On the Evolution of Evolutionary Algorithms

Jorge Tavares[1], Penousal Machado[1,2], Amílcar Cardoso[1], Francisco B. Pereira[1,2], and Ernesto Costa[1]

[1] Centre for Informatics and Systems of the University of Coimbra, Pólo II - Pinhal de Marrocos, 3030 Coimbra, Portugal
[2] Instituto Superior de Engenharia de Coimbra, Quinta da Nora, 3030 Coimbra, Portugal
{jast, machado, amilcar, xico, ernesto}@dei.uc.pt

Abstract. In this paper we discuss the evolution of several components of a traditional Evolutionary Algorithm, such as genotype to phenotype mappings and genetic operators, presenting a formalized description of how this can be attained. We then focus on the evolution of mapping functions, for which we present experimental results achieved with a meta-evolutionary scheme.

1 Introduction

In order to achieve competitive results with an Evolutionary Computation (EC) approach, it is often required the development of problem specific operators, representations, and fine tuning of parameters. As a result, much of the EC research practice focuses on these aspects. To avoid these difficulties, researchers have proposed the evolution of these EC components. We believe that the evolution of parameters, operators and representations may contribute to performance improvements, give insight to the idiosyncrasies of particular problems, and alleviate the burden of EC researchers.

We propose the evolution of several components of an EC algorithm, and present the results attained in the evolution of the genotype-phenotype mapping procedure [1]. In section 2 we make a brief perusal of related research. Next, in section 3, we study different ways of evolving EC components, presenting a formalization. The experimental results are presented in section 4, which also comprises their analysis. Finally, we draw some overall conclusions and give pointers for future research.

2 Related Work

The area of Adaptive Evolutionary Computation (AEC) focuses on the evolution of specific parameters of EC algorithms. Angeline [2] makes a formal definition and classification of AEC, proposing three levels of adaptation: population-level, individual-level and component-level.

There are several AEC approaches that allow the dynamic resizing of the genotype, allowing its expansion and contraction according to environmental requirements. Angeline and Pollack [3] propose the co-evolution of: a high-level representational language suited to the environment; and of a dynamic GA where the genotype size varies. Also related to genotype-phenotype mapping, is the work of Altenberg [4] about the notion of "evolvability" - the ability of a population to produce variants fitter than previous existing one's. In [5, 6] Altenberg explores the concept of Genome Growth, a constructional selection method in which the degree of freedom of the representation is increased incrementally. This work is directly connected to the concepts presented by Dawkins in [7], where he clearly differentiates genetics, the study of the relationships between genotypes in successive generations, from embryology, the study of the relationships between genotype and phenotype in any one generation. This leads us to the concept of embryogeny, the process of growth that defines how a genotype is mapped onto a phenotype, and to the work of Bentley. In [8], the use of such growth processes within evolutionary systems is studied. Three main types of EC embryogenies are identified and explained: external, explicit and implicit. A comparative study between these different types, using an evolutionary design problem, is also presented.

Another aspect that can be subject of self adaptation is the set of genetic operators. The most straightforward approach is to try to select, from a pre-defined set, the most useful operator according to the environment. A paradigmatic example of this type of approach can be found in [9], which describes an AEC algorithm that dynamically chooses the crossover operator to apply, and adjusts the crossover probability. The self adaptation of crossover parameters is also described in [10], where the adaptation of parameters governing the selection of a crossover point, and amount of crossover operations is described.

The evolution of the genetic operators is the obvious next step. In [11], Teller describes how genetic operators can be evolved using PADO, a graph based GP system. In [12] GP is extended to a co-evolutionary model, allowing the co-evolution of candidate solutions and genetic operators. The evolving genetic operators are applied to the population of candidate solutions, and also to themselves. Another approach to the evolution of genetic operators is described in [13]. In this study, an additional level of recombination operators is introduced, which performs the recombination of a pool of operators. In [14] Spector and Robinson discuss how an autoconstructive evolutionary system can be attained using a language called Push. In the proposed system the individuals are responsible for the production of their own children.

3 Evolving EC Components

Historically EC is divided into four families namely: Evolution Strategies (ES); Evolutionary Programming (EP); Genetic Algorithms (GA); and Genetic Programming (GP). In spite of their differences they can all be seen as particular instances of the Generic Evolutionary Algorithm (GEA) presented in figure 1.

The first step is the creation of a random set of genotypes, $G(0)$. These genotypes are converted to phenotypes through the application of a mapping

```
$t \leftarrow 0$
$G(0) \leftarrow generate\_random(t)$
$P(0) \leftarrow map(G(0))$
$F(0) \leftarrow eval(P(0))$
while $stop\ criterion\ not\ met$ do
    $G'(t) \leftarrow sel(G(t), P(t), F(t))$
    $G''(t) \leftarrow op(G'(t))$
    $G(t+1) \leftarrow gen(G(t), G''(t))$
    $P(t+1) \leftarrow map(G(t+1))$
    $F(t+1) \leftarrow eval(P(t+1))$
    $t \leftarrow t+1$
end while
return $result$
```

Fig. 1. Generic Evolutionary Algorithm.

function (*map*). In most cases there isn't a clear distinction between genotype and phenotype, so this step is typically omitted. The next step consists on the evaluation of the individuals. This is performed at the phenotype level using a fitness function, *eval*.

The main evolutionary cycle follows. A set of parents is selected, using *sel*, followed by the application of genetic operators, *op*, which yields a new set of genotypes, $G''(t)$. The next steps consist in the generation of the phenotypes and their evaluation. The evolutionary cycle continues until some termination criterion is met.

3.1 Meta-Evolution

The most obvious approach to the evolution of EC components is the use of meta-evolution.

Let's assume that we are interested in evolving one of the components of EC, for instance the mapping function, and that we resort to GP to evolve populations of these functions. Each genotype, G_i^{map}, will be an encoding of a candidate mapping function; once expressed, via map^{map}, it will result in a phenotype, P_i^{map}. We need to define: sel^{map}, op^{map} and gen^{map}. Like map^{map} and $eval^{map}$ these functions will be static. Therefore, we can use standard GP selection, operators and replacement strategy.

We also need to evaluate the mapping functions, which implies developing a fitness function, $eval^{map}$. One possibility is to use a set of rules that specify the characteristics that are considered desirable in a mapping function, and assign fitness based on the accordance with those rules. We propose an alternative approach.

Since we are mainly interested in mapping functions that promote the discovery of good solutions for an original problem, we run, for each mapping function being evaluated, P_i^{map}, a lower level EC algorithm in which P_i^{map} is used as mapping function. Fitness is assigned using a function similar to the one presented in figure 2. Thus, the fitness of each phenotype is the result of a lower level EC. This result can indicate: the fitness of the best individual (at the lower level); the time necessary to reach the optimum; the average fitness of the last population; etc.

```
$eval^{map}(P^{map})$
    for $i = 1$ to $\#(P^{map})$ do
        $F_i^{map} \leftarrow GEA(P_i^{map})$
    endfor
    return $average(F^{map})$
```

Fig. 2. Meta-level fitness function.

It should be more or less obvious that we can employ the exact same strategy to evolve: *sel*, *op*, *gen* or *eval*. If we are evolving evaluation functions, the lower level EC is guided by P_i^{eval}. However, we are interested in a P_i^{eval} which allows the discovery of individuals which are fit accordingly to some original fitness function. As such, the return value of GEA should reflect its performance according to this original function.

The main problem of the architecture presented in this section is its high computational cost. In the next section we make a brief description of alternative approaches, which are, potentially, less time consuming.

3.2 Other Approaches

One of the alternatives is to use a standard EC approach in which each genotype, G_i is composed by:

- G_i^{sol} – the encoding of a candidate solution to some original problem, and
- G_i^{Δ} – where Δ is a tuple; each of its elements being an encoding of one of the following functions $\{map, sel, op, gen, eval\}$. Different elements correspond to different functions.

Likewise, the phenotype of an individual, P_i, is composed by a candidate solution, P_i^{sol}, and by a set of functions P_i^{Δ}. We name this approach *dual-evolution.*

Assuming that *map* is part of Δ, the phenotype is calculated in two steps. First, a static mapping function is applied to G_i^{Δ} yielding P_i^{Δ}. Then, the mapping function associated with the individual, P_i^{map}, is applied to G_i^{sol}, resulting in P_i^{sol}. Needless to say, if *map* is not a part of Δ, a static mapping function is used. If *op* is part of Δ, the generation of descendants must be changed. This can be done pretty much in the same way: apply conventional EC operators to G_i^{Δ}, and apply P_i^{op} to G_i^{sol}.

This approach is appropriate for operators that take as input a single genotype. Evolving operators for sexual reproduction is slightly more complicated. Let's assume that individuals i and j where selected for sexual reproduction, and, without loss of generality, that this operation yields two descendants i' and j'. $G_{i'}^{\Delta}$ and $G_{j'}^{\Delta}$ are created by applying standard operators. $G_{i'}^{sol}$ and $G_{j'}^{sol}$ are created through the calculation of $P_i^{op}(G_i^{sol}, G_j^{sol})$, and $P_j^{op}(G_j^{sol}, G_i^{sol})$, respectively.

The integration of other elements can pose some problems. Since both components, (Δ, sol), of the individual share a single fitness value, the evolution of *eval*, *sel* or *gen*, can easily lead to stagnation. For example, if an individual is responsible for its own evaluation, then all it must do to dominate the population is assign himself a extremely high value.

One option is to consider the original fitness function as a survival fitness, which determines the chances of reaching adult age, and thus of being a possible candidate for reproduction (*sel*) or part of the next generation (*gen*). When an individual i is selected for mating it uses its own fitness function, P_i^{eval}, to evaluate a set of mating candidates.

This approach can also be used for *sel*: if i is selected for mating, it uses P_i^{sel} to select its matting partners. In this case the input to P_i^{sel} could be the static fitness values of the other individuals; the fitness values i assigns to the other individuals (using P_i^{eval}); or both. We were unable to find an elegant way of integrating *gen* that doesn't lead to stagnation.

An alternative to dual-evolution is using a co-evolutionary approach, which can be considered as an intermediate level between dual and meta-evolution. In a co-evolutionary approach we would have one population composed by candidate solutions to a given problem. Each of the remaining populations would be composed by a specific EC component that we wish to evolve. This approach will be analyzed in a future paper.

4 Experimental Results

To test our ideas we decided to apply meta-evolution to evolve mapping functions. We use a two level meta-evolution scheme composed by a GP algorithm and a GA. At the higher level we have the GP algorithm, which is used to evolve the mapping functions. At the lower level we have the GA, whose task is finding the maximum value of a mathematical function. The goal is to evolve mappings that help the GA to accomplish its task.

The function being optimized by the GA, $f(x)$, is defined over the interval $[0, 1]$, and is the sum of $f_{peak}(x)$ and $f_{wave}(x)$, which are specified as follows:

$$f_{peak}(x) = max(0,\ |1 - 2|x - peak| - (1 - \frac{1}{r})| \times 0.1 \times r) \qquad (1)$$

$$f_{wave}(x) = cos(2\pi \times r \times (x - peak)) \qquad (2)$$

f_{wave} creates a sine wave with r repetitions in the $[0, 1]$ interval, returning values between -1 and 1. By adding f_{peak} we change the values of one of the repetitions, making it reach a maximum value of 1.1. In figure 3 we present a graph of $f(x)$. To increase the complexity of the problem, variable r is set to 100, which means that the wave function repeats itself 100 times in the $[0, 1]$ interval.

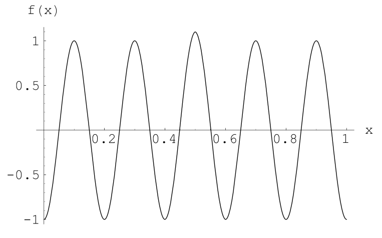

Fig. 3. Test function, $r = 5$, $peak = 0.5$

By changing the value of the variable *peak* we can change the coordinate of the maximum of $f(x)$. This ability to change the maximum is fundamental. If this value doesn't change the GP algorithm could find programs that output a constant x value corresponding to the maximum of $f(x)$. This isn't, obviously, the desired behavior. What we aim to achieve is a GP program that transforms the search space in a way that helps the GA to find the maximum value. Thus, for each value of x the GP programs should compute a new value, x'. The reorganization of the search space induced by the x to x' transformation should make the task of finding the optimum easier.

To ensure that it is possible to find a good mapping function we decided to design one by hand. We were able to develop the following function:

$$g_{optimal}(x) = \frac{x + floor(frac(x \times 10000) \times r)}{r} \tag{3}$$

Where $frac$ returns the fractional part. This function has the effect of folding the space r times, and then expanding it back to $[0, 1]$. By using this mapping function the topology of the search space is changed, resulting in a less convoluted fitness landscape. In figure 4 we present a chart of this mapping function, $g_{optimal}(x)$, and of the search space resulting from its application, $f(g_{optimal}(x))$.

To assign fitness, we run, for each GP individual, $mapping_j$ a GA. Each GA genotype, G_i^{sol}, is a binary string of size 50, encoding a value, $G_i'^{sol}$, in the $[0, 1]$ interval. The GP individual is used as mapping function for the GA. The value $G_i'^{sol}$ will be mapped to x_i' (thus, $x_i' = mapping_j(G_i'^{sol})$). x_i' is then used as the x coordinate for the function being optimized by the GA, f.

In order to get a good estimate of the quality of the mapping $functions$ we perform 30 runs of the GA for each GP individual. To prevent the specialization of the GP individuals, the value of *peak* is randomly chosen at the beginning of each GA run. The fitness the GP individual is equal to the average fitness of the best individual of the last population of the lower level GA.

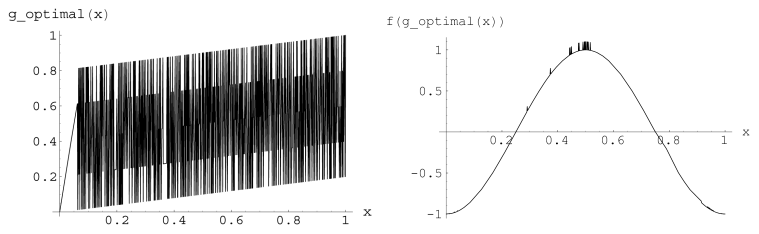

Fig. 4. On the left $g_{optimal}(x)$; on the right $f(g_{optimal}(x))$, $r = 5$.

We use the following function and terminal sets:

- *f-set* $= \{+, -, \%, \times, floor, frac\}$, where % is the protected division.
- *t-set* $= \{G_i'^{sol}, 1, 10, 100\}$, where $G_i'^{sol}$ is a variable holding the value of the GA genotype that is currently being mapped.

The settings for the GP algorithm were the following: Population size = 100; Number of generations 500; Swap-Tree crossover; Generate Random Tree mutation; Crossover probability = 70%; Mutation probability=20%; Maximum tree depth = 10.

The settings for the GA where the following: Population size = 100; Number of generations = 50; Two point crossover; Swap Mutation; Crossover probability = 70%; Mutation probability=$\{1\%, 2.5\%, 5\%\}$.

4.1 Analysis of the Results

The chart on figure 5 shows the evolution of the fitness of the best individual during the evolutionary process. An overview of the results shows that the GP algorithm is able to improve the fitness of the mapping functions. This indicates that the GP is able to find useful mappings for $f(x)$.

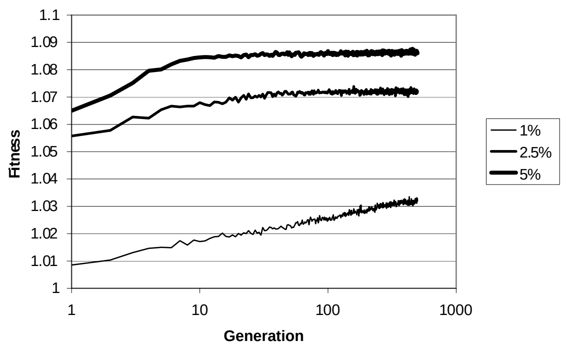

Fig. 5. Evolution of the fitness of the best individual. Averages of 30 independent runs

The main objective of our approach is to find a mapping function that consistently improves the performance of the GA algorithm. To evaluate the experimental results we need some reference points. Therefore, we conducted a series of experiments in which the mapping function was not subjected to evolution. In these tests we used the following static mapping functions: $g_{optimal}$, already described; and $g_{identity}$ with $g_{identity}(x) = G'^{sol}$. These results will be compared with the ones obtained using as mapping functions the best individuals of each GP run, $g_{evolved}$.

Table 1 shows the average fitness of the best individual of the last population of a traditional GA (Number of generations = 100) using as mapping functions $g_{identity}$, $g_{optimal}$ and $g_{evolved}$. The results are averages of 100 runs for the static mappings and of 3000 runs for the evolved mappings (100 runs per each evolved mapping).

Table 1. Average fitness of the best individual of the last population. The entries in bold indicate a statistically significant difference between these values and the corresponding $g_{identity}$ values ($\alpha = 0.01$).

Mutation	$g_{identity}$	$g_{optimal}$	$g_{evolved}$
1%	1.02713	**1.08763**	**1.04738**
2.5%	1.03442	**1.09731**	**1.07736**
5%	1.03995	**1.09964**	**1.09020**

As expected using $g_{optimal}$ considerably improves the performance of the algorithm, yielding averages close to the maximum attainable value, 1.1. Table 1 shows that the use of the evolved mapping functions significantly improves the performance of the GA. However, the results attained are inferior to the ones achieved using $g_{optimal}$. This difference diminishes as the mutation rate increases.

To get a better grasp of how the use of the evolved mapping functions alter the GA, we present, in figure 6, the evolution of the fitness of the best individual during the GA run.

For static mappings the fitness increases abruptly in the first generations, stagnating for the remainder of the run; with the evolved mappings the fitness increases steadily during the entire run. An analysis of the evolution of the average fitness of the GA populations gives insight to how the evolved mappings are improving the GA performance. As figure 7 shows, the use of evolved mappings decreases significantly the average fitness of the populations. These results, in combination with the ones presented in figure 6, indicate that the evolved mappings improve the performance of the GA by promoting phenotypic diversity, preventing the early stagnation of the GA runs.

The evolved mappings aren't similar to $g_{optimal}$, which can't be considered surprising. As is often the case when analyzing the results of a GP program, it isn't clear how the evolved mappings solve the problem of improving the GA

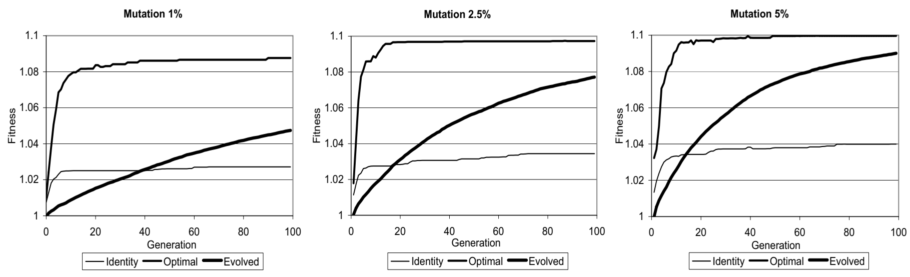

Fig. 6. Evolution of the fitness of the best individual.

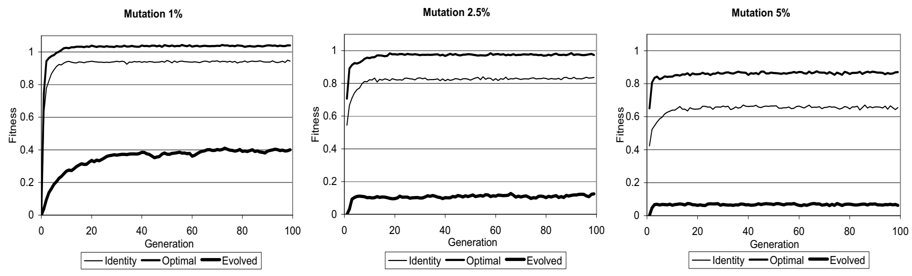

Fig. 7. Evolution of the average fitness of the GA populations.

performance. The charts suggest that reducing the number of GA generations used in the GP fitness assignment procedure, thus increasing the difficulty of the evolved mappings task, may lead to mappings closer to $g_{optimal}$. Additionally, taking into account the average of the GA populations when assigning a fitness value for the GP individuals, may also prove useful to achieve mappings closer to $g_{optimal}$.

5 Conclusions and Further Work

In this paper we discussed the extension of the canonical EC algorithm, presenting a formalization. The proposed changes involve the evolution of several components of EC which are typically static. The evolution of these components may prove useful in improving the EC performance, lessen the burden of researchers, and provide indications about the characteristics of the problems being solve.

The attained results are promising, and provide pointers for the improvement of the approach. Future research will include: making a wider set of experiments; applying the proposed approach to a different set of domains; and using dual-evolution and co-evolution to evolve EC components.

References

1. Banzhaf, W.: Genotype-phenotype-mapping and neutral variation – A case study in genetic programming. In Davidor, Y., Schwefel, H.P., Männer, R., eds.: Parallel Problem Solving from Nature III. Volume 866., Jerusalem, Springer-Verlag (1994) 322–332
2. Angeline, P.J.: Adaptive and self-adaptive evolutionary computations. In: Computational Intelligence: A Dynamic Systems Perspective. IEEE Press (1995)
3. Angeline, P.J., Pollack, J.B.: Coevolving high-level representations. In: Artificial Life III. Volume XVII of SFI Studies in the Sciences of Complexity. (1994) 55–71
4. Altenberg, L.: The evolution of evolvability in genetic programming. In Kinnear, K.E., ed.: Advances in Genetic Programming. MIT Press, Cambridge, MA (1994) 47–74
5. Altenberg, L.: Evolving better representations through selective genome growth. In: Proceedings of the 1st IEEE Conference on Evolutionary Computation. Part 1 (of 2), Piscataway N.J., IEEE (1994) 182–187
6. Altenberg, L.: Genome growth and the evolution of the genotype-phenotype map. In Banzhaf, W., Eeckman, F.H., eds.: Evolution as a Computational Process. Springer-Verlag, Berlin (1995) 205–259
7. Dawkins, R.: The evolution of evolvability. In Langton, C.G., ed.: Artificial Life, SFI Studies in the Sciences of Complexity. Volume VI., Addison-Wesley (1989) 201–220
8. Bentley, P., Kumar, S.: Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In Banzhaf, W., Daida, J., Eiben, A.E., Garzon, M.H., Honavar, V., Jakiela, M., Smith, R.E., eds.: Proceedings of the Genetic and Evolutionary Computation Conference. Volume 1., Orlando, Florida, USA, Morgan Kaufmann (1999) 35–43
9. Spears, W.M.: Adapting crossover in evolutionary algorithms. In: Proc. of the Fourth Annual Conference on Evolutionary Programming, Cambridge, MA, MIT Press (1995) 367–384
10. Angeline, P.J.: Two self-adaptive crossover operators for genetic programming. In: Advances in Genetic Programming 2. MIT Press, Cambridge, MA, USA (1996) 89–110
11. Teller, A.: Evolving programmers: The co-evolution of intelligent recombination operators. In Angeline, P.J., Kinnear, Jr., K.E., eds.: Advances in Genetic Programming 2. MIT Press, Cambridge, MA, USA (1996) 45–68
12. Edmonds, B.: Meta-genetic programming: Co-evolving the operators of variation. CPM Report 98-32, Centre for Policy Modelling, Manchester Metropolitan University, UK, Aytoun St., Manchester, M1 3GH. UK (1998)
13. Kantschik, W., Dittrich, P., Brameier, M., Banzhaf, W.: Meta-evolution in graph GP. In: Genetic Programming: Second European Workshop EuroGP'99, Springer (1999) 15–28
14. Spector, L., Robinson, A.: Genetic programming and autoconstructive evolution with the push programming language. Genetic Programming and Evolvable Machines **3** (2002) 7–40