Structured Grammatical Evolution: A Dynamic Approach

Nuno Lourenço, Filipe Assunção, Francisco B. Pereira, Ernesto Costa, and Penousal Machado

Abstract Grammars have attracted the attention of researchers within the Evolutionary Computation field, specially from the Genetic Programming community. The most successful example of the use of grammars by GP is Grammatical Evolution (GE). In spite of being widely used by practitioners of different fields, GE is not free from drawbacks. The ones that are most commonly pointed out are those linked with redundancy and locality of the representation. To address these limitations Structured Grammatical Evolution (SGE) was proposed, which introduces a one-to-one mapping between the genotype and the non-terminals. In SGE the input grammar must be pre-processed so that recursion is removed, and the maximum number of expansion possibilities for each symbol determined. This has been pointed out as a drawback of SGE and to tackle it we introduce Dynamic Structured Grammatical Evolution (DSGE). In DSGE there is no need to pre-process the grammar, as it is expanded on the fly during the evolutionary process, and thus we only need to define the maximum tree depth. Additionally, it only encodes the integers that are used in the genotype to phenotype mapping, and grows as needed during evolution. Experiments comparing DSGE with SGE show that DSGE performance is never worse than SGE, being statistically superior in a considerable number of the tested problems.

1 Introduction

Grammars are widely used by computer scientists and researchers to represent complex structures by specifying restrictions on general domains, thus limiting the number of expressions that can be generated. The most common application

N. Lourenço (✉) · F. Assunção · E. Costa · P. Machado
CISUC, Department of Informatics Engineering, University of Coimbra, Coimbra, Portugal
e-mail: naml@dei.uc.pt; fga@dei.uc.pt; ernesto@dei.uc.pt; machado@dei.uc.pt

F. B. Pereira
CISUC, Department of Informatics Engineering, University of Coimbra, Coimbra, Portugal
Polytechnic Institute of Coimbra, Quinta da Nora, Coimbra, Portugal
e-mail: xico@dei.uc.pt

C. Ryan et al. (eds.), *Handbook of Grammatical Evolution*,
https://doi.org/10.1007/978-3-319-78717-6_6

is perhaps on the specification of the syntax of a programming language to define things such as type restrictions and operators precedence. Grammars are also useful to represent and describe interaction constraints between the different components of a system. As such, it is not a surprise that grammars captivated the attention of Evolutionary Computation (EC) researchers to help in the specification of problem restrictions and constraints that guide the evolutionary process, specifically in Genetic Programming (GP) [10].

One of the first GP proposals that used grammars to define the syntax and control the search bias was introduced by Whigham with his Context-Free-Grammar GP [22]. The introduction of grammars limited the form of possible solutions, allowing the definition of an explicit structure through the use of the productions of a grammar, and enabled the definition of constraints, ensuring that different components of solutions are not mixed together.

The main achievement in Grammar-Based Genetic Programming (GBGP) came with the introduction of Grammatical Evolution (GE) by Ryan et al. [15, 19], which since then has become one of the most popular and widespread GP methods. The main difference between CFG-GP and GE is related with how individual solutions are represented. While the former relies on a derivation-tree based representation, the latter uses a variable length linear integer string and a grammar to map individuals from the search space into the problem space. This separation between genotype and phenotype is usually seen as an advantage of GE over other techniques, since it is possible to decouple the search method from the problem we are solving, simplifying its application to different domains.

Despite the popularity of GE, some studies have shown that it has some drawbacks. Firstly, GE suffers from high levels of redundancy. A representation is said to be redundant when several different genotypes map in the same phenotype. Secondly, GE has a low locality [18], i.e., how variations at the genotype level reflect on differences at the phenotype level [6]. In a representation with high locality, a small modification on the genotype usually results in a small modification on the phenotype, nurturing the conditions for an effective sampling of the search space. If this condition is not satisfied, the search performed by an Evolutionary Algorithm (EA) tends to resemble that of a random search [17].

Over the years several modifications have been made to the original GE proposal in order to overcome its limitations. The most recent one is called Structured Grammatical Evolution (SGE), proposed by the authors in [12]. Its most noticeable characteristic is having a one-to-one relationship between genes and the non-terminals of the grammar being used. In order to allow a valid mapping, each gene encodes a list of integers that represent the possible derivation choices for the corresponding non-terminal. The structured representation of SGE, in which a gene is explicitly linked to a non-terminal of the grammar, ensures that changes in a single genotypic position do not affect the derivation options of other non-terminals. In the aforementioned work we analysed and compared the properties of both SGE and GE, and concluded that the new representation not only increases locality but also reduces redundancy. These results justify the increased performance of SGE over the traditional GE representation.

Nevertheless, SGE has been criticised because we need to specify beforehand the maximum levels of recursion in order to remove it from the grammar. In this work, we introduce a new version of the SGE, called Dynamic Structured Grammatical Evolution (DSGE), which addresses this main criticism. In this new version we specify the maximum tree-depth (similarly to what happens in standard tree-based GP), and the algorithm add the mapping numbers as required during the evolutionary search. Thus, we do not need to pre-process the grammar to remove the recursive productions. Additionally, we provide mechanisms to make sure that the generated trees are always within the allowed limits. We show that the DSGE version performs as good as the initial SGE algorithm, and in some cases it even surpasses its performance.

This chapter is organised as follows: in Sect. 2 we present the background. Next, in Sect. 3 we detail our new method: DSGE. In Sect. 4 we evaluate and compare the performance of the new method with the vanilla version of SGE in classical GP benchmarks, followed by a comparison of the two approaches in the domain of NeuroEvolution (Sect. 5). Finally, Sect. 6 gathers the main conclusions.

2 Background

Context-Free-Grammars (CFGs) have been widely used to represent and control the search bias of EAs [14]. Formally, a CFG is a tuple $G = (N, T, S, P)$, where N is a non-empty set of non-terminal symbols, T is a non-empty set of terminal symbols, S is an element of N called the axiom, and P is a set of production rules of the form $A ::= \alpha$, with $A \in N$ and $\alpha \in (N \cup T)^*$. N and T are disjoint. Each grammar G defines a language $L(G)$ composed by all sequences of terminal symbols (the words) that can be derived from the axiom: $L(G) = \{w : S \overset{*}{\Rightarrow} w,\ w \in T^*\}$. An example of a CFG is presented in Fig. 1. In this section we describe the GE and SGE approaches. For an in-depth review of the developments related with grammar-based evolutionary methods the reader might refer to [14].

Fig. 1 Example of a Context-Free-Grammar in the Backus-Naur Form (BNF)

```
<start> ::= <expr><op><expr>      (0)
          | <expr>                (1)
<expr>  ::= <term><op><term>      (0)
          | (<term><op><term>)     (1)
<op>    ::= +                     (0)
          | -                     (1)
          | /                     (2)
          | *                     (3)
<term>  ::= x1                    (0)
          | 0.5                   (1)
```

2.1 Grammatical Evolution

In Grammatical Evolution (GE) there is a separation between the genotype, i.e., a linear string of integers, and the phenotype, i.e., a program in the form of a tree expression. As a consequence, a mapping process is required to map the string into an executable program, using the production rules of a CFG. The translation of the genotype into the phenotype is done by simulating a leftmost derivation from the axiom of the grammar. This process scans the linear sequence from left to right and each integer (i.e., each codon) is used to determine the grammar rule that expands the leftmost non-terminal symbol of the current partial derivation tree. Consider the set of production rules defined in Fig. 1, where there are two options to rewrite the left-hand side symbol <start>. In the beginning we have a sentential form equal to the axiom <start>. To rewrite the axiom one must choose which alternative will be used by taking the first integer of the genotype and dividing it by the number of options in which we can derive <start>. The remainder of that operation will indicate the option to be used. In the example above, assuming that the first integer is 23, it follows that $23\%2 = 1$ and the axiom is rewritten as <expr><op><expr>. Then the second integer is read, and the same method is used to the left most non-terminal of the derivation. The complete mapping of an individual is showed in Table 1. Sometimes the length of the genotype is not enough to complete the mapping. When that happens the sequence is repeatedly reused in a process known as *wrapping*. If the mapping exceeds a pre-determined number of wrappings, the process stops and the worst possible fitness value is assigned to the individual.

The separation between the search space and the problem space is usually seen as one of the biggest advantages of GE, allowing it to be easily used in different problem domains. These characteristics is also appealing to practitioners of various scientific domains, since they can easily use GE to solve their problems.

Table 1 GE mapping procedure that translates the genotype of an individual into a polynomial expression (phenotype)

Derivation step	Integers left
<start>	[23, 7, 55, 22, 3, 4, 30, 16, 203, 24]
<expr><op><expr>	[7, 55, 22, 3, 4, 30, 16, 203, 24]
(<term><op><term>)<op><expr>	[55, 22, 3, 4, 30, 16, 203, 24]
(0.5 <op><term>)<op><expr>	[22, 3, 4, 30, 16, 203, 24]
(0.5 / <term>)<op><expr>	[3, 4, 30, 16, 203, 24]
(0.5 / 0.5)<op><expr>	[4, 30, 16, 203, 24]
(0.5 / 1) + <expr>	[30, 16, 203, 24]
(0.5 / 1) + <term><op><term>	[15, 203, 24]
(0.5 / 1) + x_1 <op><term>	[203, 24]
(0.5 / 1) + x_1 * <term>	[24]
(0.5 / 1) + $x_1 * x_1$	[]

Each row represents a derivation step. The used grammar is represented in Fig. 1

GE it is not exempted from criticisms. One is concerned with its initialisation procedure, which makes it difficult to create populations with valid individuals [16]. To overcome the initialisation problem, GE adopted a method similar to the one proposed in GP [10]. Another criticism pointed at GE is concerned with its high redundancy and its low locality. Rothlauf et al. in [18] showed that in approximately 90% of the time a change in the genotype does not change the phenotype. A second important result of this work is related with the remaining 10% of the modifications. Specifically, when the genotype suffers one mutation, changes of more than one unit occur at the phenotypic level. This means that many genotypic neighbours originate highly dissimilar phenotypes. One of the first proposals to increase the locality of GE was by Byrne et al. [3, 4]. They proposed new mutation operators that worked on the phenotypic level, which only changed the labels of the nodes in the derivation tree.

2.2 Structured Grammatical Evolution

Structured Grammatical Evolution (SGE) is a recent genotypic representation aimed at overcoming the locality and redundancy issues of GE. In SGE each gene is linked to a specific non-terminal, and it is composed by a list of integers used to select the expansion option. The length of each list is determined by computing the maximum possible number of expansions of the corresponding non-terminal. This structure ensures that the modification of a gene does not affect the derivation options of other non-terminals, thus limiting the number of changes that can occur at the phenotypic level, which result in a higher locality. The values inside each list are bounded by the number of possible expansion options of the corresponding non-terminal. Therefore, the mapping procedure does not rely on the modulo rule, avoiding the redundancy associated with it.

As an example consider the grammar depicted in Fig. 1. The non-terminals set is {<start>, <expr>, <term>, <op>}. Therefore, the SGE genotype is composed by four genes, each one linked to one specific non-terminal. To determine the length of the gene's lists we calculate the maximum number of expansions of each non-terminal. The <start> symbol is expanded only once, as it is the grammar axiom. The <expr> symbol is expanded, at most, twice, because of the rule <expr><op><expr>. The computation of the list size for <term> establishes a direct dependence between this non-terminal and <expr>: each time <expr> is expanded, <term> is expanded twice (in the two possible expansion options). As the grammar allows a maximum of two <expr> expansions, it immediately follows that the list size for the <term> gene is four. Following the same line of reasoning, the list size for the <op> gene is 3. Thus, the list sizes for each gene are: <start> : 1, <expr> : 2, <term> : 4, <op> : 3. To complete the list inside each gene we take the number of derivation options c_N of the corresponding non-terminal, and assign a random value from the interval $[0, c_N - 1]$ to every position. The <start>, <expr> and <term> symbols have $c_N = 2$, whereas <op> has

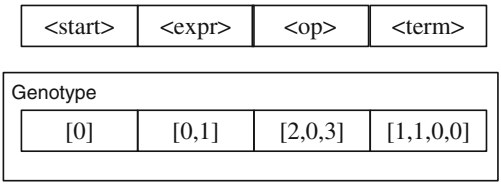

Fig. 2 Example of a SGE genotype for the grammar showed in Fig. 1

Table 2 SGE mapping procedure that converts a SGE individual into a polynomial expression

Derivation step	Integers left
<start>	[[0], [0, 1], [2, 0, 3], [1, 1, 0, 0]]
<expr><op><expr>	[[], [1, 0], [2, 0, 3], [1, 1, 0, 0]]
(<value><op><value>)<op><expr>	[[], [0], [2, 0, 3], [1, 1, 0, 0]]
(0.5 <op><value>)<op><expr>	[[], [0], [2, 0, 3], [1, 0, 0]]
(0.5 / <value>)<op><expr>	[[], [0], [0, 3], [1, 0, 0]]
(0.5 / 1)<op><expr>	[[], [0], [0, 3], [0, 1]]
(0.5 / 1) + <expr>	[[], [0], [3], [0, 1]]
(0.5 / 1) + <value><op><value>	[[], [], [3], [0, 0]]
(0.5 / 1) + x_1 <op><value>	[[], [], [3], [0]]
(0.5 / 1) + x_1 * <value>	[[], [], [], [0]]
(0.5 / 1) + x_1 * x_1	[[], [], [], []]

Each row represents a derivation step. The used grammar is represented in Fig. 1. The list of codons represents the integers needed for expanding <start>, <expr>, <op> and <value>, respectively

$c_N = 4$. Figure 2 shows an example of the genotype of a SGE individual. The complete mapping of this same individual into a polynomial expression is depicted in Table 2.

3 Dynamic Structured Grammatical Evolution

Dynamic Structured Grammatical Evolution (DSGE) is our novel GBGP approach. It derives from SGE, and addresses its common criticism of having to remove the recursion from the grammar beforehand. In this new version there is no need to pre-process the grammar in order to compute the maximum tree-sizes of each non-terminal symbol, so that intermediate grammar derivation rules are created to mimic the recursion process. Another advantage of the new proposal is that all of the genotype is used. Whilst in GE and SGE the genotype encodes the largest allowed sequence, in DSGE the genotype grows as needed. In the next sections we describe in detail the procedures needed to implement the DSGE.

Algorithm 1: Random candidate solution generation

```
1:  procedure CREATE_INDIVIDUAL(grammar, max_depth, genotype, symbol, depth)
2:      expansion = randint(0, len(grammar[symbol])-1)
3:      if is_recursive(symbol) then
4:          if expansion in grammar.recursive(symbol) then
5:              if depth ≥ max_depth then:
6:                  non_rec = grammar.non_recursive(symbol)
7:                  expansion = choice(non_rec)
8:      if symbol in genotype then
9:          genotype[symbol].append(expansion)
10:     else
11:         genotype[symbol] = [expansion]
12:     expansion_symbols = grammar[symbol][expansion]
13:     for sym in expansion_symbols do
14:         if not is_terminal(sym) then
15:             create_individual(grammar, max_depth, genotype, sym, depth+1)
```

3.1 Representation

Each candidate solution encodes an ordered sequence of the derivation steps of the used grammar that are needed to generate a specific solution for the problem at hand. The representation is similar to the one used in SGE, with one main difference: instead of computing and generating the maximum number of derivations for each of the grammar's non-terminal symbols, a variable length representation is used, where just the number of needed derivations are encoded. Consequently, there is no need to create intermediate symbols to deal with recursive rules.

To limit the genotype size, a maximum tree-depth needs to be defined. This means that the trees that are being generated will grow until a certain limit. To ensure that valid individuals are generated we need to change the initialisation and mapping procedures. These modifications are detailed in the following sections.

3.2 Initialisation

Algorithm 1 details the recursive function that is used to generate each initial solution. This procedure takes as input the following parameters: the grammar that describes the domain of the problem; the maximum tree-depth; the genotype (which is initially empty); the non-terminal symbol that we want to expand (initially the axiom is used); and the current tree-depth (initialised to 0). Then, for the non-terminal symbol given as input, one of its possible derivation rules is selected (lines 2–11) and the non-terminal symbols of the chosen derivation rule are recursively expanded (lines 12–15) following a depth-first approach. However, when selecting

Algorithm 2: Genotype to phenotype mapping procedure

```
1:  procedure MAPPING(genotype, grammar, max_depth, read_integers, symbol, depth)
2:      phenotype = ""
3:      if symbol not in read_integers then
4:          read_integers[symbol] = 0
5:      if symbol not in genotype then
6:          genotype[symbol] = []
7:      if read_integers[symbol] ≥ len(genotype[symbol]) then
8:          if depth ≥ max_depth then
9:              generate_terminal_expansion(genotype, symbol)
10:         else
11:             generate_expansion(genotype, symbol)
12:     gen_int = genotype[symbol][read_integer[symbol]]
13:     expansion = grammar[symbol][gen_int]
14:     read_integers[symbol] += 1
15:     for sym in expansion do
16:         if is_terminal(sym) then
17:             phenotype += sym
18:         else
19:             phenotype += mapping(genotype, grammar, max_depth, read_integers, sym,
                depth+1)
20:     return phenotype
```

the expansion rule there is the need to check whether or not the maximum tree-depth has already been reached (lines 3–5). If that happens, only non-recursive derivation rules can be selected for expanding the current non-terminal symbol (lines 6–7). This procedure is repeated until an initial population with the desired size is created.

3.3 Mapping Function

To map the candidate solutions genotype into the phenotype we use Algorithm 2. This procedure is similar to the one used to generate the initial population but, instead of randomly selecting the derivation rule to use in the expansion of the non-terminal symbol, we use the choice that is encoded in the individual's genotype (lines 12–19). During evolution, the genetic operators might change the genotype in a way that more integers will be necessary than the ones that we have available. When this happens new derivation rules are selected randomly and added to the genotype of the individual (lines 3–11).

3.4 Genetic Operators

To explore the problem's domain and therefore promote evolution EAs usually rely on recombination and mutation operators to explore the search space, looking for promising solutions for the problem being solved.

3.4.1 Mutation

The mutation operator is restricted to integers that are used in the genotype to phenotype mapping and changes a randomly selected expansion option (encoded as an integer) to another valid one, constrained to the restrictions imposed by the maximum tree-depth. To do so, we first select one gene. Then, we randomly select one of its integers and replace it with another valid possibility. Note that genes where there is just one possibility for expansion are not considered for mutation purposes. Figure 3 shows the application of the mutation operator.

3.4.2 Recombination

Recombination is used to recombine two parents to generate two offspring. The crossover is the same introduced by [12]. It starts by creating a random binary mask and the offspring are created by selecting the parents genes' based on the mask values. Recombination does not modify the lists inside the genes. This is

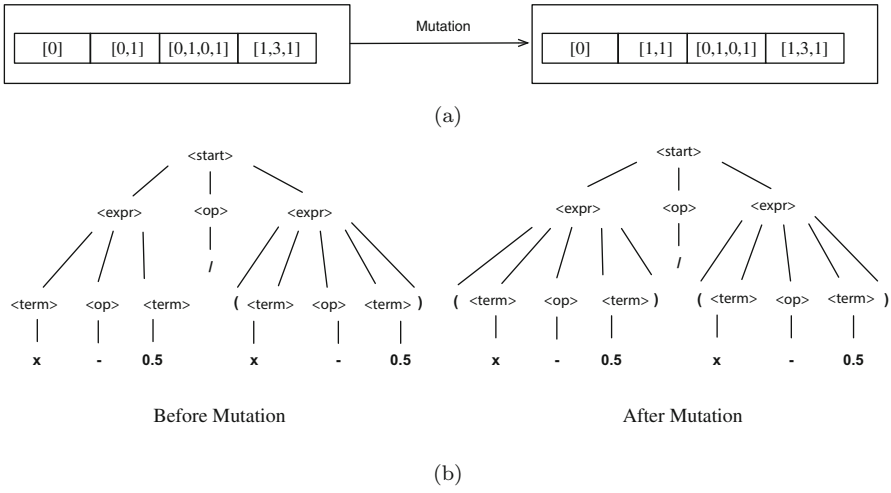

Fig. 3 Application of the mutation operator. Panel (**a**) details the application at the genotypic level, whereas panel (**b**) illustrates changes in the corresponding derivation trees. (**a**) Mutation application. (**b**) Derivation trees

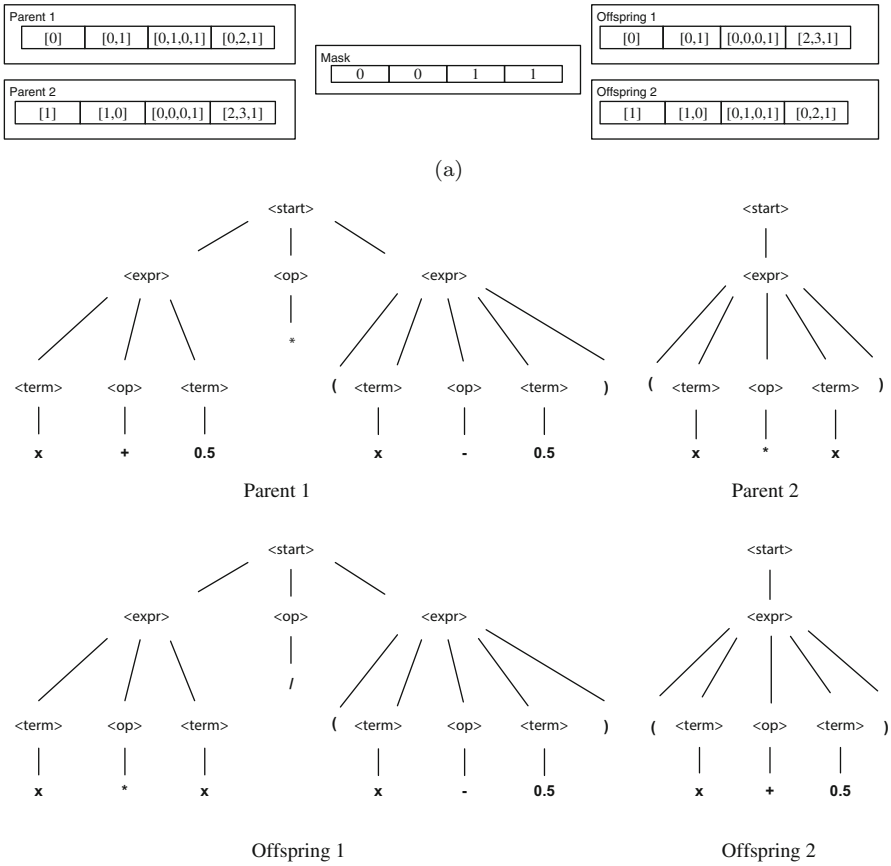

Fig. 4 Application of the recombination operator. Panel (**a**) details the application at the genotypic level, whereas panel (**b**) illustrates the changes in the corresponding derivation trees. (**a**) Crossover application. (**b**) Derivation trees

similar to what happens with uniform crossover for binary representations. Figure 4 demonstrates the application of this operator.

4 Classical Problems

In this section we present the experimental results that we conducted to assess the performance of the DSGE, conducting a relative comparison with its ancestor. We rely on the version of SGE described in [12].

Table 3 Experimental parameters used in the analysis for SGE and DSGE

Parameter	Value
Num. runs	30
Population size	1000
Generations	50
Crossover rate	95%
Mutation rate	$\frac{1}{\text{code size}}$
Tournament size	3
Elite size	1%
SGE parameter	Value
Recursion level	6
DSGE parameter	Value
Max. Initialisation depth	6
Max. depth	17

4.1 Experimental Setup

All the results reported are averages of 30 evolutionary runs, to allow for a sound statistical analysis. Considering that we are performing comparisons with only two groups, we do not assume anything about the distribution of the data, and because the initial populations of each method are different we rely on the Mann-Whitney non-parametric test with a significance level $\alpha = 0.05$. Since that we are performing comparisons between only two groups, no $p - value$ correction is needed. Note that when statistical differences are found, we also report the effect size [5] of the differences.

The numerical parameters used for the experiments performed with SGE and DSGE are presented in Table 3.[1] The *codon size* parameter stands for the number of integers in the DSGE genotype.

4.2 Benchmark Description

The problems that were considered in the experiments are the Boston housing symbolic regression, quartic symbolic regression, the 5-bit even parity, the 11-bit Boolean multiplexer and the Santa Fe Ant Trail. All the problems consider the minimisation of an error. For the regression problems, i.e., the Boston housing and quartic, we used the Root Relative Squared Error (RRSE) which is 0 for a model with a perfect fit. For the 5-bit parity and the 11-bit Boolean multiplexer we count the number of test cases that were incorrectly predicted. Finally, for the Santa Fe

[1] The implementations of SGE and DSGE are available at: SGE—https://github.com/nunolourenco/sge and DSGE—https://github.com/nunolourenco/dsge.

Ant Trail we consider the number of food pellets left after the maximum number of steps has been achieved.

4.2.1 Quartic Symbolic Regression

The relevance of this problem as a GP benchmark is highly debated [13], but we included it mostly for historical reasons. The aim is to approximate the function defined by:

$$f(x) = x^4 + x^3 + x^2 + x + 1 \tag{1}$$

where x is sampled in the interval $[-1, 1]$ with a step $s = 0.1$ The set of production rules considered for this problem is:

$$\begin{aligned}
\text{<start>} &::= \text{<expr>} \\
\text{<expr>} &::= \text{<expr><op><expr>} \\
&\quad \mid (\text{<expr><op><expr>}) \\
&\quad \mid \text{<pre_op>(<expr>)} \\
&\quad \mid \text{<var>} \\
\text{<op>} &::= + \mid - \mid * \mid / \\
\text{<pre_op>} &::= \sin \mid \cos \mid \exp \mid \text{inv} \mid \log \\
\text{<var>} &::= \text{x} \mid 1.0
\end{aligned}$$

where *inv* is $\frac{1}{f(x)}$. Moreover, we considered that the division, and the logarithm functions are protected, i.e., $\frac{1}{0} = 1$ and $log(f(x)) = 0$ *iff* $f(x) \leq 0$.

4.2.2 Boston Housing Problem

This is a regression dataset from the UCI repository [11]. The dataset is composed by the housing prices from the suburbs of Boston, and the aim is to create a regression model that predicts the median house price, given a set of demographic features. The dataset is composed by 506 examples, each one composed by 13 features (12 continuous, 1 binary), and one continuous output variable in the range $[0, 50]$. This problem corresponds to a typical machine learning task and we need to measure the ability of the evolved models to work with unseen instances. Following the guidelines from [23], the dataset is partitioned into 2 disjoint sets: (1) 90% of the examples are used as the training set to learn a model; (2) the

remaining 10% are used as the test set, to assess the performance of the model. The set of production rules is extended from the quartic problem, with the inclusion of the additional descriptive variables used for predicting house prices to the non-terminal <var>.

4.2.3 Pagie Polynomial Regression

The objective is to approximate the polynomial function defined by:

$$\frac{1}{1+x^{-4}} + \frac{1}{1+y^{-4}}. \tag{2}$$

The function is sampled in the $[-5, 5]$ interval, with a step $s = 0.4$. For this regression problem we use the following production set:

$$\begin{aligned}
\text{<start>} &::= \text{<expr>} \\
\text{<expr>} &::= \text{<expr><op><expr>} \\
&\quad \mid (\text{<expr>}) \\
&\quad \mid \text{<pre_op>}(\text{<expr>}) \\
&\quad \mid \text{<var>} \\
\text{<op>} &::= + \mid - \mid * \mid / \\
\text{<pre_op>} &::= \sin \mid \cos \mid \exp \mid \log \\
\text{<var>} &::= \text{x} \mid \text{y}
\end{aligned}$$

Even though it defines a smooth search space, the Pagie polynomial has the reputation for being difficult [9, 13].

4.2.4 Harmonic Curve Regression

The purpose is to approximate the series defined by:

$$\sum_{i}^{x} \frac{1}{i}, \tag{3}$$

where x is in the $[1, 50]$ interval, with a step $s = 1.0$. This problem is interesting as it complements the standard interpolation task with a generalisation step. In this second stage, the interval $x \in [51, 120]$ with $s = 1.0$ is considered. For the

harmonic curve regression problem we use the following production set:

$$
\begin{aligned}
\texttt{<start>} &::= \texttt{<expr>} \\
\texttt{<expr>} &::= \texttt{<expr><op><expr>} \\
&\quad \mid \texttt{(<expr>)} \\
&\quad \mid \texttt{<pre\_op>(<expr>)} \\
&\quad \mid \texttt{<var>} \\
\texttt{<op>} &::= + \mid * \\
\texttt{<pre\_op>} &::= + \mid * \mid \text{inv} \mid \text{sqrt} \\
\texttt{<var>} &::= \text{x}
\end{aligned}
$$

4.2.5 5-Bit Parity

The aim of this problem is to evolve a Boolean function that takes a binary string of length 5 as input and returns a value that indicates whether the number of 1s in the string is even (0) or odd (1). The production set for this problem is:

$$
\begin{aligned}
\texttt{<start>} &::= \texttt{<B>} \\
\texttt{<B>} &::= \texttt{<B> and <B>} \\
&\quad \mid \texttt{<B> or <B>} \\
&\quad \mid \text{not(<B> and <B>)} \\
&\quad \mid \text{not(<B> or <B>)} \\
&\quad \mid \texttt{<var>} \\
\texttt{<var>} &::= b_0 \mid b_1 \mid b_2 \mid b_3 \mid b_4
\end{aligned}
$$

where b_0, b_1, b_2, b_3, b_4 are the input bits.

4.2.6 11-Bit Boolean Multiplexer

The task of the 11-bit Boolean multiplexer is to decode a 3-bit address and return the value of the corresponding data register (d0, d1, d2, d3, d4, d5, d6, d7). The Boolean 11-multiplexer is a function of 11 arguments: three, s_0 to s_2 which correspond to the addresses and eight data registers, i_0 to i_7. The production set for this problem is defined as:

<start> ::= <B>

<B> ::= <B> and <B>

| <B> or <B>

| not(<B>)

| not(<B>or<B>)

| (<B>) if (<B>) else (<B>)

| <var>

<var> ::= $s_0 \mid s_1 \mid s_2 \mid i_0 \mid i_1 \mid i_2 \mid i_3 \mid i_4 \mid i_5 \mid i_6 \mid i_7$

4.2.7 Santa Fe Ant Trail

The goal of the Santa Fe Ant Trail problem is to evolve a set of instructions for an artificial agent so that it can collect a certain number of food pellets in a limited number of steps (650). The trail consists of 89 food pellets distributed in a 32×32 toroidal grid. The agent starts in the top-left corner of the grid facing east, and it can turn left, right, move one square forward, and check if the square ahead contains food. The production set for this problem is:

<start> ::= <code>

<code> ::= <line>

| <code>

<line>

<line> ::= if ant.sense_food():

<line>

else:

<line>

| <op>

<op> ::= ant.turn_left()

| ant.turn_right()

| ant.move_forward()

Like the Quartic Polynomial problem, it was included for historical reasons, in spite of the debate surrounding its utility as a GP benchmark.

4.3 Experimental Results

The experimental results described below are reported in terms of the mean best fitness value obtained at each generation, over the 30 independent runs.

The results for the Boston housing problem are shown in Fig. 5. Looking at the training results (left panel) it is possible to see that in the first 8 generations the performance of the two approaches is very similar. In the next two generations the SGE version seems to discover solutions that have a smaller error. However, at generation 12, the quality of the solutions found by DSGE surpasses the quality of the ones found by SGE. From this point onwards it seems that the former method has an advantage, resulting in solutions with a smaller error. This behaviour is observable until the end of the evolutionary search. The test results (right panel) exhibit a similar behaviour. Although for the first 7 generations DSGE generates worse quality solutions, its error rapidly decreases, and after about 12 generations is already smaller than the error showed by SGE. Regarding the statistical differences, the tests show no evidence of significant differences between the approaches on both the train and test phases.

Figure 6 shows the results of the harmonic curve regression problem; similarly to the previous problem the experimentation is divided into two stages: train and test. Regarding the train phase (left panel) the figure reveals that both SGE variants gradually discover better approximations as evolution progresses. However, DSGE seems to exhibit an increased effectiveness, since it discovers solutions with a lower error, in less time. In the test results (right panel), approximately from generation 10, the performance of DSGE starts decreasing (higher fitness values), indicating that it may be overfitting. Nonetheless, the test results reported by DSGE are slightly superior to the SGE ones (not statistically).

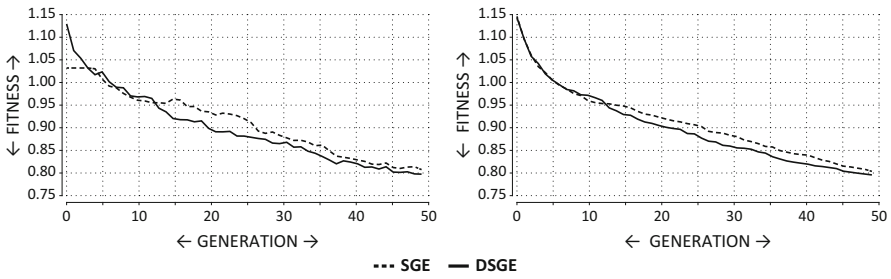

Fig. 5 Evolutionary results for the Boston Housing Problem. The panel on the left shows the results for training, and the panel on the right corresponds to the test results

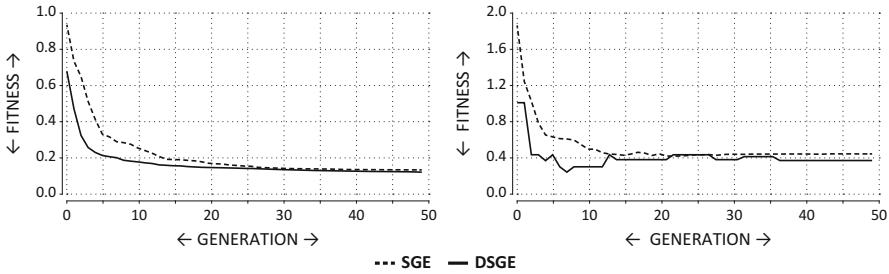

Fig. 6 Evolutionary results for the Harmonic Curve Regression. The panel on the left shows the results for training, and the panel on the right corresponds to the test results

Concerning the quartic problem, the results are depicted in Fig. 7a. It is possible to see that in both approaches the error gradually decreases over time, being almost 0 by the end of the evolutionary process. For this problem it is possible to see that there are no clear differences between DSGE and SGE. This apparent result is confirmed by the statistical analysis, which shows that there are no meaningful differences between the approaches. Looking at the Pagie polynomial, the optimization results follow a trend similar to the one identified in the previous problem (Fig. 7b). The individuals of the initial population of SGE and DSGE have comparable fitness, with the DSGE one being slightly worse. Then, as optimisation advances, DSGE gradually and consistently obtains low error solutions without stagnating. On the contrary, the vanilla version of SGE has a slower gradient, which prevents it from reaching better solutions within the allocated evolutionary time. Looking at the quality obtained by the two variants in the end of the evolutionary run, there is a noticeable difference between the methods, with DSGE obtaining solutions with considerable lower error. We applied a statistical test to look for meaningful differences, and the results showed that there are statistical significant differences between the approaches.

For the 5-bit even parity problem the results are shown in Fig. 7c. In this problem it is possible to see that having a less tightened algorithm such as DSGE is advantageous. Although in the first 8 generations SGE finds solutions with a smaller error, DSGE rapidly catches up with it, surpassing the quality of the solutions being found by the former method. This trend is visible throughout the entire evolutionary process. Statistical tests do not reveal any differences between the methods for this particular problem.

The 11-bit Boolean multiplexer results are shown in Fig. 7d. The plot shows that for this problem the DSGE clearly outperforms the vanilla version of SGE. Together with the 5-bit problem, the results seem to indicate that for Boolean problems with grammars similar to the ones described above it is advantageous to use the dynamic version of SGE. The statistical tests show that for the multiplexer problem there are meaningful differences between the two approaches, and that the effect size of the differences is large ($r > 0.5$).

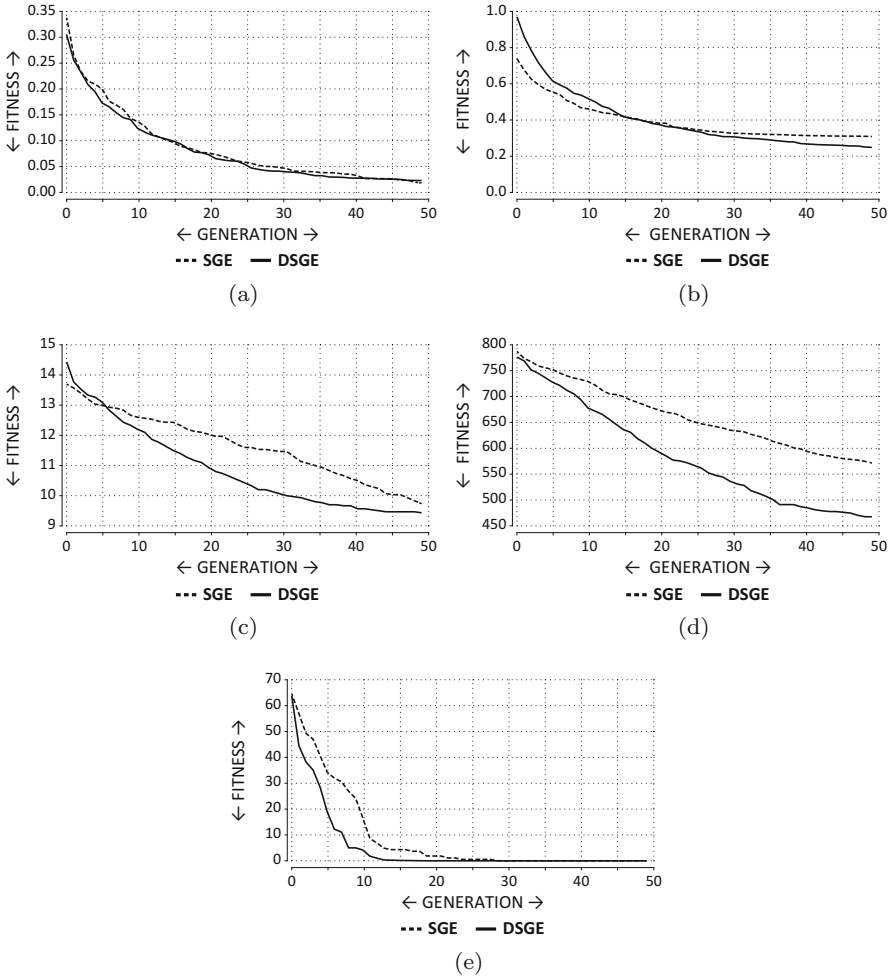

Fig. 7 Evolutionary results for the (**a**) quartic regression, (**b**) pagie polynomial regression, (**c**) 5-bit parity, (**d**) 11-bit Boolean multiplexer and (**e**) Santa Fe Ant Trail

Finally, Fig. 7e shows the results for the Santa Fe Ant problem. The figure shows that for this problem, at the end of the evolutionary search, there are no differences between the two methods. However it is possible to see that the DSGE is more efficient since it needs less generations to reach a optimal solution for the problem. The statistical tests corroborated the fact that there are no significant differences between the two methods.

5 NeuroEvolution

In addition to testing DSGE with the above classical problems we also investigate its capacity of automatically searching for the topology and weights of Artificial Neural Networks (ANNs) (also known as NeuroEvolution). This problem is deemed as a more complex task, since we have to simultaneously search for both the structure of the model, i.e., the network topology, and its real parameterisation, i.e., the synaptic weights. Our goal is to show that for this type of problems DSGE is a good alternative, able to find effective neural network architectures.

5.1 Experimental Setup

The experiments described in this section follow a setup similar to the one described in Table 3, where only the population size, number of generations and maximum depth parameters differ. We use a population size of 100 individuals, which are evolved throughout 500 generations, totalling 50,000 evaluations per evolutionary run.

The grammar used for the conducted experiments is detailed in Fig. 8. The grammar is capable of representing one-hidden-layered networks with a single output neuron. The sequence of hidden-nodes is encoded in the <sigexpr> non-terminal symbol. Each hidden-node is the result of an activation function (in this case the sigmoid function) that receives as input the connections to input nodes and a bias value. The output of the activation function is then multiplied by a weight that encodes the synaptic weight between that specific neuron and the output neuron.

Fig. 8 Grammar used for the NeuroEvolution experiments

$$
\begin{aligned}
\texttt{<sigexpr>} &::= \texttt{<node>} \\
&\quad \mid \texttt{<node>} + \texttt{<sigexpr>} \\
\texttt{<node>} &::= \text{sig}(\texttt{<sum>} + \texttt{<bias>}) * \texttt{<weight>} \\
\texttt{<sum>} &::= \texttt{<weight>} * \texttt{<features>} \\
&\quad \mid \texttt{<sum>} + \texttt{<sum>} \\
\texttt{<features>} &::= x_1 \\
&\quad \mid \ldots \\
&\quad \mid x_n \\
\texttt{<weight>} &::= \texttt{<number>} \\
\texttt{<bias>} &::= \texttt{<number>} \\
\texttt{<number>} &::= \texttt{<digit>.<digit><digit>} \\
&\quad \mid -\texttt{<digit>.<digit><digit>} \\
\texttt{<digit>} &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \\
&\quad \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

While in the previous experiments we defined a maximum depth that was counted from the root node, in the following experiments we consider the depth at the subtree level, and define maximum depths for each non-terminal symbol. This way, we are able to set the upper bounds on the number of nodes and connections. For the following experiment we define for the <sigexpr> and <sum> non-terminal symbols a maximum depth of 6 and 3, respectively.

To evolve effective ANNs, apart from defining a grammar to encode the topology and weights of the ANNs, we also need to choose a fitness function suitable to measure the performance of the candidate solutions. For the tasks described in this section we use the exponential Root Mean Squared Error (RMSE) per class, which is defined as:

$$\prod_{c=1}^{m} exp\left(\sqrt{\frac{\sum_{i=1}^{n_c}(o_i - t_i)^2}{n_c}}\right),$$

where m is the number of classes of the problem, n_c is the number of instances of the problem that belong to class c, o_i is the confidence value predicted by the evolved network, and t_i is the target value. This way, on the one hand we are able to deal with unbalanced datasets because the error is computed at a class level; on the other hand we avoid overfitting by penalising more higher errors than lower ones.

5.2 Benchmark Description

We selected four classification problems, all of them binary because of the constraints imposed by the used grammar-based approaches, that do not allow the reuse of previously created neurons (further discussed in [1]). The problems have an increasing number of features, and thus increasing complexity in terms of the classification task to be solved. The following paragraphs present a brief description of each of the used benchmarks.

Flame [7] Gathers 240 instances, each with two attributes, and the goal is to separate between two different classes: one containing 87 instances and the other with 153 instances. Typically used for clustering purposes.

Wisconsin Breast Cancer Detection (WDBC) [11, 21] Comprises 30 features that are extracted from digitalised images of breast masses. The dataset has 569 instances, where 212 are malign and 357 are benign.

Ionosphere [11, 20] Collects ionosphere radar returns, that are to be separated into good (if the returns evidences of structure, 300 instances) and bad (otherwise, 126 instances). For each instance 34 properties are given.

Sonar [8, 11] Contains 60 features of sonar signals that allow a classification model to separate between signals that are bounced off a metal cylinder (111 instances) or a rock cylinder (97 instances).

5.3 Experimental Results

Figure 9 shows the evolution of the fitness of the best individuals across generations for SGE and DSGE for the flame, WDBC, ionosphere and sonar datasets. Results are averages of 30 independent runs. Despite the similar results during the initial generations, it is perceptible that in the long term DSGE outperforms SGE on all tested problems, indicating that the new representation, where no mutation is silent, promotes locality and the efficient exploration of the search space.

Apart from analysing the fitness evolution we have also recorded several other metrics related with the performance of the generated networks, namely: RMSE, Area Under the ROC Curve (AUROC) and f-measure. These metrics are recorded for both the train and test sets; recall that only the train set is used for the evolutionary process. Table 4 reports the results that are obtained with SGE and DSGE. Results are averages of the best network (in terms of fitness) of each of the 30 evolutionary runs; each cell is formatted as: mean $\pm$ standard deviation. Moreover we also report the number of neurons and number of used features, which allow us to better understand the structure and complexity of the evolved networks.

A perusal analysis of the results confirms that DSGE outperforms SGE on all tested benchmarks, and on all recorded performance metrics, i.e., RMSE DSGE values are inferior to those reported by SGE, and accuracy, AUROC and f-measure are superior on the experiments conducted using DSGE. In addition, the standard deviation values are lower when using DSGE, which indicates that it consistently discovers effective solutions.

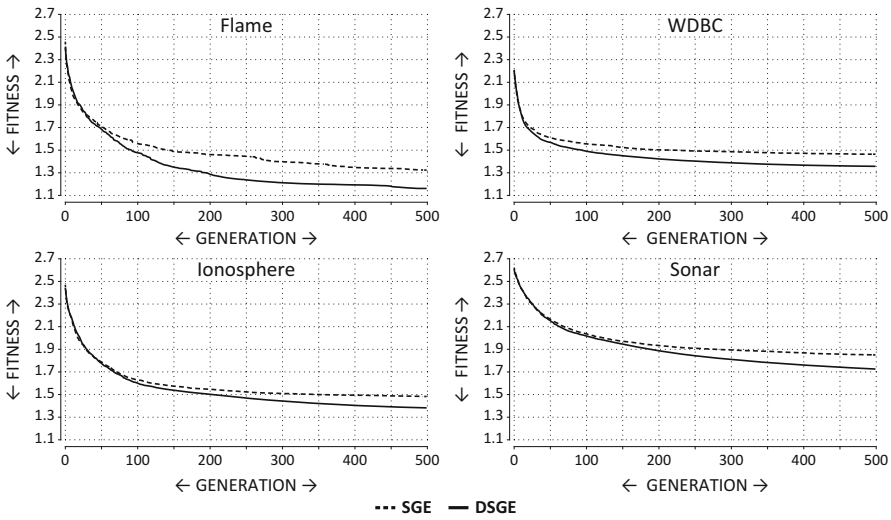

Fig. 9 Fitness evolution of the best individuals across generations for the flame, WDBC, ionosphere and sonar datasets

Table 4 Evolution of one-hidden-layered ANNs

			Flame	WDBC	Ionosphere	Sonar
Fitness		SGE	1.32 ± 0.25	1.46 ± 0.08	1.48 ± 0.18	1.85 ± 0.18
		DSGE	$1.16 \pm 0.13^{+++}$	$1.36 \pm 0.06^{+++}$	$1.38 \pm 0.13^{+++}$	$1.73 \pm 0.12^{+++}$
Train	RMSE	SGE	0.16 ± 0.13	0.19 ± 0.03	0.21 ± 0.07	0.34 ± 0.06
		DSGE	$0.08 \pm 0.06^{+++}$	$0.15 \pm 0.03^{+++}$	$0.17 \pm 0.05^{++}$	$0.30 \pm 0.05^{++}$
	Accuracy	SGE	0.96 ± 0.08	0.95 ± 0.02	0.93 ± 0.11	0.84 ± 0.12
		DSGE	$0.99 \pm 0.03^{+++}$	$0.97 \pm 0.01^{+++}$	$0.97 \pm 0.03^{+++}$	$0.90 \pm 0.04^{++}$
	AUROC	SGE	0.98 ± 0.04	0.99 ± 0.01	0.94 ± 0.04	0.91 ± 0.04
		DSGE	$0.99 \pm 0.01^{+++}$	$0.99 \pm 0.00^{+++}$	$0.96 \pm 0.03^{+++}$	$0.93 \pm 0.04^{\sim}$
	F-measure	SGE	0.96 ± 0.08	0.93 ± 0.03	0.93 ± 0.18	0.78 ± 0.23
		DSGE	$0.99 \pm 0.02^{+++}$	$0.96 \pm 0.02^{+++}$	$0.98 \pm 0.02^{+++}$	$0.88 \pm 0.05^{++}$
Test	RMSE	SGE	0.22 ± 0.13	0.23 ± 0.04	0.32 ± 0.05	0.44 ± 0.04
		DSGE	$0.14 \pm 0.08^{++}$	$0.20 \pm 0.03^{++}$	$0.28 \pm 0.04^{+++}$	$0.43 \pm 0.04^{\sim}$
	Accuracy	SGE	0.93 ± 0.09	0.93 ± 0.02	0.87 ± 0.10	0.73 ± 0.09
		DSGE	$0.97 \pm 0.05^{+++}$	$0.95 \pm 0.02^{+++}$	$0.90 \pm 0.03^{++}$	$0.76 \pm 0.05^{\sim}$
	AUROC	SGE	0.96 ± 0.08	0.98 ± 0.02	0.90 ± 0.05	0.82 ± 0.05
		DSGE	$0.99 \pm 0.03^{++}$	$0.98 \pm 0.01^{++}$	$0.93 \pm 0.04^{++}$	$0.83 \pm 0.04^{\sim}$
	F-measure	SGE	0.94 ± 0.09	0.91 ± 0.03	0.89 ± 0.17	0.64 ± 0.20
		DSGE	$0.98 \pm 0.04^{+++}$	$0.93 \pm 0.03^{+++}$	$0.93 \pm 0.02^{++}$	$0.72 \pm 0.06^{\sim}$
Num. neurons		SGE	4.87 ± 1.83	3.73 ± 1.53	3.53 ± 1.36	3.07 ± 1.39
		DSGE	6.47 ± 1.20	6.23 ± 1.58	5.97 ± 1.78	6.13 ± 1.69
Num. features		SGE	2.00 ± 0.00	12.0 ± 6.51	12.1 ± 5.79	13.3 ± 6.42
		DSGE	2.00 ± 0.00	14.5 ± 3.52	13.3 ± 4.74	17.6 ± 4.66

Comparison between DSGE and other grammar-based approaches. Results are averages of 30 independent runs. + and $\sim$ represent the result of statistical tests (see text)

To analyse the tendency of evolution to overfit we measure the difference between the train and test performances, which for SGE are, on average, 0.08, 0.06, 0.04 and 0.06, for the RMSE, accuracy, AUROC and f-measure, respectively. For DSGE the differences are, on average, 0.09, 0.06, 0.04 and 0.06, for the same metrics. However the difference in the accuracy in slightly superior when using DSGE, it is our understanding that this is a result of the DSGE greater performance, rather than an indicator of overfitting.

To verify if the differences between DSGE and SGE are significant we perform a statistical analysis. First, we use the Kolmogorov-Smirnov and Shapiro-Wilk tests, with a significance level of $\alpha = 0.05$, where it is possible to acknowledge that the data does not follow any distribution, and as such, the Mann-Whitney U test (non-parametric), with $\alpha = 0.05$ is used to perform the pairwise comparison. The results of the statistical tests are reported in Table 4 using a graphical overview, where: $\sim$ indicates no statistical difference between the compared methods and + signals that DSGE is statistically superior to SGE. The effect size is denoted by the number of

+ signals, where +, ++ and +++ correspond respectively to low ($0.1 \leq r < 0.3$), medium ($0.3 \leq r < 0.5$) and large ($r \geq 0.5$) effect sizes.

For those searching a comparison between SGE and GE on the same benchmarks, and using the same grammar, refer to [2]. It has already been demonstrated that evolving both the structure and weights of ANNs using SGE was the path to follow, in the sense that the found weights directly impact the learning of the network. Moreover, it was demonstrated that SGE was able to discover better results than GE. Consequently, as DSGE performs better than SGE we indirectly conclude that DSGE performs better than GE for these specific class of problems.

6 Conclusions

Since the early days of GP that researchers have been using grammars to specify restrictions on problem domains and guide the evolutionary process. Amongst the most successful proposals is GE, which is used by practitioners from different backgrounds, due to its simplicity and for being a plug-and-play approach. Consequently, to deal with the evolution of solutions for different problem domains, only the grammar needs to be changed. Nevertheless, GE has been target of some criticisms concerning the way in which it searches the space of solutions. Amidst these criticisms, the most infamous are concerned with the high levels of redundancy and the low locality of the representation, which arguably make it less effective [23].

SGE is a new GE variant, with its most distinctive feature being the one-to-one correspondence between genes and non-terminals of the used grammar. In spite of the good results obtained by SGE, it was perceptible that it still had some drawbacks, namely the fact that we needed to pre-process the grammar in order to remove any recursive rules. This was done by adding intermediate symbols that resemble the recursive expansions of the grammar.

In this work we introduce a new version of SGE called DSGE, which removes the need to pre-process the grammar. In concrete we define the maximum depth that each expression tree can have, and we stop the trees from growing beyond that limit (as in traditional tree-based GP). We presented the procedures needed to create and map solutions from the genotype to the phenotype, and conducted a wide and systematic set of experiments to assess the effectiveness of the new approach. We selected classical benchmark problems from the literature, and a more challenging set of problems from the NeuroEvolution area.

The results show that the performance of the new method, DSGE, is never inferior to the vanilla version of SGE, being superior in a vast set of the used benchmarks, obtaining solutions with a better quality.

Acknowledgements This research is partially funded by: Fundação para a Ciência e Tecnologia (FCT), Portugal, under the grant SFRH/BD/114865/2016. We gratefully acknowledge the support of NVIDIA Corporation for the donation of a Titan X GPU. We would also like to thank Tiago Martins for all the patience making the charts herein presented.

References

1. F. Assunção, N. Lourenço, P. Machado, B. Ribeiro, Towards the evolution of multi-layered neural networks: a dynamic structured grammatical evolution approach, in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '17* (ACM, New York, 2017), pp. 393–400. https://doi.org/10.1145/3071178.3071286
2. F. Assunção, N. Lourenço, P. Machado, B. Ribeiro, Automatic generation of neural networks with structured grammatical evolution, in *2017 IEEE Congress on Evolutionary Computation (CEC)* (2017), pp. 1557–1564. https://doi.org/10.1109/CEC.2017.7969488
3. J. Byrne, M. O'Neill, A. Brabazon, Structural and nodal mutation in grammatical evolution, in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, New York, 2009, pp. 1881–1882
4. J. Byrne, M. O'Neill, J. McDermott, A. Brabazon, An analysis of the behaviour of mutation in grammatical evolution, in *Genetic Programming*. Lecture Notes in Computer Science, vol. 6021 (Springer, Berlin, 2010), pp. 14–25
5. A. Field, *Discovering Statistics Using IBM SPSS Statistics* (Sage, Los Angeles, 2013)
6. R. Franz, *Representations for Genetic and Evolutionary Algorithms* (Springer, Berlin, 2006)
7. L. Fu, E. Medico, FLAME, a novel fuzzy clustering method for the analysis of DNA microarray data. BMC Bioinf. **8**(1), 1 (2007)
8. R.P. Gorman, T.J. Sejnowski, Analysis of hidden units in a layered network trained to classify sonar targets. Neural Netw. **1**(1), 75–89 (1988)
9. R. Harper, Spatial co-evolution: quicker, fitter and less bloated, in *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation* (ACM, New York, 2012), pp. 759–766
10. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, vol. 1 (MIT Press, Cambridge, 1992)
11. M. Lichman, UCI machine learning repository (2013). http://archive.ics.uci.edu/ml
12. N. Lourenço, F.B. Pereira, E. Costa, Unveiling the properties of structured grammatical evolution. Genet. Program. Evolvable Mach. **17**(3), 251–289 (2016)
13. J. McDermott, D.R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong, U.M. O'Reilly, Genetic programming needs better benchmarks, in *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation* (ACM, New York, 2012), pp. 791–798
14. R.I. Mckay, N.X. Hoai, P.A. Whigham, Y. Shan, M. O'Neill, Grammar-based genetic programming: a survey. Genet. Program. Evolvable Mach. **11**(3–4), 365–396 (2010)
15. M. O'Neill, C. Ryan, Grammatical evolution. IEEE Trans. Evol. Comput. **5**(4), 349–358 (2001)
16. M. O'Neill, C. Ryan, *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*. Genetic Programming, vol. 4 (Kluwer Academic, Boston, 2003)
17. F. Rothlauf, On the locality of representations, in *Genetic and Evolutionary Computation Conference*. Lecture Notes in Computer Science (2003), pp. 1608–1609
18. F. Rothlauf, M. Oetzel, On the locality of grammatical evolution, in *European Conference on Genetic Programming* (Springer, Berlin, 2006), pp. 320–330
19. C. Ryan, J. Collins, M.O. Neill, Grammatical evolution: evolving programs for an arbitrary language, in *European Conference on Genetic Programming* (Springer, Berlin, 1998), pp. 83–96

20. V.G. Sigillito, S.P. Wing, L.V. Hutton, K.B. Baker, Classification of radar returns from the ionosphere using neural networks. Johns Hopkins APL Tech. Dig. **10**(3), 262–266 (1989)
21. W.N. Street, W.H. Wolberg, O.L. Mangasarian, Nuclear feature extraction for breast tumor diagnosis, in *IS&T/SPIE's Symposium on Electronic Imaging: Science and Technology* (International Society for Optics and Photonics, Bellingham, 1993), pp. 861–870
22. P.A. Whigham, et al.: Grammatically-based genetic programming, in *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, vol. 16 (1995), pp. 33–41
23. P.A. Whigham, G. Dick, J. Maclaurin, C.A. Owen, Examining the best of both worlds of grammatical evolution, in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation* (ACM, New York, 2015), pp. 1111–1118