The Importance of the Learning Conditions in Hyper-Heuristics

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

Evolutionary Algorithms are problem solvers inspired by nature. The effectiveness of these methods on a specific task usually depends on a non trivial manual crafting of their main components and settings. Hyper-Heuristics is a recent area of research that aims to overcome this limitation by advocating the automation of the optimization algorithm design task. In this paper, we describe a Grammatical Evolution framework to automatically design evolutionary algorithms to solve the knapsack problem. We focus our attention on the evaluation of solutions that are iteratively generated by the Hyper-Heuristic. When learning optimization strategies, the hyper-method must evaluate promising candidates by executing them. However, running an evolutionary algorithm is an expensive task and the computational budget assigned to the evaluation of solutions must be limited. We present a detailed study that analyses the effect of the learning conditions on the optimization strategies evolved by the Hyper-Heuristic framework. Results show that the computational budget allocation impacts the structure and quality of the learned architectures. We also present experimental results showing that the best learned strategies are competitive with state-of-the-art hand designed algorithms in unseen instances of the knapsack problem.

Categories and Subject Descriptors

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

Keywords

Evolutionary Algorithms, Hyper-Heuristics, Automatic Evolution

1. INTRODUCTION

Evolutionary Algorithms (EAs) are computational problem solvers loosely inspired by the principles of natural selection posit by Darwin. Over time, they have been successfully applied to complex problems, i.e., those that do not have an analytical solution or are computationally intractable, in areas like optimization, learning or design. When solving a particular problem, EAs usually require a carefully adjustment of some of its parameters and components. This might be a difficult task requiring specific skills, and it is typically performed off-line, by hand, even though there are EA variants that have a limited self-adaptation capacity.

The automatic adaptation of EAs to a specific problem is a promising way to replace specialized manual adjustments. In the last years, several proposals have been made that perform the autonomous evolution of bio-inspired approaches, namely Particle Swarm Optimization [9], Ant Colony Optimization [14] and Evolutionary Algorithms [3, 4]. Hyper-Heuristics (HH) are an emerging framework to accomplish this task. In general, this term identifies architectures where problem-solving methods search a set of possible heuristics, i.e., high level heuristics choose low level heuristics [12].

There are two main groups of HH. One seeks for the best sequence of low-level heuristics, chosen from a predetermined set of methods usually applied to solve the specific problem under consideration. The second group aims at generating or constructing new heuristics. In this case, the HH framework automatically learns the low-level algorithm needed to solve the problem at hand. Learning feedback is obtained by executing each solution candidate over one, or several, simple instances of the problem one aims to solve. Genetic Programming (GP) is a EA branch that searches the space of programs for effective algorithmic strategies. For that reason, it has been increasingly adopted as an HH. In concrete, the Grammatical Evolution (GE) [6, 7] variant has been successfully used in recent works [1, 4, 14], as it allows for a straightforward enforcement of semantic and syntactic restrictions.

GE based HH incurs on a learning process with a high computational effort, as the quality of each generated strategy must be estimated by applying it to an optimization situation. This leads to the appearance of two contradictory forces that must be balanced. On the one hand, the off-line learning should not take too long, which implies relying on small instances and adopting parameters (e.g., population size, number of iterations) that minimize the computational overhead. On the other hand, the adoption of excessively simple conditions might compromise results, hindering differences between competing strategies and potentially leading to an inaccurate assessment of the quality of evolved solutions. The experiments described in this paper aim to contribute to a better understanding of the impact of the learning conditions in the evolved strategies and to identify useful guidelines that help to define better HH. With this goal in mind, we present a GE based HH for evolving complete EAs that are effective in solving different instances of the 0-1 Knapsack Problem. Additionally, we will investigate the generalization ability of the best evolved EAs by applying then to unseen instances of the problem.

The paper is structured as follows: Section 2 describes the framework used. Section 3 introduces the experimental setup and presents an empirical study on the influence of the learning conditions on the structure of the evolved EAs. Section 4 compares the effectiveness of EAs evolved under different conditions by applying them to unseen instances of the knapsack problem. Finally, Section 5 gathers the main conclusions and suggests directions for future work.

2. THE HYPER-HEURISTIC FRAMEWORK

In this section we present a brief overview of recent research from the HH area that rely on GP approaches to evolve problem solving strategies. Next we describe the grammar used in our work.

2.1 Related Work

Several efforts have been reported in the literature to automatically evolve nature-inspired algorithms. In [13], Tavares et al. adopted GP to evolve a population of mapping functions between the genotype and the phenotype. Experimental results showed that GP finds mapping functions that can obtain results as good as the ones that are designed by hand.

In [3], Dioşan et al. adopted a two-level architecture to evolve full-fledged EAs. At a higher level, an EA with fixed settings is used, while, at the lower level, a solution is produced in the form of a sequence of genetic operations and corresponding parameters. Best solutions evolved are next used to solve specific optimization problems. The results presented confirm that it is possible to evolve EAs that perform similarly to the ones used for comparison.

In [9], Poli et al. evolved, using GP, the equation that controls the movement of particles in a Particle Swarm Optimization. Experimental results showed that the evolved strategy could effectively solve some selected problems.

In [8], Pappa et al. proposed a Grammar-Based GP (GBGP) framework to learn data mining algorithms. The evolved algorithms are competitive with state of art human-designed methods for data discovery. Recently, Burke et al. [1] presented a GE approach to evolve heuristics for the bin packing problem. The experimental analysis of the work focused on the quality, efficiency and consistency of the evolved heuristics. Best evolved heuristics were able to satisfy all these features.

In [14], Tavares et al. proposed a GE framework to evolve Ant Colony Optimization Algorithms to the Traveling Salesman Problem. Finally, Lourenço et al. [4] proposed a GE based HH to evolve full-featured EAs. The results showed that the proposed architecture is able to evolve effective algorithms for the problem considered in that study.

$$
\begin{array}{rll}
< expr > ::= & < expr >< op >< expr > & (0) \\
 & |(< expr >< op >< expr >) & (1) \\
 & | < var > & (2) \\
< op > ::= & + & (0) \\
 & |- & (1) \\
< var > ::= & x & (0) \\
 & |y & (1)
\end{array}
$$

Figure 1: Example grammar

2.2 Grammatical Evolution

GE is a branch of GP and a form of GBGP. In GE, a linear genome representation is used, typically a binary string, and there is a clear separation between the genotype and the phenotype. The mapping genotype-phenotype is done by a grammar. The grammar can contain domain knowledge and the search engine is independent of the evaluation mechanism. For these reasons, a GE system is general and flexible [7]. The grammars used by GE are context-free. A context-free grammar is defined by 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 $\alpha \in (N \cup T)^*$ (see Fig. 1 for an example). Note that N and T are disjoint. A 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 $L(G) = \{w : S \overset{*}{\Rightarrow} w, w \in T^*\}$.

GE performs the genotype to phenotype mapping in successive steps. The initial binary linear sequence (i.e., the genome) is first transcribed onto a sequence of integers, each one called a *codon*. Then, the sequence of codons is used to decide which production of the grammar is used to construct a derivation tree from the axiom. Finally, the phenotype, i.e., an executable program, is extracted from the derivation tree. As an example, consider that we have the grammar exemplified in Fig. 1. In the beginning, the genome is transcribed into a string of integers and we start with a syntactical form equal to the axiom $< expr >$. The objective is to rewrite the axiom and one must choose which of the three options is selected. The first integer from the string is divided by the number of options for $< expr >$. The remainder of that operation will indicate the option to be used. In the example, if the integer is 153 then $153\%3 = 0$ and the axiom will be rewritten as $< expr >< op >< expr >$. Then the decoding proceeds by reading the second integer and apply the same method to the leftmost non-terminal of the derivation. This process is iterated, until there are no more non-terminals to rewrite. If the decoding process runs out of integers, a wrapping mechanism is used, i.e., the process goes back to the beginning of the string. 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 $3, 27, 153$, all codify for the same production alternative for $< expr >$. GE has been applied with success to different problems (for details see [6, 7]).

2.3 Grammar Definition

Our HH framework relies on GE to learn full-fledged EAs. We must then define a grammar whose words are EAs, i.e., the grammar must allow the generation of a complete algorithm, defining both its main components and its settings. In the present work, we rely on the grammar presented in Fig. 2, where $< start >$ plays the role of the axiom. The grammar enforces a sequential construction of components, thereby modeling the overall structure of the evolved algorithms. The grammar implicitly describes algorithmic components, such as selection, variation operators and replacement strategy; likewise, it also specifies parameters including the number of individuals in the initial population, the number of offspring created at each generation and the probability of applying variation operators. An inspection of the right hand side of the first production clarifies how we constrain the general structure of the evolved solutions: first, parameters *mu* and *lambda* are specified, corresponding to the number of individuals in the initial population and number of offspring created at each generation, respectively. Both values are defined in proportion to the maximum population size granted to the algorithm. Afterwards, a cycle iterates over a predetermined sequence of typical EA operations: evaluation, selection, variation and replacement. The grammar defines alternatives for each component, thus allowing the GE to learn the most suitable combination for a given situation. It is worth noting that the grammar allows the generation of solutions without some components (option ϵ denotes the empty string). In addition, several recombination and mutation operators can simultaneously appear in the same EA.

3. LEARNING EVOLUTIONARY ALGORITHMS

Experiments described in this section aim to gain insight on how the learning conditions determine the overall structure of the evolved strategies. The settings of the GE adopted for all the tests are presented in Table 1 [7].

The quality of the strategies generated by the GE is directly related to their optimization ability. In our framework, we apply each learned algorithm to a predetermined problem instance and consider that its fitness corresponds to the quality of the best solution found. Evaluating a GE population is then a computationally intensive task. To prevent the learning process from taking an excessive amount of time, we rely on the following conditions to estimate the quality of evolved strategies: i) one single instance of moderate size is used to assign fitness; ii) only one run is performed; iii) the number of evaluations is kept low. To investigate how these design options impact the quality and structure of the solutions learned by the GE, we present learning results obtained with different conditions. In concrete, this paper focuses on the length of the run used to assign fitness and also on how the number of evaluations is split between generations and population size. We consider 4 settings, detailed in Table 2.

3.1 The 0-1 Knapsack Problem

The combinatorial optimization *0-1 Knapsack Problem* (KP) was selected as the testbed for our experiments. It can be described as follows: given a set of n items, each of which with some profit p and some weight w, how should a

```
⟨start⟩ ::= mu = ⟨proportion⟩
        lambda = ⟨proportion⟩
        while(not termination condition) do
        evaluate
        ⟨selection⟩
        ⟨variation⟩
        ⟨replacement⟩
        end while


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

⟨selection⟩ ::= RouleteWheel()
      | SUS()
      | Rank()
      | Tournament(⟨t-size⟩)
      | ϵ

⟨t-size⟩ ::= ⟨integer-const⟩

⟨integer-const⟩ ::= randominteger()

⟨variation⟩ ::= ⟨operator⟩
      | ⟨operator⟩ ⟨variation⟩

⟨operator⟩ ::= ⟨recombination⟩
      | ⟨mutation⟩

⟨recombination⟩ ::= SinglePointXover(⟨prob-recombination⟩)
      | NPointXover(⟨prob-recombination⟩,⟨integer-const⟩)
      | UniformXover(⟨prob-recombination⟩)
      | ϵ

⟨mutation⟩ ::= PointMutation(⟨prob-mutation⟩)
      | BinarySwapMutation(⟨prob-mutation⟩)
      | ϵ

⟨prob-recombination⟩ ::= 0.5
      | 0.7
      | 0.9
      | 1.0
      | ⟨random-per⟩

⟨prob-mutation⟩ ::= 1.0 / n
      | 2.0 / n
      | 5.0 / n
      | 10.0 / n
      | ⟨random-per⟩

⟨random-per⟩ ::= random01()

⟨replacement⟩ ::= Generational()
      | RankReplacement()
      | RankReplacementNoDup()
      | Elitist(⟨random-per⟩)
      | ϵ
```

Figure 2: Grammar used to evolve EAs

Table 1: Parameters of the GE 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 3
Replacement	Steady State
Codon Size	8
Number of Wraps	3
Codons in the initial population	10-16
Generations	50
Runs	30

subset of items be selected to maximize the profit while keeping the sum of the weights bounded to a maximum capacity C? In all instances adopted in our study, the knapsack capacity was set to half of the sum of the weights of all items. A standard binary representation is adopted and evaluation considers a linear penalty function to punish invalid solutions [5].

3.2 Learning Results

Table 2: Evaluation Learning Settings

Setting	Population size	Number of Generations	Evaluations
1	20	100	2000
2	20	250	5000
3	50	100	5000
4	50	250	12500

Table 3: GE Learning Results

Setting	Mean Best Fitness (MBF)	Best Hits
1	38961.94 (±73.73)	0 / 30
2	39084.77 (±12.23)	29 / 30
3	39087.00 (±00.00)	30 / 30
4	39087.00 (±00.00)	30 / 30

The KP instance selected to evaluate learned EAs is composed by $n = 100$ items, in which the profit of the optimal solution is 39087. Table 3 summarizes the results of the off-line learning step. For each setting, column MBF displays the mean of the best strategies found in the 30 runs performed by the GE and the corresponding standard deviation (in brackets). Last column (Best Hits) presents the number of runs where the GE evolved strategies that were able to discover the optimal solution of the selected instance. The chart from Fig. 3 displays the evolution of MBF, for all settings, along the 50 GE generations. These results show that the framework gradually learns better optimization strategies. Moreover, the outcomes in the Table 3 reveal that evolved EAs are able to discover the optimum or near-optimum solutions. Results obtained with setting 1 are an exception to this general rule. The low number of evaluations granted to each EA to solve the KP instance (between 16% and 40% of the computational budget granted by the other settings) prevents the discovery of the highest quality solutions. Note that this does not necessarily imply that the GE with setting 1 is unable to find good optimization strategies. Learning is also occurring with setting 1 (see the corresponding line in Fig. 3) and the optimization ability of the best strategies evolved in this scenario will be accessed in the next section.

A detailed inspection reveals that different learning conditions (as defined by the 4 settings previously described) impact the structure of the evolved algorithms. For all settings, we selected the best EA learned in each run and created charts that measure the frequency of appearance of the main components (the numerical settings are not considered in this analysis). Fig. 4 contains 3 panels that group the components by type: panels a), b) and c) display selection options, replacement options and variation operators, respectively. Unlike selection and replacement, components in panel c) are not exclusive. Virtually all best learned strategies rely on RankReplacementNoDup, a replacement mechanism based on the rank of the solutions with elimination of duplicates. This is true for all settings and is in accordance with the literature that states that this mechanism outperforms all other considered replacement components [10]. In what concerns selection, there is not a clear winner, although roulette wheel and rank mechanisms are slightly prevalent.

Interesting patterns arise in the selection of variation operators. For all settings, uniform crossover and binary swap mutation achieve the highest percentage, suggesting a clear advantage over the other alternatives when exploring the search space of the KP instance selected for learning. However, a close inspection of panel c) reveals a remarkable difference between settings 2 and 3. In spite of both having the same computational budget (5000 evaluations to estimate the quality of each EA), the mutation operator is frequently disregarded in the best strategies learned with setting 3. This may be explained by examining how the computational budget is allocated. In setting 2, a low population size (20 individuals) is iterated for a considerably high number of generations (250), which might lead to premature convergence. In these conditions the mutation operator plays a crucial role in diversity maintenance, thus avoiding convergence. On the contrary, setting 3 has a higher population size (50 individuals) coupled with a lower number of generations (100). Convergence is hardly a problem and, given the moderate size of the KP instance, the EA can probably rely just on uniform crossover to perform an appropriate sampling of the search space. Therefore, it is not surprising that many EAs learned with setting 3 prefer to not include mutation.

To confirm this hypothesis, we ran an additional set of experiments, with a slightly modified grammar, that only allows the appearance of a single variation operator in each evolved structure. This way, the hyper-heuristic has to select the most suitable operator (either crossover or mutation) to include in the optimization strategy, given specific learning conditions. We repeated the experiments for all four settings presented in Table 2. Fig. 5 displays the frequency of appearance of the variation operators using the modified grammar. An overview of the chart confirms our claims. Strategies evaluated with low population size (settings 1 and 2) need to incorporate mutation operators to prevent premature convergence. When the population size is high and the number of generations is low (setting 3), all the EA needs is crossover. Finally, setting 4 grants the EA a considerable computational budget to run. This allows the appearance of strategies that can either rely on crossover or mutation. It is worth notice that, in the experiments performed with the modified grammar, binary swap mutation is completely absent from the best learned solutions. This is probably related to the fact that this operator alone cannot modify the number of items in the knapsack, thereby preventing a full exploration of the search space.

Results presented in Figs. 4 and 5 reveal that different EA architectures emerge when different learning conditions are adopted. A complete understanding of the whole process is still in progress (e.g., the size or hardness of the instance selected to evaluate learned strategies might also play a relevant role), but it is, nevertheless, a relevant contribution to a better understanding of the impact of learning conditions adopted by a hyper-heuristics framework. To complete this section, in Algorithms T1, T2, T3 and T4 we present the best evolved algorithm for each one the settings, labelled according to the scenario where they were discovered. In agreement with the previous analysis, they share the same replacement method and tend to rely on different selection strategies. In what concerns the variation operators, T1, T2 and T4 include both crossover and mutation, whereas T3 relies solely on crossover.

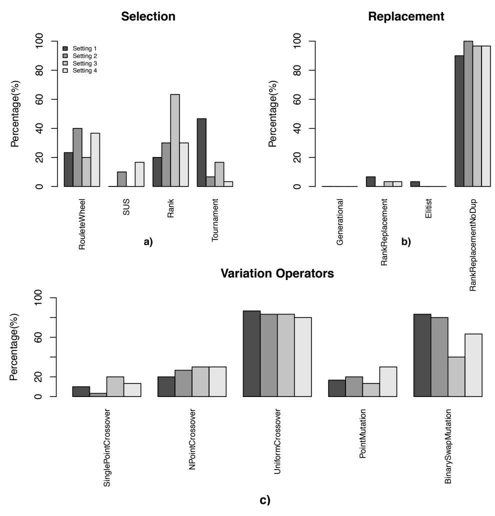


Figure 4: Frequency of components in the best evolved solutions: panels a), b) and c) display selection, replacement and variation operators, respectively.

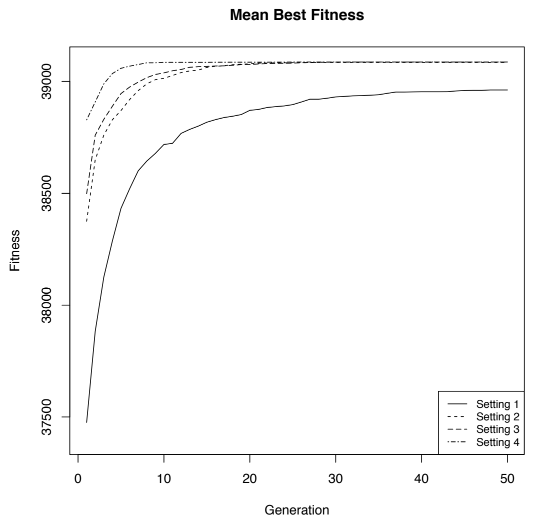


Figure 3: Evolution of GE MBF in the 4 Learning Settings

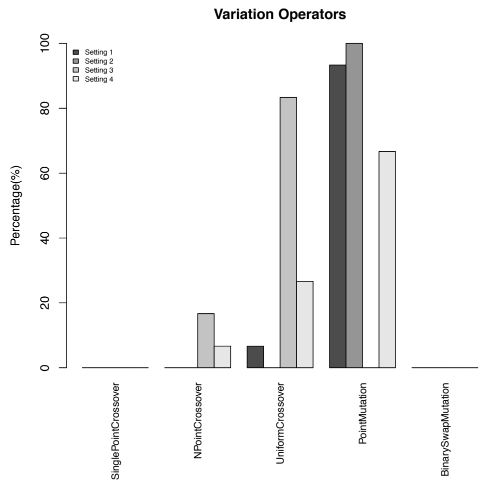


Figure 5: Frequency of components in the best evolved solutions with just one variation operator.

4. VALIDATION OF THE LEARNED EVOLUTIONARY ALGORITHMS

We present now a set of experiments to analyze how the 4 best evolved algorithms (T1, T2, T3, T4) behave in KP instances that are different from the one used in learning. Such study will help to gain insight into the optimization performance differences that may eventually arise between strategies learned in different conditions. Also, we will verify if the evolved EAs generalize well to unseen instances and are competitive with hand design approaches regularly applied to the KP. Three hand-designed algorithms (HEA1, HEA2, HEA3), based on proposals that appeared in the literature [2, 5, 11], were considered in this study. All hand-designed methods adopt the RankReplacementNoDup replacement mechanism and tournament selection with tourney size 3. They differ in the variation operators and/or corresponding rate of application, as detailed in Table 4. The EAs were applied to different KP instances, with a number of items ranging between 500 and 1000. Due to space limitations we present results obtained with just two instances (details in Table 5), although the same optimization trend is visible for other situations. To mimic the training conditions, we created four different validation scenarios in which we varied the population size and number of generations. The settings are detailed in Table 6. All EAs were applied to the two KP instances with each one of these settings. In every optimization scenario, 30 runs were performed and the best solution found was recorded.

The last two columns of Table 7 contain the optimization results, obtained in each one of the validation instances.

Algorithm T1

```
mu = 1.0
lambda = 1.0
while not termination condition do
    evaluate
    Tournament(11)
    UniformCrossover(1.0)
    BinarySwapMutation(1.0 / n)
    RankReplacementNoDup()
end while
```

Algorithm T2

```
1: mu = 0.5
2: lambda = 1.0
3: while not termination condition do
4:     evaluate
5:     RouleteWheel()
6:     UniformCrossover(0.9)
7:     BinarySwapMutation(2.0 / n)
8:     RankReplacementNoDup()
9: end while
```

Algorithm T3

```
1: mu = 1.0
2: lambda = 1.0
3: while not termination condition do
4:     evaluate
5:     Rank()
6:     UniformCrossover(1.0)
7:     RankReplacementNoDup()
8: end while
```

Algorithm T4

```
1: mu = 1.0
2: lambda = 1.0
3: while not termination condition do
4:     evaluate
5:     RouleteWheel()
6:     BinarySwapMutation(1.0 / n)
7:     UniformCrossover(0.9)
8:     RankReplacementNoDup()
9: end while
```

Table 4: Hand-designed EAs: Variation operators and rate of application.

ID	Variation Operator	Rate
HEA1	SinglePointCrossover	0.9
	PointMutation	1 / n
HEA2	UniformCrossover	0.9
	PointMutation	1 / n
HEA3	UniformCrossover	0.9
	BinarySwapMutation	1 / n

Table 5: KP instances used for validation.

	Parameters	
Instance	Items (n)	Best Solution
1	500	169350
2	1000	333460

Table 6: Validation settings.

Setting	Population size	Number of Generations	Evaluations
1	50	100	5000
2	50	400	20000
3	100	200	20000
4	100	5000	500000

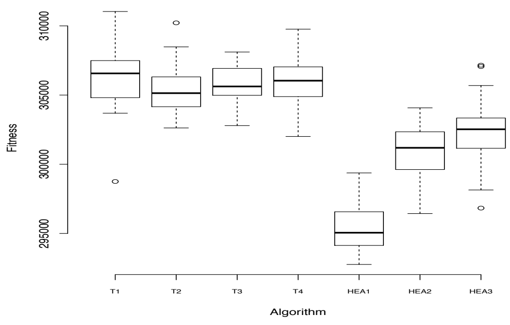
Values represent the MBF deviation from the optimum (in percentage). A brief perusal of the Table reveals that, in general, the learned EAs perform well across the different scenarios (as determined by the combination of a given instance and setting), suggesting that they are able to generalize, beyond the specific situation used for learning. Results also show that the effectiveness of the learned EAs is comparable to the hand-designed approaches, confirming that the GE framework was able to learn meaningful combinations of components and settings.

Despite the good general behavior of the evolved strategies, there are some differences in performance that require a detailed analysis. For the remainder of this section we concentrate on results obtained with validation instance 2 (the one with the higher number of items) and will investigate how a specific learning setting influences the generalization ability of evolved optimization strategies. The analysis is supported by a Friedman's ANOVA test that checks for statistical differences in the means obtained by the EAs considered in this study. When differences are detected, the *post-hoc* Wilcoxon Signed Rank Test, with Bonferroni correction, is applied to perform the pairwise comparisons. In both tests we used a significance level $\alpha = 0.05$.

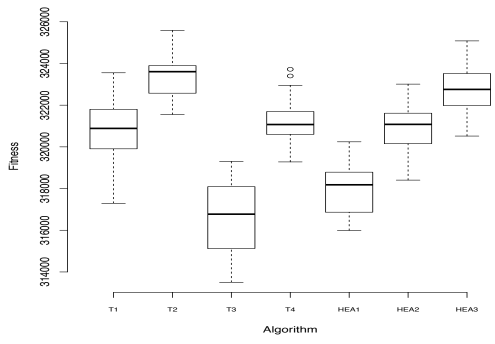
Fig. 6 presents the MBF box plot distribution of the 7 EAs (both evolved and hand-designed). Four panels, corresponding to each one of the validation scenarios, are displayed. Clearly, the performance of evolved algorithms depends on the settings adopted for the optimization. Even though differences are not always statistically significant, T1 achieves the highest MBF in setting 1, T2 is the best method in setting 2, T3 outperforms all other approaches in setting 3, and T4 is the best in setting 4. This confirms that evolved strategies contain specific features that allow them to excel in situations similar to those found during learning.

Results from Table 8 help to further clarify the relative performance of learned strategies. Considering the MBFs attained, we performed a full set of pairwise comparisons between evolved methods and present a graphical overview: A $+++$ sign indicates that the algorithm in the row is statistically better than the one in the column, and that the effect size is large ($r \geq 0.5$). As an example, T1 clearly outperforms T3 in setting 2. A $++$ sign indicates that there are statistical differences, and that the effect size is medium ($0.3 \leq r < 0.5$), whereas a $+$ identifies a significant difference with a small effect size ($0.1 \leq r < 0.3$). A - signals scenarios where the algorithm in the row is worst than the one in the column. Finally, a $\sim$ indicates that no statistical differences between the algorithms were found.

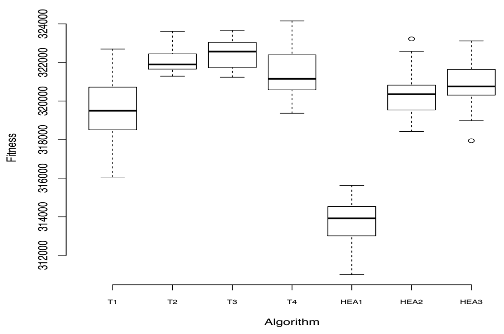
If we count the number of comparisons where a given EA outperforms another learning strategy, we can see that this



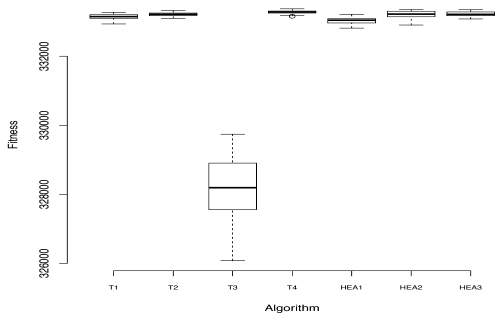
a) Setting 1



b) Setting 2



c) Setting 3



d) Setting 4

Figure 6: Box plot distribution of the MBF obtained by different EAs in the validation instance 2. Each panel corresponds to a given scenario as described in Table 6.

happens in 3 occasions for T1 and T3, 6 times for T4 and 7 times for T2. Moreover, the optimization advantage exhibited by T1 and T3 tends to be concentrated in settings that are similar to those they found during off-line learning. On the contrary, T2 and T4 maintain a reasonable effectiveness in unseen conditions. It seems then that these last two EAs have an increased robustness, which allows them to adapt to different optimization scenarios. On the contrary, T1 and T3 are brittle and tend to have a poor performance when the optimization conditions differ from those found during learning (see the position of some of the T1 and T3 box plots in Fig. 6). These outcomes allow us to conclude that the number of generations used to assign fitness in the off-line learning step is more important than the population size. The evolved strategies must be executed for a reasonable number of generations, in order to obtain an accurate estimate of their optimization ability. Finally it is worth noting that learning settings 2 and 4 were able to evolve algorithms with comparable effectiveness and robustness, even though setting 2 only needs 40% of the computational budget to evaluate candidates (when compared to setting 4).

5. CONCLUSIONS

GE is being increasingly adopted as the meta search method inside a HH framework. In this paper we analyzed a set of experiments to gain insight into the influence that learning conditions play in the effectiveness and robustness of evolved strategies. The 0-1 KP was selected as the target problem. Different off-line learning settings were defined, in what concerns the population size and the number of generations used to estimate the fitness of solutions evolved by the GE. The HH framework was executed in the different learning scenarios and the best evolved strategies were subsequently applied to unseen and larger KP instances. In the validation step, alternative optimization scenarios, with similar features to those adopted in the learning step, were considered. As a rule, evolved strategies obtained extremely good results in scenarios similar to those that they found during learning, showing that they contain features that are particularly suited for a specific environment. However, learning settings that allow the execution of a higher number of generations to estimate the fitness of solutions generated by the GE, allow the discovery of strategies with enhanced robustness. This is a relevant guideline for designing GE based HH, as it helps to distribute the existing computational budget for evaluating solutions.

The study presented here raises important research questions. In the near future we will investigate other learning design options, such as the adoption of a single or several training instances with different properties, to analyze how this impacts the quality and robustness of the evolved strategies. Also, the analysis must be expanded to other problems, in order to understand if the guidelines suggested in this paper can also be generalized to other optimization situations.

Table 7: Optimization results of the best learned EAs and hand-designed algorithms. The values depicted represent the MBF deviation from the optimum (in percentage).

		Instance	
Setting	**Algorithm**	1	2
1	T1	5.6	8.2
	T2	5.4	8.4
	T3	5.6	8.3
	T4	5.5	8.2
	HEA1	7.8	11.4
	HEA2	6.2	9.8
	HEA3	5.8	9.3
2	T1	1.8	3.8
	T2	1.4	3.0
	T3	2.9	5.0
	T4	1.7	3.7
	HEA1	2.3	4.7
	HEA2	1.7	3.9
	HEA3	1.4	3.2
3	T1	2.0	4.2
	T2	1.6	3.4
	T3	1.7	3.3
	T4	1.8	3.6
	HEA1	3.0	5.9
	HEA2	1.7	3.9
	HEA3	1.8	3.7
4	T1	0.0	0.1
	T2	0.0	0.1
	T3	0.8	1.6
	T4	0.0	0.1
	HEA1	0.1	0.1
	HEA2	0.0	0.1
	HEA3	0.0	0.1

Table 8: Statistical analysis between the learned architectures using the Wilcoxon Signed Rank Test ($\alpha = 0.05$).

	Settings	**Alg. T1**	**Alg. T2**	**Alg. T3**	**Alg. T4**
Alg. T1	1		+++	$\sim$	$\sim$
	2		-	+++	$\sim$
	3		-	-	-
	4		-	+++	-
Alg. T2	1	-		$\sim$	-
	2	+++		+++	+++
	3	+++		-	++
	4	+++		+++	-
Alg. T3	1	$\sim$	$\sim$		$\sim$
	2	-	-		-
	3	+++	++		+++
	4	-	-		-
Alg. T4	1	$\sim$	+	$\sim$	
	2	$\sim$	-	+++	
	3	+++	-	-	
	4	+++	+++	+++	

6. ACKNOWLEDGMENTS

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

7. REFERENCES

[1] E. Burke, M. Hyde, and G. Kendall. Grammatical evolution of local search heuristics. *IEEE Transactions on Evolutionary Computation*, 16(3):406–4127, 2012.

[2] P.C. Chu and J.E. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4:63–86, 1998.

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

[4] Nuno Lourenço, Francisco Pereira, and Ernesto Costa. Evolving evolutionary algorithms. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, GECCO '12, pages 51–58, New York, NY, USA, 2012. ACM.

[5] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs (3rd ed.)*. Springer-Verlag, London, UK, UK, 1996.

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

[7] M. O'Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[8] G. L. Pappa and A. Freitas. *Automating the Design of Data Mining Algorithms: An Evolutionary Computation Approach*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[9] 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.

[10] G. R. Raidl. On the importance of phenotypic duplicate elimination in decoder-based evolutionary algorithms. In *Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference*, pages 204–211, 1999.

[11] G R Raidl and J Gottlieb. Empirical analysis of locality, heritability and heuristic bias in evolutionary algorithms: A case study for the multidimensional knapsack problem. *Evolutionary computation*, 13(4):441–475, 2005.

[12] P. Ross. Hyper-heuristics. In Edmund K. Burke and Graham Kendall, editors, *Search Methodologies: introductory tutorials in optimization and decision support techniques*, chapter 17, pages 529–556. Springer US, 2005.

[13] J. Tavares, P. Machado, A. Cardoso, F. B. Pereira, and E. Costa. On the evolution of evolutionary algorithms. In Maarten Keijzer, Una-May OReilly, Simon Lucas, Ernesto Costa, and Terence Soule, editors, *Genetic Programming*, volume 3003 of *Lecture Notes in Computer Science*, pages 389–398. Springer Berlin / Heidelberg, 2004.

[14] 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.