Evolving Evolutionary Algorithms

Nuno Lourenço
CISUC, Department of Informatics Engineering
University of Coimbra, 3030
Coimbra, Portugal
naml@dei.uc.pt

Francisco B. Pereira
[1]CISUC, Department of Informatics Engineering
University of Coimbra, 3030
Coimbra, Portugal
[2]ISEC, Quinta da Nora, 3030
Coimbra, Portugal
xico@dei.uc.pt

Ernesto Costa
CISUC, Department of Informatics Engineering
University of Coimbra, 3030
Coimbra, Portugal
ernesto@dei.uc.pt

ABSTRACT

This paper proposes a Grammatical Evolution framework to the automatic design of Evolutionary Algorithms. We define a grammar that has the ability to combine components regularly appearing in existing evolutionary algorithms, aiming to achieve novel and fully functional optimization methods. The problem of the Royal Road Functions is used to assess the capacity of the framework to evolve algorithms. Results show that the computational system is able to evolve simple evolutionary algorithms that can effectively solve Royal Road instances. Moreover, some unusual design solutions, competitive with standard approaches, are also proposed by the grammatical evolution framework.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic Methods*

General Terms

Algorithms, Design

Keywords

Evolutionary Algorithms, Hyper-heurisitics, Automatic Evolution

1. INTRODUCTION

Evolutionary Algorithms (EAs) are computational methods loosely inspired by the principles of natural selection and genetics [7, 8]. When applied, EAs iteratively process a population of candidate solutions. Each iteration is typically composed by three processes: selection of the most promising individuals, creation of a new set of solutions by means of variation operators (usually crossover and mutation) and definition of survivors. In the last step, offspring solutions compete with their parents for a place in the population and a new iteration immediately starts. The process stops when a predetermined termination criterion is met (e.g., when a maximum number of iterations is achieved). The general structure of a simple EA is depicted in Alg. 1.

Algorithm 1 Evolutionary Algorithm

```
generate initial population
while termination condition not met do
  evaluate individuals
  select individuals
  apply variation operators
  define survivors to the next generation
end while
return best individual in the population
```

Despite these simple processing rules, EAs are robust search procedures able to quickly identify good quality solutions in hard problems. However, in many situations, the effectiveness of EAs can be greatly enhanced if its components are adjusted to the specific situation being addressed. Modifications are usually done manually and require a reasonable degree of expertise. Some concrete challenges that arise when aiming to maximize the effectiveness of an EA are:

1. How to represent/encode the candidate solutions processed by the EA?
2. How to select individuals for reproduction? What is the best selective pressure for a given optimization scenario?
3. How to define and apply variation operators? Which variations operators should be selected?
4. How to select the individuals that should survive?

In this paper we address these algorithmic design challenges by proposing a Grammatical Evolution (GE) [17, 13] computational framework that automatically evolves full-fledged EAs. The proposed grammar guides the selection and combination of common EA operators and it also defines the corresponding parameter settings. The application of this framework in the automatic design of EAs alleviates the task of deciding the most appropriate specification for a given problem. Moreover, this could introduce some benefits in terms of performance improvement.

Royal Road Functions (RR) [10] are selected as the target problem to test the ability of the grammar to evolve EAs.

GECCO'12 Companion, July 7–11, 2012, Philadelphia, PA, USA.

RR functions define optimization scenarios where population-based algorithms with crossover and mutation tend to outperform methods that do not rely on a combination of these variation operators. Moreover, the hardness of RR instances can be easily adjusted by changing the value of some parameters. It is therefore an appropriate environment to study the ability of a computational framework to automatically discover algorithmic structures. A careful analysis of the outcomes will allow us to gain insight into the evolved patterns and check if they resemble standard evolutionary algorithms or, on the contrary, contain unusual combinations of components.

The structure of the paper is as follows: Section 2 gives insight on recent work related to the automatic evolution of algorithms. Section 3 introduces Royal Road functions, whereas section 4 presents the GE framework built and the grammar used in the experiments. Section 5 presents an empirical study on the ability of the GE framework to evolve EAs. Section 6 gathers the main conclusions and suggests directions for future work.

2. RELATED WORK

The research area where an algorithm searches for algorithms to solve a certain problem is called Hyper-Heuristics (HH) [3, 16]. The aim of HH is to raise the level of generality at which algorithms can work. Most of the current meta-heuristics have to be manually designed when we want to apply them to a certain problem [3]. Hence they tend to become specific to the problem in question. In [4], Burke et al. propose a classification of HH into two classes: the first class corresponds to the automatic selection of existing heuristics. The second class corresponds to the automatic generation / design of new heuristics. Examples of both classes can be found in [4]. However the idea of having an EA that has the capacity of adapting itself during the evolutionary process has been studied for a while. In [1], Angeline focuses on the evolution of certain parameters that are part of EA techniques. For many tasks it is possible to dynamically adapt aspects of EA techniques to anticipate regularities in the environment and improve the search of new solutions. Moreover Angeline makes a formal definition of the different levels of adaptation: *population-level*, which consist on adapting, during the evolutionary process, the set of parameters that are shared by the population; *individual-level*, which consist on adapting the parameters related to the manipulation of the components that represent each individual; and *component-level*, which is the adaptation of the way each component of an individual behaves when a modification occurs. Following these ideas of automatically evolve parameters and operators of evolutionary algorithms, several proposals were made [2, 5, 6, 11, 12, 18].

In [2], Angeline introduced two adaptative crossover operations for GP: Selective Self-Adaptative Crossover (SSAC), and Self-Adaptative Multi-Crossover (SAMC). Both operators were designed to evolve crossover points. The difference between them is related to the adaptation-level where they work: the first works at the individual-level, whilst the second works at the component-level. In [6], Edmonds goes a step further and instead of evolving only parameters it co-evolves the genetic operators along with the candidate solutions to the problem being tackled. The mechanism that allows the construction of genetic operators is defined in such a way that it should guarantee the diversity of the candidate solutions. The evolved operators are applied to the population of candidate solutions and the population of operators itself. In [18] Tavares el al. proposed the evolution of the functions that make the mapping between the genotype and the phenotype. A good mapping function is essential in order to achieve good results, and in their work the authors propose a Genetic Programming algorithm that evolves a population of mapping functions, that are then used by an EA.

Several attempts to evolve complete evolutionary algorithms have been performed. In [12] Oltean et al. relied on Multi-Expression Programming (MEP) to evolve a non-generational EA. Moreover, in [11] Oltean extended the previous work and a generational EA was evolved using Linear Genetic Programming. In [5], Dioşan et al. tried to evolve a complete EA. They proposed to use EAs at two different levels: the first level, the macro, the algorithm has a fixed population size, fixed probability of variation operators. The second level, the micro, corresponds to the solutions encoded in the EA of the first level. A solution corresponds to an evolved sequence of genetic operations and their parameters, that will be used to solve a certain problem. Empirical results showed that the evolved algorithms perform similarly to the standard approaches to which they were compared [5, 11, 12]. In [15], Poli et al. evolved the main operator of a Particle Swarm Optimization, using Genetic Programming.

Recently, Tavares et al. proposed a GE framework to evolve Ant Colony Optimization Algorithms to the Traveling Salesman Problem [19].

3. ROYAL ROAD FUNCTIONS

The Royal Road functions (RR) were introduced by Holland, Mitchell and Forrest in 1992 [10], aiming to provide insight into the optimization behavior of EAs. These functions were designed in such a way that they can be solved by a simple population-based algorithm with crossover and mutation, but not by a hill-climber [9]. More precisely, the study was searching for answers to the following questions [10]:

1. Which problems are more suitable for EA's?
2. What is the effect of crossover on the EA's performance on different landscapes? How does it help to find good quality solutions?

A RR function takes a binary string as input, and produces a real value. The problem corresponds to a search task in which one wants to find strings with high fitness values. The RR can be described as a mapping: $F : \{0,1\}^n \rightarrow \mathbb{R}$, where n is the size of the binary string. Binary strings encoding solutions are composed by a sequence of 2^k non-overlapping contiguous regions, where k is a parameter that defines the instance of the RR. Each region is divided in two sections: a section of b bits called block, followed by a section of g bits called gap. Thus, a region is composed by $(b+g)$ bits. *A complete block* is defined when all bits of the block are set to 1. Furthermore, the RR functions are composed by *levels*. Levels correspond to contiguous sequences of 2^l complete blocks, where $0 \leq l \leq k$. Fig. 1 represents how the levels are defined for a RR instance with $k = 3$.

The standard instance has the following parameters: k = 4, b = 8, g = 7, which corresponds to a binary string of 240 bits, with 16 regions of 15 bits each [9].

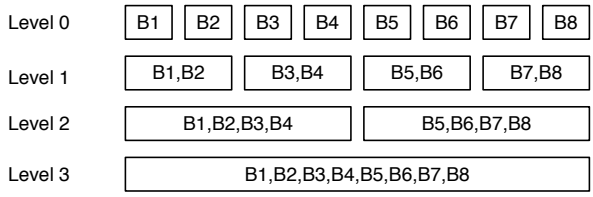

Figure 1: Levels example to a Royal Road Function with k=3. Bi, i={1..8} are the blocks.

3.1 Evaluation of the Royal Roads

The evaluation of the RR functions proceeds in 2 steps: the *PART* calculation and the *BONUS* calculation [9]. The parameters that are necessary to the fitness assessment are:

m^* Used in the PART fitness. It is the maximum number of 1's that a block may contain before being penalized. As an exception, if a block is complete it is not penalized;

v Used in the PART fitness. If the number of 1's of the block is m^* or less, it adds v, else it adds -v. If the block is complete it does not receive anything.

u^* Value added to the BONUS part by the first completed blocks at each level;

u Value added to the BONUS part by the second or subsequent completed block sets.

PART

This part of the evaluation considers each block individually. Each block receives a fitness score and in the end the individual block fitnesses are all summed to produce the PART contribution to the overall fitness. The fitness of each block is based only in the number of bits 1 that it contains. Every 1 up to a limit m^* adds a value v to the block's fitness. However if a block contains more than m^* 1s, but less than b 1s, it receives $-v$ for each 1 over the limit. Finally, if a block has all bits set to 1 it receives nothing from the PART calculation. Assuming $m^* = 4$ and $v = 0.02$ (Table 1) the PART fitness of Fig. 2 is: $PART = 0.00{+}0.08{+}0.00{+}(-0.04) = 0.04$.

BONUS

In the bonus part, we want to reward complete blocks and some combinations of complete blocks. In RR functions there are $k + 1$ distinct levels At all levels, the first sequence of completed blocks receives fitness u^*, and additional sequences of completed blocks receive u. Assuming $u^* = 1.0$ and $u = 0.2$ the BONUS fitness of Fig. 2 is: $BONUS = 1.0 + 0.2 = 1.2$.

The total fitness corresponds to the sum of the both PART and BONUS. Using the results that were calculated to the example used, the total fitness is: $PART + BONUS = 0.04 + 1.2 = 1.24$.

4. GRAMMATICAL EVOLUTION FRAMEWORK

GE is a recent bio-inspired technique proposed by Ryan et al. [17]. As in other Genetic Programming (GP) variants, the goal of GE is to evolve executable programs/algorithmic strategies that can exhibit good behavior when solving a given task. In early GP representations, programs were codified as syntax trees, and all the evolutionary operations were performed on the programs directly. On the contrary, in GE all operations are performed in binary strings. A mapping process is required to map the binary string to an executable program, via productions rules of a grammar. A grammar is a tuple $G = (N, T, S, P)$, where N is a non-empty set of non terminal symbols, T is a non-empty set of terminal symbols, S is an element of N called axiom, and P is a set of production rules of the form $A ::= \alpha$, with $A \in N$ and $\alpha \in (N \cup T)^*$. N and T are disjoint. Each grammar G defines a language $L(G)$, that is the set of all sequences of terminal symbols that can be derived from the axiom, also called words, that is $L(G) = \{w : S \stackrel{*}{\Rightarrow} w, w \in T^*\}$. The grammar used to build EA structures is depicted in Fig. 3. Two examples of words generated by the grammar (i.e., algorithms) are presented in Alg. 2 and Alg. 3. The key issue in GE is the mapping from the genotype into the phenotype, i.e., the translation of the binary string to an executable program. GE introduces a genotype to phenotype mapping in successive steps (see Fig. 4). Initially the genome is a binary linear sequence. The binary string is transcribed onto a sequence of integers, each one called a codon. Then a translation starts: a derivation tree (the phenotype) is obtained from the axiom of the grammar, and will be further decoded to an expression tree, the program. Each codon is used to determine the rule for a non-terminal symbol when it is expanded. Suppose that we have the following production rule,

$$
\begin{aligned}
< expr > ::= & < expr >< op >< expr > && (0)\\
& |(< expr >< op >< expr >) && (1)\\
& | < pre-op > (< expr >) && (2)\\
& | < var > && (3)
\end{aligned}
$$

where there are four options to rewrite its left hand side symbol $< expr >$. In the beginning we have our genome transcribed into a string of integers and a syntactical form equal to the axiom $< expr >$. We want to rewrite the axiom and must choose which alternative will be used. We take the first integer and divide it by the number of options for $< expr >$. The remainder of that operation will indicate the option to be used. In the example above, if we admit that the integer is 9 then we will have $9\%4 = 1$ and the axiom will be rewritten in $(< expr >< op >< expr >)$. Then we read the second integer and apply the same method to the left most non-terminal of the derivation. We iterate this process, that stops when we do not have more non-terminals to rewrite. If we run out of integers we use a wrapping mechanism: we restart from the beginning of the string of integers. The existence of redundancy is also worth noting, for different integers may correspond to the same alternative due to the nature of the operation remainder. In the example above, the integers $5, 9, 13, \ldots$ all codify for the same production alternative for $< expr >$. GE has been applied with success to different problems (for details see [17, 13, 14]).

5. EXPERIMENTS

In this section we study the capacity of evolving EAs to solve the RR problem, using the GE with the grammar already described. Firstly we study the ability of the GE

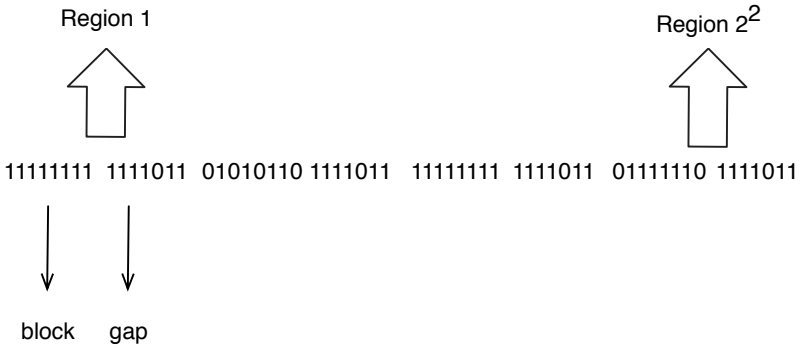

Figure 2: Royal Road Binary String with k = 2, b = 8, g = 7

Table 1: PART block fitness for The RR default parameters with $m^* = 4$ and $v = 0.02$

1's in block	0	1	2	3	4	5	6	7	8
Block Fitness	0.00	0.02	0.04	0.06	0.08	-0.02	-0.04	-0.06	0.00

Algorithm 2 Traditional Genetic Algorithm

```
1: (lambda,0.5)
2:     evaluate
3:     RouleteWheelSelection
4:     SinglePointCrossover(0.9)
5:     PointMutation(0.05)
6:     RnkReplacement
7:     Elite(0.01)
```

Algorithm 3 (20, 5) Evolutionary Strategy

```
1: (lambda,0.25)
2:     evaluate
3:     RouleteWheelSelection
4:     PointMutation(0.8)
5:     GenerationalReplacement
```

framework to evolve EA algorithms using a simple RR instance. Secondly we study the generalization ability by applying the obtained algorithms to several different RR instances, and analyze their solving capacity. The RR instances used are described in Table 2.

Table 2: Royal Road functions used

	Parameters							
Instance	k	b	g	m^*	v	u^*	u	Optimum
1	3	6	5	3	0.02	1.0	0.3	7.3
2	4	8	7	4	0.02	1.0	0.2	12.8
3	4	8	7	2	0.02	1.0	0.2	12.8
4	5	8	7	2	0.02	1.0	0.2	23.1
5	5	8	7	4	0.02	1.0	0.2	23.1

5.1 Training

The first phase of the experimental study was dedicated to analyze the capacity of the GE framework to evolve EAs. The settings used in the framework are presented in Table 3.

The quality of each individual generated by the GE is assessed by solving the instance 1 of Table 2. When solving one RR instance, the most effective algorithms are able to accomplish three main tasks: *1*) create complete blocks from scratch; *2*) complete nearly finished blocks; and *3*) join complete blocks; Therefore, to assign fitness to a solution generated by the GE, we perform 3 runs (one for each task),

Table 3: Parameters of the Grammatical Evolution Framework

Parameter	Value
One Point Crossover Probability	0.9
Bit Flip Mutation	0.01
Codon Duplication Probability	0.01
Codon Pruning Probability	0.01
Population Size	100
Selection	Tournament with size equal 5
Replacement	Steady State
Codon Size	8
Number of Wraps	3
Codons in the initial population	10-16
Generations	50
Number of Runs	30

of the aforementioned instance. In each different run we seed the initial population with solutions that allow to access the ability of the EA to succeed in one of the previously identified tasks (e.g., the ability to join complete blocks is tested by starting the EA with an initial population containing solutions with already some completed blocks). EAs ran for 2000 evaluations and the fitness value provided as feedback to the GE corresponds to the mean of the best level achieved in each of the scenarios. The limited number of runs was adopted because evaluating an EA is a computational intensive task.

The algorithms evolved by the GE were able to find the best solution for the selected training instance. The average

⟨*start*⟩ ::= (<pop_parameter>, <proportion>)(<EA>)

⟨*proportion*⟩ ::= 0.25
| 0.5
| 0.75
| 1.0

⟨*pop_parameter*⟩ ::= lambda
| mu

⟨*EA*⟩ ::= evaluate <selection> <variation> <replacement> <elitism>

⟨*selection*⟩ ::= RouleteWheel
| SUS
| RankBased
| Tournament(<t_size>)
| λ

⟨*t_size*⟩ ::= <integer_const>

⟨*integer_const*⟩ ::= random_integer

⟨*variation*⟩ ::= <operator><variation>
| λ

⟨*operator*⟩ ::= <recombination>(<prob>)
| <mutation>(<prob>)
| λ

⟨*recombination*⟩ ::= OnePoint
| NPoint(<integer_const>)
| Uniform
| λ

⟨*mutation*⟩ ::= BitFlip
| λ

⟨*replacement*⟩ ::= Generational
| RnkReplacement
| λ

⟨*e_size*⟩ ::= 0.01
| 0.05
| 0.1

⟨*prob*⟩ ::= 0.01
| 0.05
| 0.1
| 0.5
| 0.9
| 1.0
| <random_per>

⟨*elitism*⟩ ::= Elitism(<e_size>)
| λ

⟨*random_per*⟩ ::= random_0_1

Figure 3: Grammar used to evolve EAs

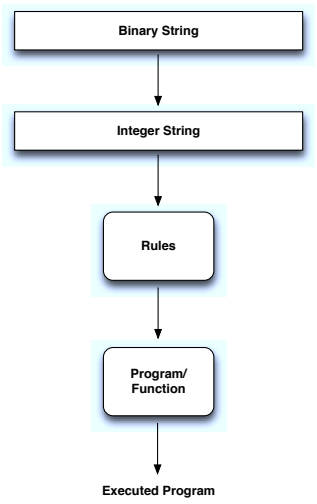

Figure 4: GE: from a binary string to a program (adapted from [14])

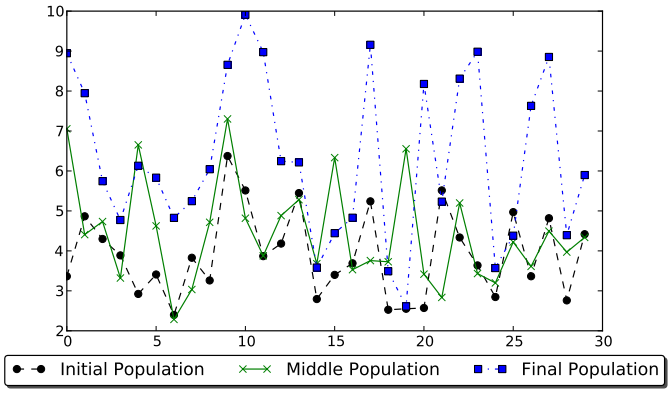

Figure 5: Evolution Analysis of 30 best individuals selected from the inital, middle and final populations

fitness obtained by the 30 best algorithms (one from each run) in the training phase was 7.099356 (± 0.530). To analyze if there was any evolution ocurring in the framework we took the 30 best individuals of the initial population, 30 best individuals of the middle population, and 30 best individuals of the last population and applied them to the instance 2 of Table 2. Fig. 5 presents the results of this analysis. The results show that evolution is occurring: looking at the fitness values of the individuals of the initial population we see that they are clearly the worst ones. The individuals in the middle population start to exhibit good capacities on the discovering of good solutions, whilst the ones on last population have the capacity of discovering better solutions than the other ones.

We also wanted to analyze if the GE engine was able to evolve innovative EA structures. Clearly, the defined grammar imposes syntactic limitations in the organization of evolved algorithms, thereby hindering the emergence of unusual structures. Moreover, it is well-known that stan-

dard EAs are effective in the optimization of RR functions. The combination of these two facts suggests the existence of a good direction for GE to evolve standard EAs.

An inspection of the 30 best evolved algorithms (i.e, the best evolved strategy in each GE run) confirms that the computational framework tends to converge to solutions similar to those regularly used to solve the RR functions. Most of the differences appear in the selection and replacement methods. In table 4 we present the frequency of appearance of the components in these best solutions (values are in percentage). The operators components are not exclusive, and they can appear together in the EA. In what concerns selection, there is not a clear winner, although roulette wheel is the least adopted method. As for replacement, the one based on rank clearly outperformed the generational approach. That is a more conservative strategy, maintaining in the population solutions that are not outperformed by descendants (in terms of fitness). Results show that keeping good solutions in the loop helps to enhance the effectiveness of algorithms when seeking for good RR solutions.

Table 4: Frequency of components appearance on the EAs evolution phase

Components		
Replacement	Generational	16.7%
	RankReplacement	80.0%
Selection	RouleteWheel	13.3%
	Rank	30.0%
	Stochastic Universal	26.7%
	Tournament	26.7%
Operators	SinglePointCrossover	36.7%
	NPointCrossover	16.7%
	UniformCrossover	30.0%
	PointMutation	36.7%

5.2 Test

The results of the previous section allowed us to assess the capacity of our GE framework to evolve EAs. However it is important to analyze how the evolved algorithms behave in instances that are different from the one used in training. This would help us to verify if the evolved EAs are competitive with the standard algorithm in RR optimization. In these set of experiments we selected the seven best evolved algorithms and a typical EA used to optimize the RR: roulette wheel selection, single point crossover with 0.9 probability, point mutation with 0.01 probability and generational replacement without elitism; applied them to instance 2 of Table 2. Now, each algorithm is allowed to perform 256000 function evaluations [10]. Optimization results are presented in Fig. 6.

Looking at the results we can see that the evolved algorithms have a similar behavior when compared to the standard one (algorithm number 8). To see if there was any statistical difference between the evolved algorithms and the standard one we applied a statistical test to assess if the mean best fitness of the evolved algorithms was equal to the mean best fitness of the standard one. The Friedman's ANOVA revelead statistical differences in the means. Then, and using the standard EA as control group (algorithm number 8 of Fig. 6), we applied Wilcoxon Signed Rank test at a significance level $\alpha = 0.05$ with Bonferroni correction. Table

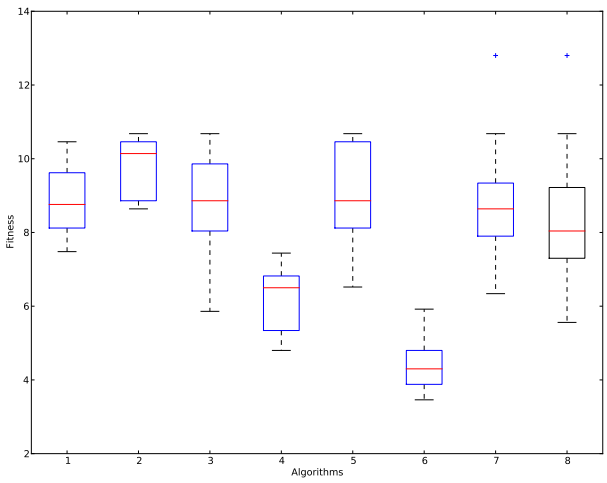

Figure 6: Validation results

5 shows the results of the statistical analysis. A $+$ represents a statistical significant difference in the means, and that the evolved algorithm has an higher rank; $-$ represents a difference in the means, and that the evolved algorithm has a lower rank; and $\sim$ represents that there was no difference in the means.

Table 5: Results of the Wilcoxon Signed Rank test at a significance level $\alpha = 0.05$

Pair	Result
Alg1 - StandardEA	$\sim$
Alg2 - StandardEA	$+$
Alg3 - StandardEA	$\sim$
Alg4 - StandardEA	$-$
Alg5 - StandardEA	$\sim$
Alg6 - StandardEA	$-$
Alg7 - StandardEA	$\sim$

The statistical results of Table 5 reveal that most of the evolved algorithms are equivalent to the standard EA. However two of the algorithms are worst than the standard and one is better. An analysis of the components of the worst algorithms reveals that Alg6 relies on Uniform Crossover as the main variation operator, which is probably too disruptive to RR optimization. Alg4 is composed by the two typical variation operators, point mutation and single point crossover. However the crossover has low probability rate, delaying the combination of blocks.

The structure of Alg2 is different from the usual and is presented in Alg. 4. It has 2 types of crossover (NPoint crossover with 16 cut-points and single point point crossover, both with a high probability rate), separated by the application of point mutation with a low probability rate. Since the algorithm is different from the usual architectures we applied it to three additional instances of the RR (instances 3, 4, 5 of Table 2). Table presents the results of the statistical comparison between Alg4 and the standard EA, using the Wilcoxon Signed Rank test at a significance level $\alpha = 0.05$. Looking at the presented results we can see that

Algorithm 4 Evolved algorithm Alg4

```
Rank Selection
NPointCrossover(1.0, 16)
PointMutation(0.01)
SinglePointCrossover(1.0)
Rank Replacement
```

the standard EA performs better in the two harder instances ($m^* = 2$), and that the Alg4 perform better in the other. These results seems to indicate that Alg4 promotes the appearance of more 1s in the blocks than the the standard EA which is not good when we have a small m^*, since many 1s lead to higher penalizations. On the contrary, Alg. 4 effectively optimizes the instance with a higher m^*, as this defines a situation where more 1s do not lead to an excessive penalization.

Table 6: Statistical Results of the comparison Alg4-Standard EA

Instance	Result
Instance 3	−
Instance 4	−
Instance 5	+

6. CONCLUSIONS

In this paper, a GE framework to evolve EAs was proposed. The grammar presented is composed by a full set of existing components. The flexibility of the grammar is limited, hindering the appearance of unusual architectures, still it allows some variations on number and order of operators that can appear. To assess the capacity of the GE framework to evolve EAs we used RR functions as the benchmark problem, to which we knew one good algorithmic solution. The results of the experiments revealed that the framework is able to evolve well known algorithm architectures. Moreover it was able to evolve an architecture that is different from the traditional ones. Additional experiments were conducted to assess the effectiveness of this new architecture.

The work presented in this paper is preliminary, but it suggests that the automatic evolution of EA is possible. Furthermore, it raises several important research questions such as: *1*) How can we assess the quality of an EA, and retain the important information, while reducing the computational effort? *2*) How to select the instances of a certain problem to the algorithm training phase? *3*) Which information should we take from the evaluation of the evolved algorithms? Lastly, and pointing towards future work, applying this approach to solve other problems would be important in order to achieve a framework that is able to evolve EAs.

7. ACKNOWLEDGMENTS

This work was supported by Fundação para a Ciência e Tecnologia (FCT), Portugal, under the grant SFRH/BD/79649/2011.

8. REFERENCES

[1] P. J. Angeline. Adaptive and self-adaptive evolutionary computations. In *Computational Intelligence: A Dynamic Systems Perspective*, pages 152–163. IEEE Press, 1995.

[2] P. J. Angeline. Two self-adaptive crossover operators for genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in genetic programming*, pages 89–109. MIT Press, 1996.

[3] E. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg. Hyper-Heuristics: An Emerging Direction in Modern Search Technology. In *Handbook of Metaheuristics*, International Series in Operations Research & Management Science, chapter 16, pages 457–474. Springer, 2003.

[4] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and J. R. Woodward. A classification of hyper-heuristics approaches. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*, chapter 15, pages 449–468. Springer, 2nd edition, 2010.

[5] L. Dioşan and M. Oltean. Evolutionary design of Evolutionary Algorithms. *Genetic Programming and Evolvable Machines*, 10(3):263–306, 2009.

[6] B. Edmonds. Meta-genetic programming: Co-evolving the operators of variation. CPM Report 98-32, Centre for Policy Modelling, Manchester Metropolitan University, UK, 1998.

[7] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.

[8] J. H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.

[9] T. Jones. A description of holland's royal road function. *Evolutionary Computation*, 2(4):409–415, 1994.

[10] M. Mitchell, S. Forrest, and J. H. Holland. The royal road for genetic algorithms: Fitness landscapes and ga performance. In *Proceedings of the First European Conference on Artificial Life*, pages 245–254. MIT Press, 1991.

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

[12] M. Oltean and C. Grosan. Evolving evolutionary algorithms using multi expression programming. In *Proceedings of The 7 th European Conference on Artificial Life*, pages 651–658. Springer-Verlag, 2003.

[13] M. O'Neill. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4), 2001.

[14] M. O'Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, volume 4 of *Genetic programming*. Kluwer Academic Publishers, 2003.

[15] R. Poli, C. Di Chio, and W. B. Langdon. Exploring extended particle swarms: a genetic programming approach. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 169–176, New York, NY, USA, 2005. ACM.

[16] P. Ross. Hyper-heuristics. In E. K. Burke and G. Kendall, editors, *Search Methodologies*, pages 529–556. Springer US, 2005.

[17] C. Ryan, J. Collins, and M. OŃeill. Grammatical evolution: Evolving programs for an arbitrary language. In *Lecture Notes in Computer Science 1391, Proceedings of the First European Workshop on*

Genetic Programming, pages 83–95. Springer-Verlag, 1998.

[18] J. Tavares, P. Machado, A. Cardoso, F. Pereira, and E. Costa. On the evolution of evolutionary algorithms. In M. Keijzer, U.-M. OReilly, S. Lucas, E. Costa, and T. Soule, editors, *Genetic Programming*, volume 3003 of *Lecture Notes in Computer Science*, pages 389–398. Springer Berlin / Heidelberg, 2004.

[19] J. Tavares and F. B. Pereira. Automatic design of ant algorithms with grammatical evolution. In *Proceedings of the 15th European conference on Genetic programming*, 2012.