Noname manuscript No.
(will be inserted by the editor)

Designing Automatically a Representation for Grammatical Evolution

Eric Medvet · Alberto Bartoli · Andrea De Lorenzo · Fabiano Tarlao

Received: date / Accepted: date

Abstract A long-standing problem in Evolutionary Computation consists in how to choose an appropriate representation for the solutions. In this work we investigate the feasibility of synthesizing a representation *automatically*, for the large class of problems whose solution spaces can be defined by a context-free grammar. We propose a framework based on a form of meta-evolution in which individuals are candidate representations expressed with an ad hoc language that we have developed to this purpose. Individuals compete and evolve according to an evolutionary search aimed at optimizing such representation properties as redundancy, uniformity of redundancy, and locality. We assessed experimentally three variants of our framework on established benchmark problems and compared the resulting representations to human-designed representations commonly used (e.g., classical Grammatical Evolution). The results are promising as the evolved representations indeed exhibit better properties than the human-designed ones. Furthermore, the evolved representations compare favorably with the human-designed baselines in search effectiveness as well. Specifically, we select a *best evolved representation* as the representation with best search effectiveness on a set of learning problems and assess its effectiveness on a separate set of challenging validation problems. For each of the three proposed variants of our framework, the best evolved representation exhibits an average fitness rank on the set of validation problems that is better than the average fitness rank of the human-designed baselines on the same problems.

Keywords Genotype-phenotype mapping, Grammatical Evolution, Meta-evolution

1 Introduction

The choice of the representation of individuals in an Evolutionary Algorithm (EA) has been a central point in Evolutionary Computation since its inception [34, 27].

Department of Engineering and Architecture, University of Trieste, Trieste, Italy
emedvet@units.it, bartoli.alberto@units.it, andrea.delorenzo@units.it, ftarlao@units.it

In many cases, that choice has been guided *a priori* by analogies with biology, in which researchers looked for inspiration while designing their artificial evolutionary systems, on the assumption that Nature eventually succeeded as an effective search method [41]. On the other hand, the impact of the representation on EA search effectiveness has also been widely studied *a posteriori*. In this respect, a common and well established practice consists in investigating any possible relationship between properties of the representation such as, e.g., redundancy and locality [28,20,37], and higher level properties of the EA, e.g., neutrality [5] and evolvability [19].

Despite these efforts, it is fair to claim that both approaches (a priori and a posteriori) failed in clearly determining if and when a representation can guarantee search effectiveness of an EA: copying from Nature does not necessarily lead to a good design [35,9] and there is not a clear view of which properties actually explain a good or a poor search effectiveness [1]. Indeed, the debate is still lively, with arguments ranging from (deemed) misuse of Nature analogies [42] to experimental-based (counter-)evidence [29] and outcomes including guidelines for the design of a representation [41] or directions for future research [34].

A case of particular interest is the one of indirect representations, i.e., those in which each individual is represented by means of a *genotype* and a *phenotype* and a mapping function exists for mapping the former to the latter. Practical motivations for choosing an indirect representation include the possibility of using standard genetic operators—whose behavior is well known—and, at the same time, tackling problems for which specific constraints act on the solutions (i.e., phenotypes). Moreover, indirect representations do have a counterpart in biology, where the form of living organisms depends on the result of a transcription process operating on encoded genetic material. Finally, indirect representation properties can be easily defined and studied both analytically and experimentally based on the mapping function.

One of the most used EAs based on an indirect representation is Grammatical Evolution (GE) [30], a form of grammar-based Genetic Programming (GP) [14], which captures all the three aspects of indirect representations described above. First, GE allows tackling the large class of problems in which constraints on the solutions may be expressed by means of a context-free grammar (CFG). Second, according to its inventors, the overall GE framework was directly inspired by Nature [22]. Third, the properties of the GE genotype-phenotype mapping function have been widely studied [38,37,15]: indeed, those properties eventually served as main goals while designing new GE variants, essentially consisting in new mapping functions which were shown to be more effective than the original approach [12, 16].

In this work, we attempt to provide new insights on the long-standing, undercurrent topic of the choice of the representation. To this end, we consider the broad class of EAs corresponding to grammar-based GP and propose a novel approach for the automatic design of a representation driven by an evolutionary search aimed at optimizing the representation properties. Our proposal thus tries, in a sense, to merge the a priori and a posteriori approaches.

Our contribution consists of the following:

- we define a class of representations in which the genotype is a variable-length bit string and the phenotype is a valid string w.r.t. a user-provided grammar;

– we propose an evolutionary framework for searching the aforementioned space of representations;
– we experimentally investigate the ability of the proposed framework to generate representations whose properties and search effectiveness are better than existing, established representations.

In detail, the class of representations is defined by a genotype-phenotype mapping function template whose variable parts are described with a language which we defined by means of a CFG. The mapping function template and the language are such that: (i) any representation in the resulting class is a valid genotype-phenotype mapping function—i.e., any input bit string is mapped to a valid phenotype in a finite number of steps; (ii) it is possible to express such existing and established representations as the original GE mapping [30] and the recently proposed HGE and WHGE [16]. Having defined the search space in terms of a CFG, we use a grammar-based evolutionary search method (CFG-GP [43]) which we augmented using a diversity promotion strategy in order to improve search effectiveness [18]. For driving the search, we use a fitness function measuring to which degree an individual (i.e., a genotype-phenotype mapping function) exhibits such mapping properties as redundancy, uniformity of redundancy, and locality. We compute those measures on a large amount of mappings obtained from a sample grammar and a set of randomly generated genotypes.

We investigated 3 search variants differing in the fitness definition. We assessed each obtained representation experimentally not only in terms of the mapping properties, but also in terms of higher level EA properties (diversity) and of the search effectiveness achieved on a set of benchmark problems previously used in the literature for assessing GE and its variants. The results are promising as some of the automatically generated representations are better than the existing ones. Specifically, we select a *best evolved representation* as the representation with best search effectiveness on a set of learning problems and assess its effectiveness on a separate set of challenging validation problems. For each of the three proposed variants of our framework, the best evolved representation exhibits an average fitness rank on the set of validation problems that is better than the average fitness rank of the human-designed baselines on the same problems. Although our findings do not imply that automatically-designed representations may fully surrogate carefully human-designed representations, they further corroborate the importance of representation properties and might ignite new research in the novel field of “self-evolving” evolutionary algorithms.

The present paper is an extended version of [17]. We here provide a more detailed description of the language for defining the genotype-phenotype mapping functions and a deeper discussion (with examples) on how existing GE representations may be described in terms of the proposed language. Moreover, we include a broader and deeper experimental analysis, concerning both the possibility of obtaining automatically a representation with better properties and the fact that the obtained representation is also effective when used inside an actual EA.

The remainder of the paper is organized as follows. In Section 2, we briefly survey the state-of-the-art. In Section 3, we provide the background for our work by describing how Grammatical Evolution works. In Section 4, we introduce our genotype-phenotype mapping function template and the related CFG for describing its variable parts. In Section 5, we describe which are the properties we use to

drive the evolution of the mapping function and how we compute them. In Section 6, we present and discuss the results of our experimental evaluation. Finally, in Section 7, we draw the conclusions.

2 Related work

Broadly speaking, our proposal is a form of meta-evolution [8] (also known as hyper-heuristic [25] or self-adaptation [31]), where parts of an EA are chosen or tuned according to a second-level evolutionary search. In most cases, the literature focuses on specific EA parameters which can be optimized, rather than designed from scratch—e.g., mutation and crossover rate in Genetic Semantic Programming [4] or trial vector and control parameters in Differential Evolution [26]. The application of evolutionary computation to evolve (online or offline) components, rather than parameter values, of an EA is instead still believed to be in its infancy [34], in particular for representation and variation operators. For the former, the scarcity of research results may be explained by its hardness, as observed by De Jong [7]: "perhaps the most difficult and least understood area of EA design is that of adapting its internal representation."

Concerning the evolution of operators, the authors of [10] show how they evolved a general purpose mutation operator for Evolutionary Programming which outperforms existing operators on classes of functions (i.e., problems); they also experimentally show that a mutation operator evolved for a specific problem is better than a general purpose evolved operator. A similar goal is aimed at in [6], where a framework for the online evolution of the operators, together with the solutions, is proposed: as in the previously cited work, operators are represented as trees and evolved using GP. Similarly to the present work, [6] also considers other EA properties (diversity) other than search effectiveness as a criterion of analysis.

Concerning the automatic design or adaptation of representations, a proposal is presented in [33], where genotype-phenotype mapping for continuous optimization problems is considered. The authors show, using a proof-of-concept self-adaptation mechanism, that feed-forward neural networks can be used to represent and improve a genotype-phenotype mapping, also for problems of realistic complexity. Similarly to our work, the authors carefully consider redundancy and locality in their analysis.

Another view on automatic design of representation is given by [32], which again addresses the class of real-valued optimization problems: here, the representation is the way in which the real values are encoded using a bit string. With the premise that they focused only on (few) synthetic problems, due to the high computational costs implied by meta-evolution, the authors find that an evolved representation may improve the classical Gray encoding.

Also relevant w.r.t. our work are some proposals concerning grammar-based GP in which the grammar itself is evolved (or improved) online, during the evolution [45, 23]. Despite the evolution of a new, general purpose representation was not among the goals of the cited papers (they rather attempt to discover more knowledge about the problem defined by the user-provided grammar by improving the grammar itself), they somehow demonstrate how a representation can change while still enforcing the problem-specific constraints on the solutions. In

conclusion, to the best of our knowledge, our work is the first attempt of evolving a general purpose representation for a large class of problems, as the one addressable with grammar-based GP.

3 Background: CFG-based representation with bit strings genotype

In this article, we consider a family of EAs with an indirect representation where the *genotype* $\hat{g}$ is a variable-length bit string and the *phenotype* $\hat{p}$ is a string of a language $\mathcal{L}(\mathcal{G})$ defined by a CFG $\mathcal{G} = (N, T, s_0, R)$, where: N is the set of non-terminal symbols, T is the set of terminal symbols (with $T \cap N = \emptyset$), $s_0 \in N$ is the starting symbol, and R is the set of production rules. We do not pose any constraint on components of the EA other than the representation (e.g., selection criteria for reproduction of removal of individuals, initialization). It is worth to note that many significant and widely used variants of GE (beyond its original version) belong to this family of EAs (e.g., πGE [21], HGE, and WHGE [16]).

For completeness of description, we will provide an overview of the original GE proposal in the next section. Then, in Section 4, we will present our approach.

3.1 Grammatical Evolution

In the original GE proposal [30], the genotype is split into substrings of n_{codon} consecutive bits which are then translated into integers using the natural binary encoding—each integer being called *codon*. The value of the parameter n_{codon} is conventionally set to 8. These integers are then used for selecting a production rule and deriving the corresponding symbol to append to the phenotype.

In detail, the procedure for mapping the genotype $\hat{g}$ into a phenotype $\hat{p}$ is iterative and starts with $\hat{p} = s_0$, a counter $i = 0$, and a counter $w = 0$. Then, the following steps are iterated (we denote by r_s the derivation rule for the symbol s).

1. The leftmost non-terminal s in $\hat{p}$ is derived using the jth production rule in r_s (zero-based indexing). The value of j is set to $c_i \mod |r_s|$, i.e., the remainder of the division between the value $c_i = \mathsf{int}(\mathsf{substring}(\hat{g}, i n_{\text{codon}}, (i+1) n_{\text{codon}} - 1))$ of the ith codon (zero-based indexing) and the number $|r_s|$ of derivation options in r_s.
2. The counter i is incremented; if it exceeds the number of codons $\frac{1}{n} \lfloor \frac{|\hat{g}|}{n_{\text{codon}}} \rfloor$, then i is set to 0 and w is incremented. If w exceeds a predefined threshold n_w, then the mapping is aborted.
3. If $\hat{p}$ contains at least one non-terminal, return to step 1, otherwise end.

The re-use of the genotype which is triggered by the first condition at step 2 is called *wrapping*. Counter w ensures that only a maximum of n_w wrappings are allowed; whenever all of them are executed, the mapping is aborted: the individual is then referred to as *invalid* or non-valid and conventionally associated with the worst possible fitness value [15]. Wrapping allows GE mapping to handle the case in which the genotype is consumed before the mapping is ended, i.e., when one or more non-terminals are still present in the phenotype. This case may occur in particular with complex or recursive grammars. An example of mapping will be given in Section 4.3

4 Our approach

4.1 Representation template

We define a *representation template*, i.e., a template of a mapping between a variable-length bit string (genotype) and a string in $\mathcal{L}(\mathcal{G})$ (phenotype), as follows. The mapping is based on the notion of *derivation tree* of a symbol s in $N \cup T$. Such a tree is rooted at s and the children of each non-terminal node $s' \in N$ are symbols (in the proper order) of one of the derivation options for s' in $\mathcal{G}$. The derivation tree is constructed with the algorithm specified below. The mapping occurs in two steps: the input genotype $\hat{g}$ is mapped to a derivation tree of the initial symbol s_0 of $\mathcal{G}$; the corresponding phenotype $\hat{p}$ is then obtained by concatenating, from the left to the right, the leaf nodes of the derivation tree.

Construction of a derivation tree is performed by a function $\textsc{Map}(s, g, d)$, where s is a symbol of $T \cup N$, g is a bit string, and $d \in \mathbb{N}^+ \cup \{0\}$ is a positive number. This function essentially consists in three key steps: (i) choose one derivation option among the ones available for s, by invoking function $\textsc{Choose}()$; (ii) obtain from g several bit strings, by invoking function $\textsc{Divide}()$; (iii) recursively call itself for each symbol in the chosen derivation option, with the symbol, one of the bit strings previously obtained, a counter $d+1$ of recursion depth as input parameters.

Functions $\textsc{Choose}()$ and $\textsc{Divide}()$ are *parameters* of $\textsc{Map}()$ and their signature includes a bit string as input argument. Their domain consists of all the functions that can be defined by a language described in Section 4.2 that we developed. The search space for representations, thus, essentially consists in all the possible implementations for $\textsc{Choose}()$ and $\textsc{Divide}()$.

The mapping of $\hat{g}$ to a derivation tree of s_0 is done by invoking $\textsc{Map}(s_0, \hat{g}, 0)$. The corresponding phenotype $\hat{p}$ is then obtained by concatenating the leaf nodes of the derivation tree.

In details, $\textsc{Map}()$ is shown in Algorithm 1 and works as follows. If s is a terminal node, the tree composed by a single node s is returned by $\textsc{Map}(s, g, d)$, regardless of the values of g and d. Otherwise, the following steps are performed.

1. The derivation rule r_s for the input argument s is obtained by looking up the set R of rules.
2. A vector $\mathbf{e} \in \mathbb{R}^{|r_s|}$ is built, where $|r_s|$ is the number of options in r_s and each element e_j is the product of the *expressiveness* of all the symbols in the jth option of r_s. The expressiveness of a symbol s' (denoted by $\textsc{Expressiveness}(s')$ in Algorithm 1) is a measure of the expressive power of s': we quantify expressiveness with the number of different derivation trees which can be obtained from s'. We limit the counting to derivation trees with a maximum d_{expr} depth (an implicit parameter of $\textsc{Expressiveness}()$ and hence of the representation itself) in order to cope with non-finite languages, for which $\textsc{Expressiveness}(s')$ may be infinite.
3. If the input argument d is greater than or equal to a predefined value $d_{\max}$ (a parameter of the representation), the index i of the chosen rule option is set to the value for which e_i is the lowest in $\mathbf{e}$. Otherwise, i is set to the return value of a function $\textsc{Choose}()$ which takes as input $g, \mathbf{e}, d$ and returns a number that will be used at the next step for choosing one of the options of the derivation rule r_s.

Algorithm 1 The genotype-phenotype recursive mapping function, which is first invoked as $\textsc{Map}(s_0, \hat{g}, 0)$.

```
function Map(s, g, d)
    t ← TreeNode(s)
    if s ∈ N then                                              ▷ s is a non-terminal
        r_s ← RuleFor(s)
        for j ∈ {1, ..., |r_s|} do
            e_j ← ∏_{s' ∈ Symbols(r_s, j)} Expressiveness(s')
        end for
        e ← (e_1, ..., e_{|r_s|})
        if d ≥ d_max then                                      ▷ maximum depth reached
            i ← argmin_{j ∈ {1,...,|r_s|}} e_j
        else
            i ← Choose(g, e, d)
        end if
        (s_1, ..., s_n) ← Symbols(r_s, i)
        for j ∈ {1, ..., n} do
            e_j ← Expressiveness(s_j)
        end for
        e ← (e_1, ..., e_n)
        (g_1, ..., g_m) ← Divide(g, e, d)
        for j ∈ {1, ..., n} do                                 ▷ Append children
            AppendChild(t, Map(s_j, g_j, d + 1))
        end for
    end if
    return t
end function
```

4. The sequence of symbols $s_1, \dots, s_n$ corresponding to the ith option of the r_s rule is obtained. We denote by $\textsc{Symbols}()$ the corresponding grammar look-up function in Algorithm 1; $\textsc{Symbols}()$ is protected, i.e., it works for any i by using $\min(|r_s| - 1, \max(0, \lfloor i \rfloor))$ instead of the original argument i.
5. The vector $\mathbf{e}$ is reset to $(e_1, \dots, e_n)$, where e_j is the expressiveness of s_j obtained at the previous step.
6. A sequence $(g_1, \dots, g_m)$ of bit strings is set to the return value of a function $\textsc{Divide}()$ which takes as input $g, \mathbf{e}, d$ and returns a sequence of bit strings. Each of these bit strings will be used at the next step for constructing subtrees to be appended to the derivation tree being constructed.
7. For each symbol s_j in $s_1, \dots, s_n$, the tree obtained by recursively invoking the $\textsc{Map}(s_j, g_j, d+1)$ is appended to the tree (initially) composed of the only node s, which is eventually returned. While performing this step, in case $j > m$ (i.e., if there are fewer bit strings than symbols to built the children of s), an empty bit string is passed to $\textsc{Map}()$ as g_j.

Regardless of the actual behavior of $\textsc{Choose}()$ and $\textsc{Divide}()$, it can be easily seen that $\textsc{Map}()$ always returns a derivation tree (from which a valid phenotype is then obtained) in a finite number of steps. First, whenever the value of d (which is increased at each recursive invocation) reaches a threshold, the derivation option is chosen as the one with the lowest expressiveness, instead of by using the $\textsc{Choose}()$ function: since in any valid CFG, for any non-terminal symbol, there is at least one derivation option with a finite expressiveness, this guarantees that in a finite number of steps $\textsc{Map}()$ will be invoked with a terminal symbol $s \in T$. Second,

```
⟨mapper⟩ ::= ⟨n⟩ ⟨lg⟩
     ⟨n⟩ ::= ⟨const.n⟩ | ⟨var.n⟩ | ⟨fun.n.g⟩ ( ⟨g⟩ ) | ⟨fun.n.n,n⟩ ( ⟨n⟩ , ⟨n⟩ ) | ⟨fun.n.ln⟩ ( ⟨ln⟩ ) |
             ⟨fun.n.ln,n⟩ ( ⟨ln⟩ , ⟨n⟩ ) | ⟨fun.n.lg⟩ ( ⟨lg⟩ )
    ⟨ln⟩ ::= ⟨var.ln⟩ | ⟨fun.ln.n⟩ ( ⟨n⟩ ) | ⟨fun.ln.n,n⟩ ( ⟨n⟩ , ⟨n⟩ ) | apply ( ⟨fun.n.g⟩ , ⟨lg⟩ )
     ⟨g⟩ ::= ⟨var.g⟩ | ⟨fun.g.g,n⟩ ( ⟨g⟩ , ⟨n⟩ ) | ⟨fun.g.lg,n⟩ ( ⟨lg⟩ , ⟨n⟩ )
    ⟨lg⟩ ::= ⟨fun.lg.g,n⟩ ( ⟨g⟩ , ⟨n⟩ ) | ⟨fun.lg.g,ln⟩ ( ⟨g⟩ , ⟨ln⟩ ) | apply ( ⟨fun.g.g,n⟩ , ⟨ln⟩ , ⟨g⟩ )
⟨const.n⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  ⟨var.g⟩ ::= g
 ⟨var.ln⟩ ::= ln
  ⟨var.n⟩ ::= depth | g.count.r | g.count.rw
⟨fun.n.g⟩ ::= size | weight | weight.r | int
⟨fun.n.n,n⟩ ::= + | - | * | / | %
⟨fun.n.ln⟩ ::= length | max.index | min.index
⟨fun.n.ln,n⟩ ::= get
⟨fun.n.lg⟩ ::= length
⟨fun.ln.n⟩ ::= seq
⟨fun.ln.n,n⟩ ::= repeat
⟨fun.g.g,n⟩ ::= rotate.left | rotate.right | substring
⟨fun.lg.g,n⟩ ::= split | repeat
⟨fun.g.lg,n⟩ ::= get
⟨fun.lg.g,ln⟩ ::= split.w
```

Fig. 1 The CFG $\mathcal{G}_{\text{MAP}}$ defining the language for the CHOOSE() and DIVIDE() functions and hence for an instance of the genotype-phenotype mapping function template defined by MAP().

regardless of the return value of CHOOSE(), a valid derivation is always chosen for s, since only options of r_s are considered.

4.2 Language for the mapping function

Functions CHOOSE() and DIVIDE() are parameters of the mapping function. The space of possible values for these parameters consists of all the functions that may be described by the CFG $\mathcal{G}_{\text{MAP}}$ specified in Figure 1 and discussed below.

Data types available in the language are represented with dedicated non-terminal symbols: ⟨n⟩ represents numbers, ⟨ln⟩ represents sequences of numbers, ⟨g⟩ represents bit strings, ⟨lg⟩ represents sequences of bit strings. Terminal symbols represent numerical constants (0, …, 9), input arguments for CHOOSE() and DIVIDE() (g for bit string g, ln for $\mathbf{e} \in \mathbb{R}^{|r_s|}$, and depth for integer d), and functions (e.g., size returns the length of a bit string, all functions are detailed below and listed in Figure 1).

Names for the non-terminal symbols that begin with var represent input arguments for CHOOSE() and DIVIDE() (terminal symbols ⟨g.count.r⟩ and ⟨g.count.rw⟩ are discussed below). Names for the non-terminal symbols representing functions begin with fun and encode the signature of the function with a simple conventional

rule. For example, ⟨fun.lg.g,n⟩ represents functions whose return value is of type sequence of bit strings (⟨lg⟩) and whose list of input arguments consists of types bit string (⟨g⟩) and number (⟨n⟩).

Terminal symbols that represent functions are described in detail in Table 1. All the functions are type-protected, i.e., they guarantee that a correctly typed value is always returned. Furthermore, we carefully specified the behavior of each function to ensure that all the functions accept every possible value for their arguments, i.e., they never “throw an exception”. This property makes the language particularly suitable for evolving, rather than manually designing, the Choose() and Divide() functions because any string of the language $\mathcal{L}(\mathcal{G}_{\text{Map}})$ defines a pair of the two functions which are syntactically valid and can both be applied to any input arguments.

Terminal symbols g.count.r and g.count.rw may only appear where numbers are expected; they correspond to accessing a global counter, the former reads the value of the counter while the latter reads and then increments its value. By “global” we mean that a single counter is maintained during the execution of both Choose() and Divide(); this counter is set to 0 when the enclosing Map() is first called with parameters $s_0, \hat{g}, 0$. Including a global counter allows to express also genotype-phenotype mapping functions which are not inherently recursive, but can be expressed as recursive function thanks to the counter: the original GE mapping fits this case (see Figure 2a).

Finally, non-terminal symbol ⟨mapper⟩ is the crucial component for expressing an instance of the genotype-phenotype mapping function, i.e., of functions Choose() and Divide(). This symbol can be derived only as a pair ⟨n⟩, ⟨lg⟩: the concatenation of the leaves of the derivation tree rooted at the left child of ⟨mapper⟩ is the function Choose(); similarly, the right child represents the function Divide().

4.3 Examples of mapping

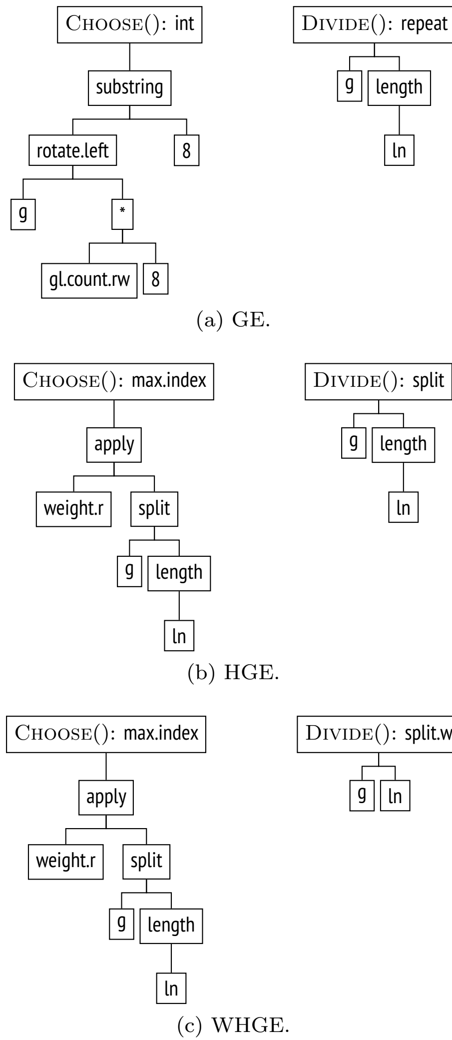
As stated in the introduction, a key feature of our proposal is that it allows expressing such existing and established genotype-phenotype mapping functions as those used in GE, HGE, and WHGE. Indeed, Figure 2 shows, in the form of syntax trees, the Choose() and Divide() functions corresponding to (a slightly improved version of) GE, HGE, and WHGE.

For GE, it can be seen from Figure 2a that Choose() returns the integer value corresponding to the 8 most significant bits of a bit string (through the substring function); that bit string is obtained by rotating the input bit string g for a number of bits given by 8 times the value of the global counter; the global counter is incremented by 1 whenever it is read (through the symbol gl.count.rw). Since Choose() is invoked by Map(), which is itself invoked recursively, this procedure corresponds to deriving each non-terminal by using non-overlapping substrings of g upon each invocation, as in the original GE mapping. Divide() simply returns $|\mathbf{e}|$ copies of the input bit string g. Unlike the original GE mapping, our procedure does not require a mechanism for aborting the mapping when it looks endless (in [30] there is a maximum number of genotype reuses, i.e., wrappings), since that case is addressed by comparing d against $d_{\max}$ in Map().

Concerning HGE and WHGE, it can be seen from Figures 2b and 2c that Choose() returns the index of the chunk of the input bit string with the largest

Table 1 Return value of the functions used in the language defined by the CFG of Figure 1. In each signature, n denotes a number, g a bit string, N a sequence of numbers, G a sequence of bit strings, and f a typed function. Indexes in sequences start from 0. Each function replaces all input numbers that exceed a predefined threshold with that threshold.

Signature	Description
size$(g) \to n$	Size of the bit string g (0, if g is empty).
weight$(g) \to n$	Number of bits set to 1 in the bit string g (0, if g is empty).
weight.r$(g) \to n$	Number of bits set to 1 in the bit string g divided by the size of the bit string (0, if g is empty).
int$(g) \to n$	Integer number represented by the bit string g (0, if g is empty), according the natural binary encoding (as in the standard GE mapping).
+$(n, n) \to n$	Sum of the input numbers.
-$(n, n) \to n$	Subtraction of the second input number from the first one.
*$(n, n) \to n$	Product of the input numbers.
/$(n, n) \to n$	Quotient of division of the first input number by the second one (0 if the second input number is 0).
%$(n, n) \to n$	Remainder of division of the first input number by the second one (0 if the second input number is 0).
length$(N) \to n$	Number of elements in the input sequence (0, if N is empty).
max.index$(N) \to n$	Index in the input sequence of the element containing the maximum number (0, if N is empty; smallest index in case of tie).
min.index$(N) \to n$	Index in the input sequence of the element containing the minimum number (0, if N is empty; smallest index in case of tie).
get$(N, n) \to n$	Number in the nth element of the input sequence (0, if N is empty; number in the first element if $n < 0$ and in the last element if $n \geq \|N\|$).
length$(G) \to n$	Number of elements in the input sequence (0, if G is empty).
seq$(n) \to N$	Sequence $\{0, \dots, n-1\}$ of n elements ($\{0\}$, if $n < 1$).
repeat$(n, n) \to N$	Sequence with length equal to the second input number with all elements containing the first input number (sequence with only one element equal to the first input number, if the second one is ≤ 0).
rotate.left$(g, n) \to g$	Bit string obtained with left circular shift of g by n bits (g if $n \leq 0$; empty bit string, if g is empty).
rotate.right$(g, n) \to g$	Bit string obtained with left circular shift of g by n bits (g if $n \leq 0$; empty bit string, if g is empty).
substring$(g, n) \to g$	Bit string composed of only the first n bits of g (all bits of g if $n \geq \|g\|$; empty bit string, if g is empty or if $n \leq 0$).
split$(g, n) \to G$	Sequence of $\min(n, \|g\|)$ elements in which the ith element contains the ith chunk of bit string g; chunks do not overlap and are constructed so as to minimize the variance of their length (sequence $\{g\}$, if $n \leq 0$; all elements contain an empty bit string, if g is empty).
repeat$(g, n) \to G$	Sequence with n identical elements containing bit string g (with only one element if $n \leq 0$).
get$(G, n) \to g$	Bit string in the nth element of the input sequence (empty bit string, if G is empty; bit string in the first element of G, if $n < 0$, and in the last element, if $n \geq \|N\|$).
split.w$(g, N) \to G$	Sequence with the same number of elements as N, in which the ith element contains the ith chunk of bit string g; chunks do not overlap and their lengths of chunks are proportional to the number contained in the corresponding element of N ($\{g\}$, if $\|N\| = 0$ is empty; all elements are an empty bit string if g is empty).
apply$(f_{g \to n}, G) \to N$	Sequence with the same number of elements as G, in which the ith element contains the result produced by the input function $f_{g \to n}$ on the ith element of G (empty sequence, if N is empty).
apply$(f_{g,n \to g}, N, g) \to G$	Sequence with the same number of elements as N, in which the ith element contains the result produced by the input function $f_{g,n \to g}$ on the ith element of N and on g (empty sequence, if N is empty).



(a) GE.

(b) HGE.

(c) WHGE.

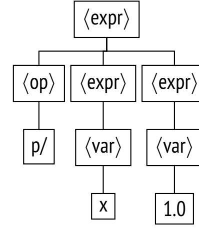
Fig. 2 Choose() and Divide() function trees for the original GE mapping (top), for HGE (center), and for WHGE (bottom).

relative weight (through the weight.r function), after having it split in a number $|\mathbf{e}|$ of chunks of equal length. Divide() works differently for the two mappings: in HGE, it simply splits the input bit string in $|\mathbf{e}|$ chunks of equal length; in WHGE, it splits the input bit string in chunks whose lengths are proportional to the values of $\mathbf{e}$ (through the split.w function). We refer the reader to the original paper for full details [16] about HGE and WHGE.

A step-by-step example of execution of Map() is provided in Figure 3a, with reference to Choose() and Divide() for GE (Figure 2a) and the CFG of Pagie1 problem shown in Figure 5. In particular, Figure 3a shows the input values of the arguments s, g, d, for each recursive invocation of Map() (one row for each invocation) along with the corresponding value for the global counter, while Fig-

s	g	d	GC
⟨expr⟩	1011101001101011000000101	0	0
⟨op⟩	1011101001101011000000101	1	1
p/	1011101001101011000000101	2	1
⟨expr⟩	1011101001101011000000101	1	2
⟨var⟩	1011101001101011000000101	2	3
x	1011101001101011000000101	3	4
⟨expr⟩	1011101001101011000000101	1	4
⟨var⟩	1011101001101011000000101	2	5
1.0	1011101001101011000000101	3	6

(a) Sequence of recursive MAP() calls.

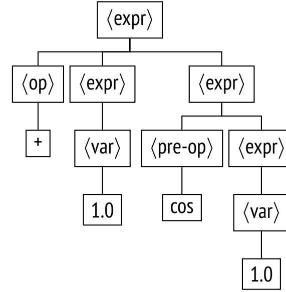


(b) Derivation tree.

Fig. 3 Example of the execution of MAP() on a genotype $\hat{g}$ = 1011101001101011000000101, the grammar of Pagie1 problem shown in Figure 5, and the functions CHOOSE() and DIVIDE() for GE (Figure 2a). The left figure shows the value for the input arguments of MAP() at each recursive invocation, along with the value of the global counter. The resulting derivation tree on the right represents the mathematical expression $1/x$.

s	g	d
⟨expr⟩	111110110100000000010000111101011	0
⟨op⟩	11111011010	1
+	11111011010	2
⟨expr⟩	00000000100	1
⟨var⟩	00000000100	2
1.0	00000000100	3
⟨expr⟩	00111101011	1
⟨pre-op⟩	001111	2
cos	001111	3
⟨expr⟩	01011	2
⟨var⟩	01011	3
1.0	01011	4

(a) Sequence of recursive MAP() calls.



(b) Resulting derivation tree.

Fig. 4 Example of the execution of MAP() on a genotype $\hat{g}$ = 111110110100000000010000111101011, the grammar of Pagie1 problem shown in Figure 5, and the functions CHOOSE() and DIVIDE() for HGE (Figure 2b). The left figure shows the input arguments of MAP() at each recursive invocation (values of g are indented for ease of comprehension). The resulting derivation tree on the right represents the mathematical expression $1 + \cos 1$.

ure 3b illustrates the resulting derivation tree. A further step-by-step example is given in Figure 4a, in this case with reference to CHOOSE() and DIVIDE() for HGE (Figure 2b) and the same CFG as above.

5 Properties-driven evolution

Since we defined the search space of the problem of the automatic design of a representation by means of the CFG $\mathcal{G}_{\text{MAP}}$, we can tackle that problem using any grammar-based GP approach (e.g., GE, πGE, SGE, HGE, WHGE, CFG-GP), provided that we define a fitness function suitable for driving the search. In this work, we want a fitness function able to capture the degree to which a candidate representation $m \in \mathcal{L}(\mathcal{G}_{\text{MAP}})$ exhibits the desired mapping properties.

Among the several properties of indirect representations which have been studied in the literature (see [27] for a comprehensive analysis), we considered redundancy, uniformity of redundancy, and locality—we actually considered non-uniformity and non-locality in order to conform to the semantics of "the lower, the better".

We measure the properties of a representation m basing on how m maps a predefined set G of genotypes to a corresponding set P of phenotypes using a predefined CFG $\mathcal{G}_{\text{learn}}$. That is, for each $\hat{g} \in G$ we construct $\hat{p} = m(\hat{g})$ by concatenating, from the left to the right, the leaf nodes of the derivation tree returned by $\textsc{Map}(\hat{g}, s_0, 0)$, where $\textsc{Map}()$ is the instance of the map function template corresponding to m and s_0 is the starting symbol of $\mathcal{G}_{\text{learn}}$. Having constructed P from G according to m, we quantify the properties of interest as follows.

The *redundancy* of m is measured as $1 - \frac{|P|}{|G|}$, i.e., one minus the ratio between the number $|P|$ of unique phenotypes and the number $|G|$ of unique genotypes.

The *uniformity of redundancy* of m is measured by means of the coefficient of variation of the size of the partitions of G for which every genotype in the partition corresponds to the same phenotype. More formally, let $G_1, \dots, G_{|P|}$ be the partitions of G such that, for each k, $\forall \hat{g}_i, \hat{g}_j \in G_k : m(\hat{g}_i) = m(\hat{g}_j)$, and let $S = |G_1|, \dots, |G_{|P|}|$ contains the sizes of the partitions. The non-uniformity is the coefficient of variation $\frac{\sigma_S}{\mu_S}$ of S, i.e., the ratio between the standard deviation and the mean of the values in S.

Finally, the *locality* of m is measured as the Pearson correlation between the distances among genotypes and distances among phenotypes. More formally, let D^G be the sequence of $\frac{|G|(|G|-1)}{2}$ genotype distances (i.e., $d^G_{i,j} = d^G(\hat{g}_i, \hat{g}_j)$ is the distance between the ith and the jth elements of G, with $j < i$) and let D^P be the corresponding sequence of phenotype distances (i.e., $d^P_{i,j} = d^P(m(\hat{g}_i), m(\hat{g}_j))$). The locality is the Pearson correlation $\text{cor}(D^G, D^P)$ between D^G and D^P. As distances, we used the edit distance for both bit strings and strings of $\mathcal{L}(\mathcal{G}_{\text{learn}})$. The non-locality is measured as $1 - \frac{1+\text{cor}(D^G, D^P)}{2}$, such that it is 0 when genotype and phenotype distances are perfectly correlated ($\text{cor}(D^G, D^P) = 1$), and 1 when they are inversely correlated ($\text{cor}(D^G, D^P) = -1$).

In order to define a criterion for driving the evolutionary search in the space of representations, we considered that, according to many studies, redundancy, locality, and uniformity of redundancy appear to affect the effectiveness of the search in the respective order [22,28,19]. We hence explored three variants for driving the search for a representation: by minimizing redundancy only, by minimizing the sum of redundancy and non-locality, and by minimizing the sum of redundancy, non-locality, and non-uniformity. We denote the respective *search variants* by R, R+NL, and R+NL+NU. In the preliminary version of this work [17] we experimented with variants based on a multi-objective search and the overall findings were in line with those obtained in this work.

In all of our experiments, we used CFG-GP [43] as the evolutionary search algorithm. We augmented the original algorithm with a diversity promotion mechanism which resembles deterministic crowding where children compete with the parents for survival [13,36]. More in detail, a child replace the closest parent in the population only if the former is fitter than the latter. We measure the distance between the child and its parents using the edit distance between the corresponding phenotypes, i.e., strings of $\mathcal{L}(\mathcal{G}_{\textsc{Map}})$, according to the findings of [18].

Table 2 Parameters for the learning and validation runs.

	Learning	Validation
Representation	CFG-GP	Learned representation
Population size	500	500
Pop. initialization	Ramped half-and-half	Random
Generations	50	50
Max depth $d_{\max}$	14	9
Expressiveness depth d_{expr}	N. A.	2
Genotype size	N. A.	256
Crossover rate	0.8	0.8
Crossover operator	CFG-GP crossover	two-points same length
Mutation rate	0.2	0.2
Mutation operator	CFG-GP mutation	bit flip w. $p_{\text{mut}} = 0.01$
Selection for reproduction	tournament with size 3	tournament with size 3
Selection for removal	worst individual	worst individual
Replacement	$m + m$ w. overlapping	$m + m$ w. overlapping

6 Experiments and discussion

6.1 Procedure

We performed an experimental evaluation aimed at answering the two following research questions. RQ1: Can we evolve a representation which is better than the existing ones in terms of redundancy, uniformity of redundancy, and locality? RQ2: Are the evolved representations also effective when used inside an actual EA? We conducted the experiments with a software that we developed and made publicly available[1].

We considered a set L of *learning CFGs* and proceeded according to a *leave-one-out* procedure, as follows.

1. For each CFG $\mathcal{G}_{\text{out}} \in L$ we executed $n_{\text{run}}^{\text{learning}} = 30$ *learning runs* across the CFGs in $L \setminus \mathcal{G}_{\text{out}}$. In each learning run we obtained one *learned representation* as the individual with the best fitness at the last generation. We computed fitness of an individual as the average of the properties of the representation encoded by that individual, computed as described in Section 5 (the value of each property was the average of the values obtained using the CFGs in $L \setminus \mathcal{G}_{\text{out}}$). We used the evolutionary parameters shown in Table 2 (left) for all learning runs.
2. For each learned representation, we measured its properties on the *left-out* CFG $\mathcal{G}_{\text{out}}$, i.e., the one which was not used during the corresponding learning run.
3. For each learned representation, we measured its search effectiveness with $n_{\text{run}}^{\text{validation}} = 5$ *validation runs* on the problem associated with the left-out CFG and on an additional set of *validation problems*. We used the evolutionary parameters shown in Table 2 (right) for all validation runs.

The composition of the set L of learning CFGs and the additional validation problems are specified in the next section. When presenting the results obtained with $L \setminus \mathcal{G}_{\text{out}}$, we will label them with the name of the left-out CFG $\mathcal{G}_{\text{out}}$—e.g.,

[1] `https://github.com/ericmedvet/evolved-ge`

results labelled with Text refer to the representation learned using L without Text as learning CFGs.

In order to measure the properties of representations at step 1, we proceeded as follows. We composed the set of genotypes G with the following steps: (i) we randomly generated a seed set of 10 bit strings, each of 64 bits; and, (ii) for each genotype in the seed set, we obtained other 9 genotypes by iteratively applying the bit-flip mutation operator (with $p_{\text{mut}} = 0.01$). The rationale was to obtain a uniform distribution of distances among the genotypes, useful in particular for measuring of the locality property. We set $d_{\max} = 9$ (see Algorithm 1).

In order to measure the properties of representations at step 2, we proceeded in the same way except that we used a seed set with bit strings of 256 bits (rather than bit strings of 64 bits).

We executed steps 2 and 3 also for 4 GE approaches proposed earlier in the literature that can be used as *baseline*: GE [30], HGE and WHGE [16], and GE_{opt}. GE_{opt} denotes a version of GE where the size n_{codon} of the codon is set depending on the CFG, instead of being statically set to $n_{\text{codon}} = 8$: in particular, in GE_{opt}, $n_{\text{codon}} = \max_{r_s \in R} \lceil \log_2 |r_s| \rceil$, i.e., the lowest size which allows to express all the options of the largest derivation rule. We included this modified version of GE, because it should have lower redundancy than the original GE: indeed, with the same aim a similar choice has been made in previous applications of GE to practical problems, e.g., in [3].

We emphasize that all the baseline are *human-designed*, i.e., they are the result of dedicated research efforts.

6.2 Benchmark problems

We considered the following benchmark problems—the corresponding CFGs are shown in Figure 5. We included symbolic regression, Boolean, and synthetic problems: some of them has been recommended as standard benchmarks for GP performance evaluation [44].

- Keijzer6 [11]: symbolic regression of the function $f(x) = \sum_{i=1}^{x} \frac{1}{i}$ on 50 points evenly spaced in $[1, 50]$. The fitness is given by the average absolute error on the points.
- KLandscapes-5 and KLandscapes-7: K Landscapes with $k = 5$ and $k = 7$ (harder), a tunable, GP-specific benchmark [40] for which we built a CFG for expressing the corresponding trees. We here express the fitness of a solution t as $f(t) = 1 - f_0(t)$, where $f_0(t)$ is the original fitness function described in [40], in order to make it consistent with the other problems, for which the lower the fitness, the better.
- MOPM-3: Multiple outputs parallel 3-bit multiplier. The fitness is given by the rate of the number of errors among all the input cases.
- Nguyen7 [39]: symbolic regression of the function $f(x) = \log(x+1) + \log(x^2+1)$ on a set of 20 points uniformly sampled in $[0, 2]$. The fitness is given by the average absolute error on the points.
- Pagie-1 [24]: symbolic regression of the function $f(x, y) = \frac{1}{1+x^{-4}} + \frac{1}{1+y^{-4}}$ on a set of 125 points resulting from 25 values evenly spaced in $[-5, 5]$ for both x and y. The fitness is given by the average absolute error on the points.

Keijzer6

⟨expr⟩ ::= ⟨op⟩ ⟨expr⟩ ⟨expr⟩ | pre-op ⟨expr⟩ | ⟨var⟩
⟨op⟩ ::= + | *
⟨pre-op⟩ ::= uminus | 1/ | sqrt
⟨var⟩ ::= x

KLandscapes-5 (and -7)

⟨N⟩ ::= ⟨n⟩ ⟨N⟩ ⟨N⟩ | ⟨t⟩
⟨n⟩ ::= n0 | n1
⟨t⟩ ::= t0 | t1 | t2 | t3

MOPM-3

⟨o⟩ ::= ⟨e⟩ ⟨e⟩ ⟨e⟩ ⟨e⟩ ⟨e⟩ ⟨e⟩
⟨e⟩ ::= .or ⟨e⟩ ⟨e⟩ | .xor ⟨e⟩ ⟨e⟩ | .and ⟨e⟩ ⟨e⟩ |
.and1not ⟨e⟩ ⟨e⟩ | ⟨v⟩
⟨v⟩ ::= v1.1 | v1.2 | v1.3 | v2.1 | v2.2 | v2.3

Nguyen7

⟨expr⟩ ::= ⟨op⟩ ⟨expr⟩ ⟨expr⟩ | pre-op ⟨expr⟩ | ⟨var⟩
⟨op⟩ ::= + | - | p/ | *
⟨pre-op⟩ ::= sin | cos | exp | plog
⟨var⟩ ::= x | 1.0

Pagie1

⟨expr⟩ ::= ⟨op⟩ ⟨expr⟩ ⟨expr⟩ | pre-op ⟨expr⟩ | ⟨var⟩
⟨op⟩ ::= + | - | p/ | *
⟨pre-op⟩ ::= sin | cos | exp | plog
⟨var⟩ ::= x | y | 1.0

Parity-3

⟨e⟩ ::= .or ⟨e⟩ ⟨e⟩ | .and ⟨e⟩ ⟨e⟩ | .not ⟨e⟩ | ⟨v⟩
⟨v⟩ ::= v1 | v2 | v3

Text

⟨text⟩ ::= ⟨sentence⟩ ⟨text⟩ | ⟨sentence⟩
⟨sentence⟩ ::= ⟨Word⟩ ␣ ⟨sentence⟩ | ⟨word⟩ ␣ ⟨sentence⟩ |
⟨word⟩ ⟨punct⟩
⟨word⟩ ::= ⟨letter⟩ ⟨word⟩ | ⟨letter⟩
⟨Word⟩ ::= ⟨Letter⟩ ⟨word⟩
⟨letter⟩ ::= ⟨vowel⟩ | ⟨consonant⟩
⟨vowel⟩ ::= a | e | i | o | u
⟨consonant⟩ ::= b | c | d | ... | z
⟨Letter⟩ ::= ⟨Vowel⟩ | ⟨Consonant⟩
⟨Vowel⟩ ::= A | E | I | O | U
⟨Consonant⟩ ::= B | C | D | ... | Z
⟨punct⟩ ::= ! | ? | .

Fig. 5 CFGs of the benchmark problems: in the symbolic regression problems, p/ and plog are the protected versions of the division and the logarithm, respectively.

- Parity-3: 3-bit parity. The fitness is given by the rate of the number of errors among all the input cases.
- Text [15]: generation of the target string Hello world!; the fitness is given by the edit distance between the string corresponding to the solution and the target string.

We used the CFGs of KLandscapes-5, Pagie1, Parity-3, and Text as learning CFGs (all the problems have been used as validation problems, as discussed in the previous section). We selected this subset of problems because the corresponding CFGs well represent those of the validation problems. In particular, it can be noted from Figure 5 that the CFG of Text is more complex than the other CFGs, both in the depth of the dependencies among non-terminals and in the number of production rules for each non-terminal. This allows for a better investigation about the ability of the proposed approach in learning representations which exhibit good properties over different kinds of problems.

6.3 Results and discussion: representation properties

This subsection illustrates the results related to RQ1, i.e., can we evolve a representation which is better than the existing ones in terms of redundancy, uniformity of redundancy, and locality?

Table 3 shows the property values for the evolved representations and for the baselines. In particular, it contains 7 groups of rows: 3 corresponding to the proposed search variants (R, R+NL, and R+NL+NU) and 4 corresponding to the human-designed GE approaches used as baseline. Each group of rows describes

Table 3 Representation properties computed on the learning CFGs $L \setminus \mathcal{G}_{\text{out}}$ and on the left-out CFG $\mathcal{G}_{\text{out}}$. For the learned representations, values are averaged across the 30 learning runs.

		On learning CFGs $L \setminus \mathcal{G}_{\text{out}}$			On left-out CFG $\mathcal{G}_{\text{out}}$		
	$\mathcal{G}_{\text{out}}$	R	NL	NU	R	NL	NU
R	KLandscapes-5	0			0.317	0.567	0.847
	Pagie1	0.001			0.005	0.494	0.045
	Parity-3	0			0.303	0.805	0.185
	Text	0.001			0.336	1	0.158
	Average	0			0.242	0.719	0.311
R+NL	KLandscapes-5	0.019	0.545		0.245	0.346	0.832
	Pagie1	0.034	0.545		0.097	0.342	0.344
	Parity-3	0.03	0.568		0.182	0.736	0.172
	Text	0.038	0.321		0.374	1	0.455
	Average	0.03	0.495		0.225	0.606	0.451
R+NL+NU	KLandscapes-5	0.011	0.613	0.03	0.133	0.442	0.536
	Pagie1	0.006	0.629	0.023	0.009	0.455	0.07
	Parity-3	0.009	0.611	0.031	0.068	0.896	0.063
	Text	0.012	0.417	0.045	0.413	1	0.189
	Average	0.009	0.567	0.032	0.156	0.698	0.214
GE	KLandscapes-5				0.998	1	0
	Pagie1				0.998	1	0
	Parity-3				0.998	1	0
	Text				0.978	1	2.529
	Average				0.993	1	0.632
GE_{opt}	KLandscapes-5				0.963	0.39	1.39
	Pagie1				0.928	0.415	2.167
	Parity-3				0.805	0.441	3.386
	Text				0.95	1	1.201
	Average				0.911	0.561	2.036
HGE	KLandscapes-5				0.833	0.447	3.472
	Pagie1				0.713	0.451	3.988
	Parity-3				0.57	0.39	1.86
	Text				0.515	1	0.74
	Average				0.658	0.572	2.515
WHGE	KLandscapes-5				0.84	0.468	3.394
	Pagie1				0.645	0.449	4.424
	Parity-3				0.308	0.425	2.243
	Text				0.498	1	1.195
	Average				0.573	0.585	2.814

the results obtained with a different set $L \setminus \mathcal{G}_{\text{out}}$ of learning CFGs (step 1 in Section 6.1); the last row in each group is the average across all sets. The values for the representation properties are provided separately for the learning CGFs $L \setminus \mathcal{G}_{\text{out}}$ (only for the learned representations) and for the left-out CFG $\mathcal{G}_{\text{out}}$.

It can be seen that redundancy and non-uniformity are much better for the evolved representations than for the baselines, while non-locality is slightly worse. Interestingly, though, the evolved representations exhibit better locality than the

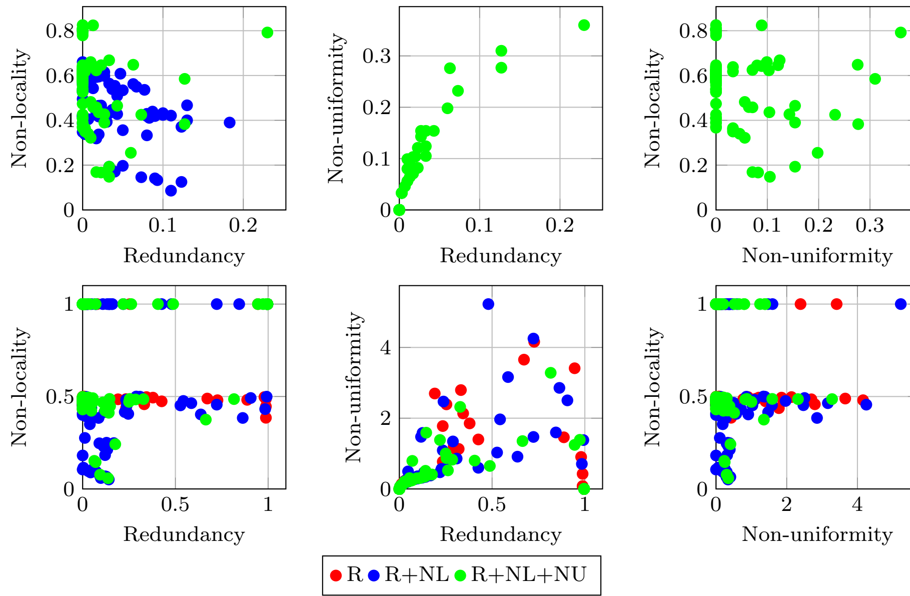


Fig. 6 Pair-wise relationships among the representation properties measured on the learning CFGs (available only for variants R+NL and R+NL+NU) and on the left-out CFG (below), one mark for each learning run.

original GE approach. It can also be seen that, for the evolved representations, the properties that can be observed on $L \setminus \mathcal{G}_{\text{out}}$ tend to be weakly to the properties on $\mathcal{G}_{\text{out}}$. Furthermore, the best values are overall those obtained with R+NL+NU.

Further insights on the evolved representations can be obtained from Figure 6 (upper row), which shows the pair-wise relationships among the representation properties for each single learning run (available only for variants R+NL and R+NL+NU). As it turns out, optimizing redundancy and non-uniformity seems to be easier than optimizing either of these properties and non-locality: it can be seen from the middle plot that redundancy and non-uniformity are well correlated, whereas redundancy and non-locality (leftmost plot) and non-uniformity and non-locality (rightmost plot) exhibit a remarkably lower correlation. Moreover, there seems to be a trade-off between non-locality and each one of the other two properties. These observations are corroborated by Figure 6 (bottom row), which provides the representation properties for each learned representation on the corresponding left-out CFG (available for the three variants R, R+NL, and R+NL+NU). It can be seen that there are several runs where non-locality is at its maximum value while the other property spans its full range of values (redundancy in the leftmost plot and non-uniformity in the rightmost plot). On the other hand, it can be seen that the vast majority of runs are near the origin in the middle plot.

Figure 7 illustrates the value of each fitness objective during the evolution. We observe that all the fitness objectives tend to improve in all the variants. However, non-locality remains to a larger value and tends to decrease much more slowly than redundancy and non-uniformity.

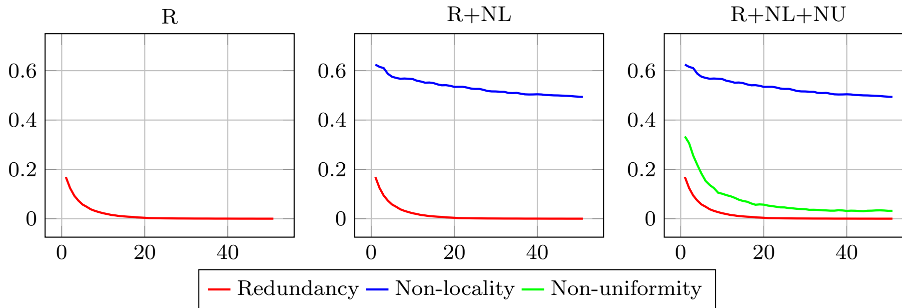


Fig. 7 Values for the three objectives of the fitness (i.e., the properties) during the evolution, averaged across the 30 learning runs and the 4 sets of learning CFGs.

6.4 Results and discussion: search effectiveness

In order to answer RQ2 (i.e., are the evolved representations also effective when used inside an actual EA?), we then examined the search effectiveness of the evolved representations. That is, we examined the fitness values for each of the validation problems when solved with the evolved representations and when solved with the baseline representations.

We analyzed the fitness that can be achieved with learned representations *on the average* and with the *best learned representation* for each variant. We determined the best learned representations based on the search effectiveness on the problems corresponding to the learning CFGs, as follows: (i) we considered all the $3 \times 4 \times 30$ (search variants, sets of learning CFGs, learning runs) learned representations and the 4 human-designed representations; (ii) for each validation problem, we considered the final fitness values obtained in the 5 validation runs by each representation and determined the average percentile rank of that representation; (iii) for each learned representation, we computed the average percentile rank on the validation problems corresponding to the learning CFGs (i.e., KLandscapes-5, Pagie1, Parity-3, and Text); (iv) finally, for each search variant, we chose the representation with the lowest rank at the previous step as the best learned representation for that variant. We emphasize that the best learned representation may be identified based solely on its search effectiveness on the learning CFGs, without any knowledge of its effectiveness on the additional validation problems.

Table 4 shows the resulting fitness percentile rank, computed as described above and cast in interval $[0, 1]$. The first group of rows shows the average rank obtained with the three search variants, while the second group of rows shows the average rank of the best representation obtained with each variant—the *same* representation across *all* validation problems.

The main finding is that the best representation learned with R+NL exhibits a ranking (averaged across the 8 validation problems) better than *all* the human-designed representations. We believe this is a very significant result. The two other best representations, those obtained with R and R+NL+NU, also exhibit very good behavior as their ranking is better than 3 of the 4 baselines (all the baselines except for WHGE). It is also interesting to remark that the ranking of the average evolved representations is better than the ranking of the original GE representation, for all the three variants R, R+NL, and R+NL+NU.

Table 4 Percentile rank in $[0, 1]$ of the final best fitness achieved with the learning representations and the baselines.

		Keijzer6	KLand.-5	KLand.-7	MOPM-3	Nguyen7	Pagie1	Parity-3	Text	Avg.
Avg.	R	0.523	0.518	0.524	0.491	0.571	0.592	0.264	0.576	0.508
	R+NL	0.487	0.411	0.417	0.404	0.425	0.414	0.26	0.413	0.404
	R+NL+NU	0.511	0.526	0.55	0.414	0.508	0.5	0.262	0.501	0.472
Best	R	0.077	0.111	0.045	0.066	0.179	0.085	0	0.022	0.075
	R+NL	0.04	0.005	0.073	0.017	0.13	0.169	0	0.037	0.061
	R+NL+NU	0.106	0.152	0.111	0.025	0.156	0.032	0	0.015	0.075
	GE	0.441	0.997	0.997	0.294	0.705	0.637	0.987	0.123	0.647
	GE_{opt}	0.07	0.89	0.895	0.015	0.099	0.194	0	0.037	0.282
	HGE	0.095	0.147	0.031	0.09	0.29	0.31	0	0.006	0.131
	WHGE	0.047	0.147	0.013	0.041	0.094	0.145	0.051	0.01	0.069

When comparing the three search variants, the figures of Table 4 show that the best option is R+NL, though the differences with R and R+NL+NU are not particularly evident. Optimizing for both redundancy and non-locality, thus, appears to be the better choice from the point of view of search effectiveness. It is worth to note, however, that the comparison previously made in terms of representation properties suggested that R+NL+NU delivers better representations than the two other variants (see Table 3).

Table 5 provides the raw fitness values obtained at the end of the evolution. The Table contains 4 groups of rows: 3 corresponding to the proposed search variants (R, R+NL, and R+NL+NU) and the last one corresponding to the human-designed baselines. Each group of rows corresponding to a proposed search variant contains: one row for each set of learning CFGs (as in Table 3, the label indicates the left-out CFG $\mathcal{G}_{\text{out}}$) which shows the average across the 30 learned representations; one row showing the average across all the sets of learning CFGs, i.e., across 4×30 learned representations; and, one row showing the values obtained with the best learned representation for that search variant.

The foremost finding is that the best learned representation with the R+NL search variant (i) is at least as effective as the most effective human-designed representation on 5 out of 8 problems (Keijzer6, KLandscapes-5, Nguyen-7, and Parity-3, and (ii) outperforms the human-designed GE representation in *all* the problems. The best learned representations with R and R+NL+NU, instead, outperform the most effective human-designed representation on 3 problems: KLandscapes-5, Pagie1, and Parity-3.

On the other hand, the baselines tend to perform better than the average fitness of the learned representations. Interestingly, though, for both K-Landscape-5 and K-Landscape-7, the average fitness of the learned representations is better than GE and GE_{opt}. We speculate that the peculiar fitness landscape of this benchmark, in which the larger the value of k, the stronger the interaction among parts of the solution [40], could highlight different aspects of the representations than the other problems.

Table 5 Best fitness at the end of the evolution (averaged across the 5 validation runs). Values for each search variant are averaged across the 30 learning runs. Rows "Average" and "Best" correspond, respectively and for each search variant, to the average across the learned representations and to the best learned representation .

	$\mathcal{G}_{\text{out}}$	Keijzer6	KLand.-5	KLand.-7	MOPM-3	Nguyen7	Pagie1	Parity-3	Text
R	KLandscapes-5	$\approx 10^8$	0.61	0.56	0.82	0.19	0.73	0.13	66.91
	Pagie1	$\approx 10^{12}$	0.61	0.57	0.88	0.27	1.19	0.25	56.91
	Parity-3	$\approx 10^{14}$	0.62	0.58	0.89	0.27	1.14	0.33	57.99
	Text	$\approx 10^{11}$	0.62	0.57	0.92	0.31	1.48	0.34	105.3
	Average	$\approx 10^{13}$	0.61	0.57	0.88	0.26	1.14	0.26	71.78
	Best	0.13	0.43	0.42	0.75	0.05	0.12	0	7.4
R+NL	KLandscapes-5	NaN	0.49	0.53	0.83	0.11	0.35	0.17	32.12
	Pagie1	NaN	0.56	0.58	0.82	12.37	16.4	0.23	36.18
	Parity-3	NaN	0.57	0.54	0.89	0.36	0.86	0.35	58.07
	Text	NaN	0.57	0.6	0.85	0.18	0.68	0.29	32.92
	Average	NaN	0.55	0.56	0.85	3.26	4.57	0.26	39.82
	Best	0.11	0.14	0.38	0.72	0.03	0.16	0	7.8
R+NL+NU	KLandscapes-5	NaN	0.59	0.6	0.81	0.18	0.45	0.14	40.1
	Pagie1	NaN	0.62	0.68	0.85	0.17	0.59	0.29	38.45
	Parity-3	$\approx 10^4$	0.58	0.54	0.88	5.02	6.1	0.3	71.5
	Text	NaN	0.66	0.64	0.88	11.82	9.5	0.32	82.13
	Average	NaN	0.61	0.62	0.85	4.3	4.16	0.27	58.05
	Best	0.16	0.47	0.45	0.73	0.04	0.08	0	7.2
GE		1.25	1	1	0.82	0.36	0.84	0.5	9.2
GE_{opt}		0.14	0.73	0.89	0.69	0.03	0.18	0	7.8
HGE		0.15	0.58	0.41	0.76	0.08	0.25	0	6.6
WHGE		0.12	0.58	0.37	0.73	0.03	0.15	0.02	5.2

We analyzed the statistical significance of the results shown in Table 5 by performing a Mann-Whitney U-Test on the final best fitness obtained with the three best learned representations (one for each search variant) and the human-designed representations. The results are shown, in terms of p-values, in Table 6. It can be seen that the difference between R+NL and GE is statistically significant ($p < 0.05$) in all the problems. Moreover, it can also be seen that none of the human-designed representation is significantly better than a best learned representation.

Figure 8 plots the best fitness during the evolution in the validation runs, for the best learned representations and the human-designed representations; each curve plots the average values across the 5 validation runs. It can be seen that the fitness indeed tends to improve during the evolution and that the curves for R, R+NL, R+NL+NU tend to be similar to those of the human-designed representations HGE and WHGE. It can also be seen that the GE standard representation is unable to favor any improvement of the fitness during the evolution for most of the problems: we think this limitation is due to the combination of the codon size and the short genotype length. As a confirmation, it can be seen that GE_{opt} does

Table 6 p-values returned by the Mann-Whitney U-Test on the final best fitness, with the alternative hypothesis that the values obtained with the 1st representation are less than those obtained with the 2nd representation (one sided). For the learned representations, only the 3 best ones (see text) are considered. p-values lower than 0.05 are highlighted.

1st	>2nd	Keijzer6	KLand.-5	KLand.-7	MOPM-3	Nguyen7	Pagie1	Parity-3	Text
R	>R+NL	0.895	0.997	0.194	0.989	0.896	**0.006**	1	0.135
R	>R+NL+NU	0.338	0.333	0.059	0.994	0.662	0.99	1	0.784
R	>GE	**0.004**	**0.005**	**0.003**	**0.005**	**0.004**	**0.004**	**0.004**	**0.004**
R	>GE_{opt}	0.662	**0.006**	**0.005**	0.988	0.896	0.087	1	0.135
R	>HGE	0.5	**0.036**	0.617	0.102	0.148	**0.006**	1	0.978
R	>WHGE	0.895	**0.036**	0.998	0.902	0.942	0.148	0.251	0.973
R+NL	>R	0.148	**0.005**	0.86	**0.02**	0.147	0.997	1	0.911
R+NL	>R+NL+NU	**0.014**	**0.01**	0.147	0.094	0.458	0.997	1	0.973
R+NL	>GE	**0.004**	**0.004**	**0.004**	**0.005**	**0.004**	**0.004**	**0.004**	**0.004**
R+NL	>GE_{opt}	0.148	**0.006**	**0.006**	0.833	0.853	0.338	1	0.56
R+NL	>HGE	**0.018**	**0.003**	0.958	**0.009**	0.104	**0.006**	1	0.995
R+NL	>WHGE	0.232	**0.003**	0.955	0.16	0.799	0.583	0.251	0.987
R+NL+NU	>R	0.735	0.741	0.963	**0.012**	0.417	**0.018**	1	0.3
R+NL+NU	>R+NL	0.992	0.995	0.896	0.937	0.624	**0.006**	1	**0.046**
R+NL+NU	>GE	**0.004**	**0.005**	**0.004**	**0.004**	**0.004**	**0.004**	**0.002**	**0.004**
R+NL+NU	>GE_{opt}	0.982	**0.006**	**0.006**	0.948	0.898	0.071	1	**0.046**
R+NL+NU	>HGE	0.853	0.304	0.987	**0.004**	0.072	**0.006**	1	0.965
R+NL+NU	>WHGE	0.992	0.304	0.997	0.292	0.928	0.071	0.212	0.967
GE	>R	0.998	0.998	0.999	0.997	0.998	0.998	0.999	0.998
GE	>R+NL	0.998	0.998	0.998	0.997	0.998	0.998	0.999	0.998
GE	>R+NL+NU	0.998	0.998	0.998	0.998	0.998	0.998	0.999	0.998
GE	>GE_{opt}	0.998	0.997	0.998	0.997	0.998	0.998	0.999	0.998
GE	>HGE	0.998	0.999	0.999	0.997	0.998	0.998	0.996	0.998
GE	>WHGE	0.998	0.999	0.998	0.997	0.998	0.998	0.999	0.997
GE_{opt}	>R	0.417	0.997	0.997	**0.02**	0.147	0.942	1	0.911
GE_{opt}	>R+NL	0.895	0.997	0.997	0.226	0.2	0.735	1	0.56
GE_{opt}	>R+NL+NU	**0.03**	0.997	0.997	0.08	0.145	0.954	1	0.973
GE_{opt}	>GE	**0.004**	**0.005**	**0.004**	**0.006**	**0.004**	**0.004**	**0.004**	**0.004**
GE_{opt}	>HGE	0.105	0.998	0.997	**0.009**	**0.018**	0.072	1	0.995
GE_{opt}	>WHGE	0.942	0.998	0.997	0.079	0.735	0.735	0.251	0.987
HGE	>R	0.583	0.98	0.5	0.933	0.895	0.997	1	**0.038**
HGE	>R+NL	0.989	0.998	0.065	0.995	0.929	0.997	1	**0.009**
HGE	>R+NL+NU	0.201	0.78	**0.022**	0.998	0.953	0.997	1	0.061
HGE	>GE	**0.004**	**0.003**	**0.003**	**0.005**	**0.004**	**0.004**	**0.014**	**0.004**
HGE	>GE_{opt}	0.928	**0.004**	**0.005**	0.995	0.989	0.953	1	**0.009**
HGE	>WHGE	0.994	1	0.998	0.988	0.982	0.989	0.376	0.933
WHGE	>R	0.148	0.98	**0.004**	0.145	0.087	0.895	0.868	**0.043**
WHGE	>R+NL	0.827	0.998	0.069	0.888	0.265	0.5	0.868	**0.022**
WHGE	>R+NL+NU	**0.014**	0.78	**0.006**	0.778	0.105	0.954	0.885	0.052
WHGE	>GE	**0.004**	**0.003**	**0.003**	**0.005**	**0.004**	**0.004**	**0.003**	**0.005**
WHGE	>GE_{opt}	0.087	**0.004**	**0.006**	0.948	0.338	0.338	0.868	**0.022**
WHGE	>HGE	**0.011**	1	**0.004**	**0.021**	**0.03**	**0.018**	0.829	0.099

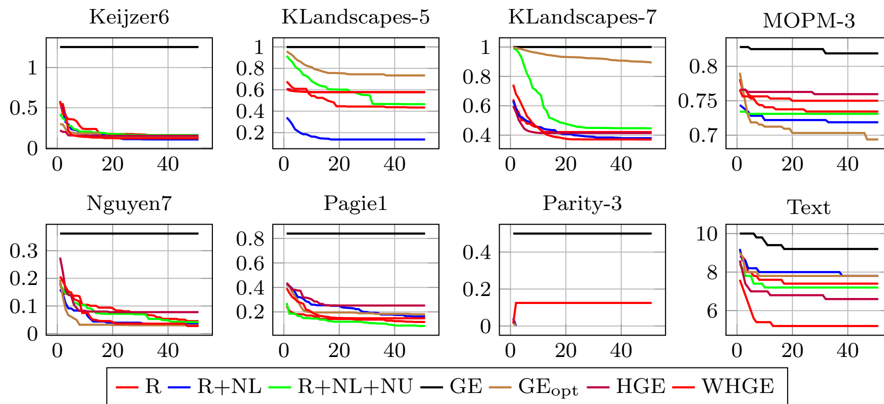


Fig. 8 Average (across the 5 validation runs) best fitness during the evolution for the 3 best learned representations (see text) and the baselines, for each problem.

not exhibit this behavior and is, instead, the best performing representation on MOPM-3.

Finally, in order to gain further insights into the evolved representations, we analyzed the populations of the validation runs in terms of diversity at the level of phenotype. We measured diversity as the rate of unique individuals in the initial and in the final populations (Table 7 and Table 8, respectively). The structure of each of the two tables is the same as the one of Table 4.

It can be seen that, in the initial populations, all the proposed search variants tend to lead to higher phenotypic diversity than the baselines, which is generally beneficial to search effectiveness. We believe the higher initial diversity for the learned representations is a further indication of the soundness of our proposal, because: (i) the learned representations were designed for minimizing redundancy; and, (ii) phenotypic diversity in the initial population strongly depends on redundancy, due to the random initialization of the population.

Concerning the final populations, the best learned representations and the baselines exhibit similar diversity. The fact that the average values for the learned representations tend to be higher, coupled with the fact that the average final fitness tend to be worse than the baselines, could indicate that the selection pressure for the learned representations was not strong enough. In this respect, it could be interesting to explore the behavior of the evolved representations in the presence of strategies aimed at ensuring adequate selection pressure while preserving phenotypic diversity [2,36].

7 Concluding remarks and future work

In the attempt of providing new insights into the long-standing problem of choosing the most appropriate representation for an EA, we have presented a method for the automatic synthesis of a representation for the large class of problems whose solutions spaces can be defined by a CFG. We have defined a representation template for genotype-phenotype mapping, in the form of a recursive function with two parameter functions that can be described using an ad hoc language that we

Table 7 Initial phenotype diversity for each validation problem (averaged across the 5 validation runs).

		Keijzer6	KLand.-5	KLand.-7	MOPM-3	Nguyen7	Pagie1	Parity-3	Text	Avg.
Avg.	R	0.974	0.974	0.976	0.36	0.976	0.98	0.608	0.962	0.851
	R+NL	0.97	0.982	0.982	0.994	0.978	0.98	0.994	0.978	0.982
	R+NL+NU	0.978	0.976	0.974	0.748	0.98	0.974	0.996	0.96	0.948
Best	R	0.964	0.826	0.834	0.882	0.974	0.974	0.925	0.968	0.918
	R+NL	0.762	0.932	0.93	0.952	0.848	0.882	0.868	0.97	0.894
	R+NL+NU	0.916	0.75	0.764	0.928	0.948	0.956	0.948	0.982	0.899
	GE	0	0.01	0.01	0.048	0.01	0	0.01	0.048	0.017
	GE_{opt}	0.154	0.192	0.222	0.784	0.138	0.184	0.455	0.19	0.286
	HGE	0.338	0.752	0.748	0.84	0.368	0.344	0.64	0.834	0.605
	WHGE	0.414	0.818	0.82	0.884	0.436	0.43	0.83	0.802	0.679

Table 8 Final phenotype diversity for each validation problem (averaged across the 5 validation runs).

		Keijzer6	KLand.-5	KLand.-7	MOPM-3	Nguyen7	Pagie1	Parity-3	Text	Avg.
Avg.	R	0.016	0.076	0	0.992	0.03	0.028	0.992	0.002	0.267
	R+NL	0.166	0.858	0.17	0.122	0.41	0.238	0.984	0	0.368
	R+NL+NU	0.002	0.922	0.146	0.448	0.018	0.07	0.986	0.11	0.338
Best	R	0	0.002	0	0.006	0.004	0	0.925	0.006	0.097
	R+NL	0	0	0	0.002	0	0	0.76	0.006	0.079
	R+NL+NU	0	0.02	0	0.004	0	0	0.948	0.004	0.122
	GE	0	0	0	0.01	0	0	0.004	0.002	0.002
	GE_{opt}	0	0	0.008	0.198	0.004	0.006	0.408	0.008	0.071
	HGE	0	0.23	0	0.522	0	0	0.64	0.024	0.139
	WHGE	0	0.716	0	0.578	0	0	0.832	0.004	0.266

have developed for this purpose. Our representation template is expressive enough to describe the classic GE mapping and more recent proposals such as HGE and WHGE; at the same time, our template is much more general and ensures that any instance representation is valid, i.e., it maps any input variable-length bit string to a string of the user-provided language in a finite number of steps. We used CFG-GP to evolve the representations expressed by our template and optimize 3 crucial representation properties: redundancy, non-locality, and non-uniformity.

We executed a number of experiments and carefully assessed the evolved representations using human-designed representations proposed earlier in the literature, i.e., GE, HGE, and WHGE. The results show that our proposal indeed allows automatically designing a representation which exhibits better properties than the human-designed ones. Specifically, the learned representations tend to exhibit better redundancy and non-uniformity than all the human-designed baselines and better locality than the original GE proposal.

The most relevant result, though, is that our proposal synthesized automatically representations whose search effectiveness compare favorably to the human-designed baselines. Specifically, for one of the proposed search variants (R+NL), the best performing representation on the learning problems turned out to exhibit an average fitness rank on a set of validation problems much larger from those of the learning problems that was *better* than the average fitness rank of *all* the human-designed baselines on the same problems. We believe this is a significant result demonstrating the potential of the proposed approach.

We hope that our work might open new research perspectives in the young field of automatic design of representations.

References

1. Altenberg, L.: Probing the axioms of evolutionary algorithm design: Commentary on "On the mapping of genotype to phenotype in evolutionary algorithms" by Peter A. Whigham, Grant Dick, and James Maclaurin. Genetic Programming and Evolvable Machines **18**(3), 363–367 (2017). DOI 10.1007/s10710-017-9290-3
2. Bartoli, A., De Lorenzo, A., Medvet, E., Tarlao, F.: Learning text patterns using separate-and-conquer genetic programming. In: European Conference on Genetic Programming, pp. 16–27. Springer (2015)
3. Bartoli, A., De Lorenzo, A., Medvet, E., Tarlao, F.: Syntactical similarity learning by means of grammatical evolution. In: Parallel Problem Solving from Nature – PPSN XIV: 14th International Conference, Edinburgh, UK, September 17-21, 2016, Proceedings, pp. 260–269. Springer International Publishing, Cham (2016). DOI 10.1007/978-3-319-45823-6_24
4. Castelli, M., Manzoni, L., Vanneschi, L., Silva, S., Popovič, A.: Self-tuning geometric semantic genetic programming. Genetic Programming and Evolvable Machines **17**(1), 55–74 (2016)
5. Correia, M.B.: A study of redundancy and neutrality in evolutionary optimization. Evolutionary computation **21**(3), 413–443 (2013)
6. Cruz-Salinas, A.F., Perdomo, J.G.: Self-adaptation of genetic operators through genetic programming techniques. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '17, pp. 913–920. ACM, New York, NY, USA (2017). DOI 10.1145/3071178.3071214
7. De Jong, K.: Parameter setting in eas: a 30 year perspective. Parameter setting in evolutionary algorithms pp. 1–18 (2007)
8. Fogel, D.B., Fogel, L.J., Atmar, J.W.: Meta-evolutionary programming. In: Signals, systems and computers, 1991. 1991 Conference record of the twenty-fifth asilomar conference on, pp. 540–545. IEEE (1991)
9. Foster, J.A.: Taking "biology" just seriously enough: Commentary on "On the Mapping of Genotype to Phenotype in Evolutionary Algorithms" by Peter A. Whigham, Grant Dick, and James Maclaurin. Genetic Programming and Evolvable Machines **18**(3), 395–398 (2017). DOI 10.1007/s10710-017-9296-x
10. Hong, L., Drake, J.H., Woodward, J.R., Özcan, E.: A hyper-heuristic approach to automated generation of mutation operators for evolutionary programming. Applied Soft Computing (2017)
11. Keijzer, M.: Improving symbolic regression with interval arithmetic and linear scaling. Genetic programming pp. 275–299 (2003)
12. Lourenço, N., Pereira, F.B., Costa, E.: Sge: a structured representation for grammatical evolution. In: International Conference on Artificial Evolution (Evolution Artificielle), pp. 136–148. Springer (2015)
13. Mahfoud, S.W.: Niching methods for genetic algorithms. Urbana **51**(95001), 62–94 (1995)
14. Mckay, R.I., Hoai, N.X., Whigham, P.A., Shan, Y., O'Neill, M.: Grammar-based genetic programming: a survey. Genetic Programming and Evolvable Machines **11**(3-4), 365–396 (2010)

15. Medvet, E.: A comparative analysis of dynamic locality and redundancy in grammatical evolution. In: Genetic Programming: 20th European Conference, EuroGP 2017, Amsterdam, Netherlands, April 19-21, 2017, Proceedings, p. to appear. Springer International Publishing, Cham (2017)
16. Medvet, E.: Hierarchical grammatical evolution. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO (2017)
17. Medvet, E., Bartoli, A.: On the automatic design of a representation for grammar-based genetic programming. In: M. Castelli, L. Sekanina, M. Zhang, S. Cagnoni, P. García-Sánchez (eds.) Genetic Programming, pp. 101–117. Springer International Publishing, Cham (2018)
18. Medvet, E., Bartoli, A., Squillero, G.: An effective diversity promotion mechanism in grammatical evolution. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '17, pp. 247–248. ACM, New York, NY, USA (2017). DOI 10.1145/3067695.3076057
19. Medvet, E., Daolio, F., Tagliapietra, D.: Evolvability in grammatical evolution. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '17, pp. 977–984. ACM, New York, NY, USA (2017). DOI 10.1145/3071178.3071298
20. Miller, J.F., Smith, S.L.: Redundancy and computational efficiency in cartesian genetic programming. IEEE Transactions on Evolutionary Computation **10**(2), 167–174 (2006)
21. O'Neill, M., Brabazon, A., Nicolau, M., Garraghy, S.M., Keenan, P.: πGrammatical Evolution, pp. 617–629. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). DOI 10.1007/978-3-540-24855-2_70
22. O'Neill, M., Ryan, C.: Genetic code degeneracy: Implications for grammatical evolution and beyond. In: European Conference on Artificial Life, pp. 149–153. Springer (1999)
23. O'Neill, M., Ryan, C.: Grammatical evolution by grammatical evolution: The evolution of grammar and genetic code. Genetic Programming pp. 138–149 (2004)
24. Pagie, L., Hogeweg, P.: Evolutionary consequences of coevolving targets. Evolutionary computation **5**(4), 401–418 (1997)
25. Pappa, G.L., Ochoa, G., Hyde, M.R., Freitas, A.A., Woodward, J., Swan, J.: Contrasting meta-learning and hyper-heuristic research: the role of evolutionary algorithms. Genetic Programming and Evolvable Machines **15**(1), 3–35 (2014). DOI 10.1007/s10710-013-9186-9
26. Qin, A.K., Huang, V.L., Suganthan, P.N.: Differential evolution algorithm with strategy adaptation for global numerical optimization. IEEE transactions on Evolutionary Computation **13**(2), 398–417 (2009)
27. Rothlauf, F.: Representations for genetic and evolutionary algorithms. In: Representations for Genetic and Evolutionary Algorithms, pp. 9–32. Springer Berlin Heidelberg (2006)
28. Rothlauf, F., Goldberg, D.E.: Redundant representations in evolutionary computation. Evolutionary Computation **11**(4), 381–415 (2003)
29. Ryan, C.: A rebuttal to Whigham, Dick, and Maclaurin by one of the inventors of grammatical evolution: Commentary on "On the mapping of genotype to phenotype in evolutionary algorithms" by Peter A. Whigham, Grant Dick, and James Maclaurin. Genetic Programming and Evolvable Machines pp. 1–5 (2017). DOI 10.1007/s10710-017-9294-z
30. Ryan, C., Collins, J., Neill, M.O.: Grammatical evolution: Evolving programs for an arbitrary language, pp. 83–96. Springer Berlin Heidelberg, Berlin, Heidelberg (1998). DOI 10.1007/BFb0055930
31. Saravanan, N., Fogel, D.B., Nelson, K.M.: A comparison of methods for self-adaptation in evolutionary algorithms. BioSystems **36**(2), 157–166 (1995)
32. Scott, E.O., Bassett, J.K.: Learning genetic representations for classes of real-valued optimization problems. In: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, pp. 1075–1082. ACM (2015)
33. Simões, L.F., Izzo, D., Haasdijk, E., Eiben, A.E.: Self-adaptive genotype-phenotype maps: neural networks as a meta-representation. In: International Conference on Parallel Problem Solving from Nature, pp. 110–119. Springer (2014)
34. Spector, L.: Introduction to the peer commentary special section on "On the Mapping of Genotype to Phenotype in Evolutionary Algorithms" by Peter A. Whigham, Grant Dick, and James Maclaurin. Genetic Programming and Evolvable Machines **18**(3), 351–352 (2017). DOI 10.1007/s10710-017-9287-y
35. Squillero, G., Tonda, A.: (Over-)Realism in evolutionary computation: Commentary on "On the mapping of genotype to phenotype in evolutionary algorithms" by Peter A. Whigham, Grant Dick, and James Maclaurin. Genetic Programming and Evolvable Machines pp. 1–3 (2017). DOI 10.1007/s10710-017-9295-y

36. Squillero, G., Tonda, A.P.: Divergence of character and premature convergence: A survey of methodologies for promoting diversity in evolutionary optimization. Information Sciences **329**, 782–799 (2016). DOI 10.1016/j.ins.2015.09.056. URL `http://porto.polito.it/2622368/,http://linkinghub.elsevier.com/retrieve/pii/S002002551500729X`
37. Thorhauer, A.: On the non-uniform redundancy in grammatical evolution. In: International Conference on Parallel Problem Solving from Nature, pp. 292–302. Springer (2016)
38. Thorhauer, A., Rothlauf, F.: On the locality of standard search operators in grammatical evolution. In: International Conference on Parallel Problem Solving from Nature, pp. 465–475. Springer (2014)
39. Uy, N.Q., Hoai, N.X., O'Neill, M., McKay, R.I., Galván-López, E.: Semantically-based crossover in genetic programming: application to real-valued symbolic regression. Genetic Programming and Evolvable Machines **12**(2), 91–119 (2011)
40. Vanneschi, L., Castelli, M., Manzoni, L.: The k landscapes: a tunably difficult benchmark for genetic programming. In: Proceedings of the 13th annual conference on Genetic and evolutionary computation, pp. 1467–1474. ACM (2011)
41. Whigham, P.A., Dick, G., Maclaurin, J.: On the mapping of genotype to phenotype in evolutionary algorithms. Genetic Programming and Evolvable Machines pp. 1–9 (2017). DOI 10.1007/s10710-017-9288-x
42. Whigham, P.A., Dick, G., Maclaurin, J., Owen, C.A.: Examining the best of both worlds of grammatical evolution. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, pp. 1111–1118. ACM (2015)
43. Whigham, P.A., et al.: Grammatically-based genetic programming. In: Proceedings of the workshop on genetic programming: from theory to real-world applications, vol. 16, pp. 33–41 (1995)
44. White, D.R., Mcdermott, J., Castelli, M., Manzoni, L., Goldman, B.W., Kronberger, G., Jaśkowski, W., O'Reilly, U.M., Luke, S.: Better gp benchmarks: community survey results and proposals. Genetic Programming and Evolvable Machines **14**(1), 3–29 (2013)
45. Wong, P.K., Wong, M.L., Leung, K.S.: Hierarchical knowledge in self-improving grammar-based genetic programming. In: International Conference on Parallel Problem Solving from Nature, pp. 270–280. Springer (2016)