00	1.94E+00	2.23E-01	1.53E-01
f_6	2.97E+02	5.97E+02	3.21E+02	3.99E+02	1.25E+02	3.65E+02
f_7	6.18E+00	3.47E+00	6.30E+00	2.62E+00	4.54E+00	2.38E+00
f_8	9.63E-01	1.12E+00	2.97E+00	8.02E-01	2.39E-01	5.80E-01
f_9	2.73E-01	1.61E-01	7.44E-01	2.21E-01	1.64E-01	7.73E-02
f_10	-1.96E+03	1.06E+02	-2.02E+03	7.06E+01	-2.01E+03	8.30E+01

Further numerical experiments will analyze 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 the optimal solutions.

6. REFERENCES

[1] D. H. Ackley. *A Connectionist Machine for Genetic Hillclimbing*. Kluwer Academic Publishers, Boston, MA, 1987.

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

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

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

[5] D. E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison Wesley, 1989.

[6] J. H. Holland. *Adaptation in natural artificial systems*. University of Michigan Press, Ann Arbor, 1975.

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

[8] N. Krasnogor. *Studies on the Theory and Design Space of Memetic Algorithms*. PhD thesis, University of the West of England, Bristol, 2002.

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

[10] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 1992.

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

[12] M. Oltean. Evolving evolutionary algorithms with patterns. *Soft Computing*, 2007.

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

[14] B. J. Ross. Searching for search algorithms: Experiments in meta-search. Technical Report CS-02-23, Department of Computer Science, Brock University, Dec. 06 2002.

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

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

[17] G. Syswerda. A study of reproduction in generational and steady state genetic algorithms. In G. J. E. Rawlins, editor, *Proceedings of Foundations of Genetic Algorithms Conference*, pages 94–101. Morgan Kaufmann, 1991.

[18] A. Teller. Evolving programmers: The co-evolution of intelligent recombination operators. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming II*, pages 45–68. MIT Press, Cambridge, MA, 1996.

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

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

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