Evolving Evolutionary Algorithms for Function Optimization

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

Abstract

In this paper, a generational Evolutionary Algorithm (EA) for function optimization is evolved using the Linear Genetic Programming (LGP) technique. Numerical experiments show that the evolved EA significantly outperforms a standard GA.

1. Introduction

Evolutionary Algorithms (EAs) [2] are nonconventional tools for solving difficult real-world problems. They have been developed under the pressure generated by the inability of the classical (mathematical) methods to solve difficult real-world problems. Many of these unsolved problems are (or could be transformed into) optimization problems.

Many EAs have been proposed for solving optimization problems. Many solutions representations and search operators have been proposed and tested within a wide-range of evolutionary models. There are several natural questions that have to be answered in all these evolutionary models: which is the optimal population size?, which is the optimal individual representation?, which are the optimal probabilities for applying specific genetic operators?, which is the optimal number of generations before halting the evolution?, etc. Wolpert and McReady proved [8] that all the black-box algorithms perform the same over the entire set of optimization problems. Those results [8] have stricken all the efforts for developing a universal black-box optimization algorithm able to solve best all the optimization problems. However, we can develop optimal or near-optimal algorithms for particular problems. An interesting approach consists of developing computer programs capable of writing other computer programs.

The most prominent effort in this direction is Genetic Programming (GP) [3], an evolutionary technique used for breeding a population of computer programs. Instead of evolving solutions for a particular problem instance, GP has been mainly intended for discovering computer programs able to solve particular classes of optimization problems.

In this paper, an EA for function optimization is evolved. Due to the special task which is solved here we will work with EAs at two levels: the first (macro) level consists of an Linear Genetic Programming (LGP) [1] model which is very suitable for evolving computer programs that may be easily translated into an imperative language (like ***C*** or ***Pascal***). The second (micro) level consists of the solution encoded in a chromosome by the GA on the first level. These solutions are EAs whose structure will be evolved by the LGP algorithm.

The evolved EA is a generational one. Note that this approach is very different from that proposed in [4] where nongenerational EAs are evolved using the Multi Expression Programming technique.

The paper is organized as follows. The LGP technique is described in section 2. In section 3, the model used for evolving EAs is presented. Test functions used for assessing the performance of the evolved evolutionary algorithm are presented in section 4. Several numerical experiments are performed in section 5.

2. LGP Technique

In this paper we use steady-state [6] as underlying mechanism for LGP. Steady-state LGP algorithm starts with a randomly chosen population of individuals. The following steps are repeated until a termination condition is reached: Two parents are selected (out of 4 individuals) using a binary tournament procedure and are recombined with a fixed crossover probability. By recombination of two parents two offspring are obtained. The offspring are mutated and the best of them replace the worst individual in the current population (if the offspring is better than the worst individual in the current population).

An LGP individual is represented by a 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. An example of LGP program is the following one:

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

A linear genetic program can be turned into a functional representation by successive replacements of variables starting with the last effective instruction [1].

3. LGP for Evolving EAs

In order to use LGP for evolving EAs we have to modify the structure of an LGP chromosome and to define a set of function symbols.

3.1. Individual Representation

Instead of working with registers, our LGP program will modify an array of individuals (the population). In what follows we denote by *Pop* the array of individuals (the population) which will be modified by the LGP program.

The set of function symbols consist of the genetic operators that may appear into an evolutionary algorithm. Usually there are 3 types of genetic operators that may appear into an EA: *Select* – that selects the best solution among several already existing solutions; *Crossover* – that recombine two existing solutions, and *Mutate* – that varies an existing solution.

These operators will act as possible function symbols that may appear into an LGP chromosome. Thus, each simple ***C*** instruction that has appeared into a standard LGP chromosome will be replaced by a more complex instruction containing genetic operators. More specific, in the modified LGP chromosomes we may have three types of instructions. These are:

Pop[*k*] = *Select* (*Pop*[*i*], *Pop*[*j*]);
```
// Select the best individual from those
// stored in Pop[i] and Pop[j] and keep the
// result in position k.
```

Pop[*k*] = *Crossover* (*Pop*[*i*], *Pop*[*j*]);
```
// Crossover the individuals stored in Pop[i]
// and Pop[j] and keep the result in Pop[k].
```

Pop[*k*] = *Mutate* (*Pop*[*i*]);
```
// Mutate the individual stored in position i
// and keep the result in position k
```

These statements will be considered as operations that are executed during an EA generation. Since our purpose is to evolve a **generational** EA we have to add a wrapper loop around the genetic operations that are executed during a generation. More than that, each EA usually starts with a random population of individuals. Thus, an LGP chromosome storing an EA is:

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

Remark: The initialization function and the **for** cycle are 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 evolutionary algorithm encoded into a LGP chromosome and a macro level GA, which evolves LGP individuals. Macro level GA execution is bounded by known rules for GAs (see [2]).

For computing the fitness of a LGP individual we have to compute the quality of the EA encoded in that chromosome. For this purpose the EA encoded into a LGP chromosome is run on the particular problem being solved.

Roughly speaking, the fitness of an LGP individual is equal to the fitness of the best solution generated (in the last generation) by the evolutionary algorithm encoded into that LGP chromosome. But, since the EA encoded into a LGP chromosome use pseudo-random numbers it is very possible as successive runs of the same EA to generate completely different solutions. This stability problem is handled in a standard manner: the EA encoded into a LGP chromosome is executed (run) more times (in fact 200 runs as many are executed in all the experiments performed in this paper) and the fitness of a LGP chromosome is the average of the fitness of the EA encoded in that chromosome over all runs.

3.3. The Model used for Evolving EAs

For evolving EAs we use the LGP algorithm described in section 2 of this paper. This method may be viewed as a training process of an algorithm for a given set of problems. For increasing the generalization ability (e.g.

the ability of the evolved EA to yield good results on new test problems), the problem set has been divided into three sub-sets, suggestively called training set, validation set and test set [5]. The training set consists of a difficult (multimodal) test function. Validation is performed using another difficult test function. The test set consists of 8 well-known benchmarking problems.

A method called *early stopping* is used to avoid overfitting of the population individuals to the particular training examples used (see [5]). This method consists of computing the test set performance for that chromosome which had minimum validation error during the search process. Using the early stopping technique will increase the generalization performance [5].

4. Test Problems

Ten test problems f_1-f_{10} (given in Table 1) are used to asses the performance of the evolved EA. Functions f_1-f_6 are unimodal test function. Functions f_7-f_{10} are highly multimodal (the number of local minima increases exponentially with the problem dimension [7]).

5. Numerical Experiments

In this section an EA for function optimization is evolved. Then the evolved EA is compared to a standard GA by using the test functions in Table 1.

For evolving an EA we use function f_7 as the training problem and function f_8 as the validation problem.

In order to establish the parameters of the evolved EA we have to compute the number of genetic operators that are performed during a generation of the Evolved EA. There is a wide range of EAs that can be evolved using the technique described above. Since, the evolved EA will be compared to another algorithm (such as a standard GA or an ES), the parameters of the evolved EA should be similar to the parameters of the algorithm used for comparison.

For instance, a standard GA uses a primary population of N individuals and an additional population (the new population) that stores the offspring [2]. Thus, the memory requirements for a standard GA is O(2*N). In each generation there will be 2*N *Selections*, N *Crossovers* and N *Mutations* (we assume here that only one offspring is obtained by crossover of two parents and each offspring is subject of mutation). Thus, the number of genetic operators (*Crossovers*, *Mutations* and *Selections*) is 4 * N in a standard GA. This algorithm is given below:

Test function	$f_{\min}$
$f_1(x) = \sum_{i=1}^{n} (i \cdot x_i^2).$	0
$f_2(x) = \sum_{i=1}^{n} x_i^2.$	0
$f_3(x) = \sum_{i=1}^{n} \lvert x_i \rvert + \prod_{i=1}^{n} \lvert x_i \rvert.$	0
$f_4(x) = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j^2 \right).$	0
$f_5(x) = \max_i \{x_i, 1 \le i \le n\}.$	0
$f_6(x) = \sum_{i=1}^{n-1} 100 \cdot (x_{i+1} - x_i^2)^2 + (1 - x_i)^2.$	0
$f_7(x) = 10 \cdot n + \sum_{i=1}^{n} (x_i^2 - 10 \cdot \cos(2 \cdot \pi \cdot x_i))$	0
$f_8(x) = -a \cdot e^{-b\sqrt{\frac{\sum_{i=1}^{n} x_i^2}{n}}} - e^{\frac{\sum \cos(c \cdot x_i)}{n}} + a + e.$	0
$f_9(x) = \frac{1}{4000} \cdot \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos(\frac{x_i}{\sqrt{i}}) + 1.$	0
$f_{10}(x) = \sum_{i=1}^{n} (-x_i \cdot \sin(\sqrt{\lvert x_i \rvert}))$	$-n \cdot 418.9829$

Table 1. Test functions used in our experimental study, where *n* is the space dimension (*n* = 5 in our numerical experiments) and $f_{\min}$ is the minimum value of the function. The definition domains are not given but they may be taken from [7].

```
Init_pop(old_pop);
for (i=0; i < NumberOfGenerations; i++){
  for (k = 0; k < PopSize; k++){
    p1 = Select();
    p1 = Select();
    o = Crossover(p1, p2);
    Mutate(o);
    new_pop[k] = o;
  }
  old_pop = new_pop;
}
```

Remark. The evolved EA has the same memory requirements and the same number of genetic operations as the standard GA described above. Our comparison is based on the memory requirements (i.e. the population size) and the number of genetic operators used by the during the search process. A better comparison could be made if we take into account the number of function evaluations performed during the search. Unfortunately this comparison cannot be realized since in our model we cannot control the number of function evaluations (this number is decided by the evolution). The whole number of genetic operations (*Selections* + *Crossovers* + *Mutations*) is the only parameter that can be controlled in our model.

The parameters of the LGP algorithm are: *Population Size* = 500; *Code Length* = 80 instructions; *Number Of Generations* = 100; *Crossover type* = *Uniform*; *Crossover prob*. = 0.7; *Mutation prob.* = 0.01.

The parameters of the evolved EA (similar to those used by a standard GA) are: *Population Size* = 40; *Individual Encoding* = Real; *Number of Generations* = 100; *Crossover type* = *Convex* with $\alpha = ½$; *Mutation* = *Gaussian* with $\sigma = 0.01$.

10 runs have been performed. In each run an EA yielding very good results has been evolved.

For assessing the performance of the evolved EA we will compare it with a standard GA. For this comparison we use the test functions given in Table 1. The parameters of the GA are similar to those used by the evolved EA. To see if the differences between the results obtained by the evolved EA and the results obtained by the GA are significant we use a T-test with 95 % confidence.

The results of this experiment are presented in Table 2.

#	**Evolved EA**		**Standard GA**		**T**
	Mean	*Dev*	*Mean*	*Dev*	
f_1	0.61	0.84	3.16	3.79	6E-4
f_2	273.70	235.77	817.12	699.26	1E-4
f_3	2.05	1.16	4.88	2.42	3E-7
f_4	340.27	348.37	639.22	486.78	8E-3
f_5	10.33	4.20	20.65	8.82	3E-7
f_6	10123	18645	208900	444827	1E-2
f_7	2.60	1.70	5.82	3.94	1E-4
f_8	7.59	2.50	10.89	2.76	9E-6
f_9	2.46	1.63	6.01	4.48	1E-4
f_{10}	-552.0	218.85	-288.34	200.55	6E-5

Table 2. Results of applying the Evolved EA and the Standard GA for the considered test problems. Dev stands for standard deviation. The values in the last column represent the P-values (in scientific notation) of a T test with 29 degree of freedom. Results are averaged over 30 runs.

From Table 2 it can be seen that the Evolved EA outperforms the standard GA on all the considered test problems. Moreover, the difference between the Evolved EA and the standard GA is statistically significant for all the test functions.

6. Conclusions and Further Work

In this paper, a technique for evolving Evolutionary Algorithms has been proposed. Using this technique, an EA for function optimization has been evolved. Numerical experiments show that the Evolved EA performs better that a standard Genetic Algorithm for several test functions.

Further efforts will be dedicated for evolving Evolutionary Algorithms for solving other difficult problems such as Traveling Salesman Problem, Quadratic Assignment Problem etc.

References

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

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

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

[4] M. Oltean and C. Groşan, “Evolving Evolutionary Algorithms using Multi Expression Programming”, European Conference on Artificial Life, 14-17 September 2003, Dortmund, Accepted.

[5] L. Prechelt, “PROBEN1 – A set of neural network problems and benchmarking rules”, Technical Report 21, University of Karlsruhe, 1994.

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

[7] X. Yao, Y. Liu and G. Lin, “Evolutionary programming made faster”, IEEE Transaction on Evolutionary Computation, Vol. 3(2), 82-102, 1999.

[8] D.H. Wolpert and W.G. McReady, “No Free Lunch Theorems for Optimization”, IEEE Transaction on Evolutionary Computation, Vol. 1, 67-82, 1997.