Information Sciences 333 (2016) 21–43

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

neat Genetic Programming: Controlling bloat naturally

Leonardo Trujillo [a,*], Luis Muñoz [a], Edgar Galván-López [b], Sara Silva [c,d]

[a] *Tree-Lab, Doctorado en Ciencias de la Ingeniería, Departamento de Ingeniería Eléctrica y Electrónica, Instituto Tecnológico de Tijuana, Blvd. Industrial y Av. ITR Tijuana S/N, Mesa Otay C.P. 22500, Tijuana, BC, Mexico*
[b] *School of Computer Science and Statistics, Trinity College, Dublin, Ireland*
[c] *BioISI – Biosystems & Integrative Sciences Institute, Faculty of Sciences, University of Lisbon, Portugal*
[d] *NOVA IMS, Universidade Nova de Lisboa, Lisbon 1070-312, Portugal*

ARTICLE INFO

Article history:
Received 5 July 2015
Revised 21 October 2015
Accepted 2 November 2015
Available online 10 November 2015

Keywords:
Genetic programming
Bloat
NeuroEvolution of augmenting topologies
Flat operator equalization

ABSTRACT

Bloat is one of the most widely studied phenomena in Genetic Programming (GP), it is normally defined as the increase in mean program size without a corresponding improvement in fitness. Several theories have been proposed in the specialized GP literature that explain why bloat occurs. In particular, the Crossover-Bias Theory states that the cause of bloat is that the distribution of program sizes during evolution is skewed in a way that encourages bloat to appear, by punishing small individuals and favoring larger ones. Therefore, several bloat control methods have been proposed that attempt to explicitly control the size distribution of programs within the evolving population. This work proposes a new bloat control method called *neat*-GP, that implicitly shapes the program size distribution during a GP run. *neat*-GP is based on two key elements: (a) the NeuroEvolution of Augmenting Topologies algorithm (NEAT), a robust heuristic that was originally developed to evolve neural networks; and (b) the Flat Operator Equalization bloat control method, that explicitly shapes the program size distributions toward a uniform or flat shape. Experimental results are encouraging in two domains, symbolic regression and classification of real-world data. *neat*-GP can curtail the effects of bloat without sacrificing performance, outperforming both standard GP and the Flat-OE method, without incurring in the computational overhead reported by some state-of-the-art bloat control methods.

1. Introduction

Genetic Programming (GP) [9,10,21] is an evolutionary computation (EC) paradigm used for automatic program induction, its general goal is to generate computer programs through an evolutionary search. In its most common form, GP can be understood as a supervised learning algorithm that attempts to construct a syntactically valid expression using a finite set of basic functions and input variables, guided by a domain dependent objective or cost function [21]. In its original form [9], GP is characterized by two main features that distinguishes it from other EC techniques. Firstly, evolved solutions represent valid syntactic expressions or programs, that might be used as models, predictors, operators or classifiers. The ability of GP to construct syntactic expressions directly, without assuming a prior model, can allow it to produce highly interpretable solutions, that not only solve the

* Corresponding author. Tel.: +52 6643389391.
E-mail addresses: leonardo.trujillo@tectijuana.edu.mx (L. Trujillo), lmunoz@tectijuana.edu.mx (L. Muñoz), edgar.galvan@scss.tcd.ie (E. Galván-López), sara@fc.ul.pt (S. Silva).
URL: http://www.tree-lab.org (L. Trujillo)

http://dx.doi.org/10.1016/j.ins.2015.11.010
0020-0255/

problem but also provide insights into the problem domain. Secondly, GP uses a variable length encoding scheme, where the set of candidate solutions contains programs of different size and shape.

EC literature contains many examples of the problem solving abilities of GP, that illustrate the flexibility of the search paradigm [8]. Indeed, GP can be understood as a hyper-heuristic, an algorithmic approach for the automatic synthesis of heuristic approaches, a view that has strong theoretical background and real-world applicability [19]. Despite its success, GP is still not used as an off-the-shelf methodology [16], in the way that, for example, Support Vector Machines or Linear Regression are used. This lack of wider acceptance stems from some important pragmatic limitations of the GP approach.

In particular, syntactic search can be inefficient, due to its poor local structure and ill-defined fitness landscape (refer to [18] and [22] where the authors reviewed some of the main open issues in GP). Among them, one of the most studied problems is the bloat phenomenon, which occurs when program trees tend to grow unnecessarily large, without a corresponding increase in fitness [21,25]. In some sense, bloat seems to be an unavoidable consequence of the nature of the search space in GP and fitness driven search [11,12]. Moreover, bloat causes several undesirable side effects, since evaluating large programs is more time consuming, and large solutions are more difficult to interpret. Therefore, multiple approaches have been studied to deal with bloat, ranging from modifications of the basic search operators up to investigating the use of different search spaces, such as semantic space [6,33] and behavioral space [30].

This paper presents a novel approach toward bloat control, that leverages the insights of recent studies [23] and an algorithm originally developed for neuroevolution [28]. Silva [23] suggests that a powerful bloat control strategy is to induce a uniform distribution of program sizes within the evolving population. In particular, she proposed the Flat Operator Equalization bloat control method (Flat-OE), which explicitly forces the evolving population to follow a uniform distribution of program sizes, while the range of the distribution remains constant across all generations.

The contribution of our work is the development of a GP-based system based on the NeuroEvolution of Augmenting Topologies (NEAT) algorithm, which uses speciation to protect novel solution topologies and promote the incremental evolution of complexity [28]. In our recent work, we showed that NEAT can run bloat free, using a careful parametrization and system configuration [29]. However, it was unclear if the results obtained from neuroevolution could be replicated in a traditional GP domain.

The proposed algorithm is called *neat*-GP and it can be understood as a stripped down version of the original NEAT algorithm, which is adapted to the GP paradigm, designed to induce similar search dynamics as those shown by Flat-OE. Experiments are carried out using a tree-based representation and tested on several benchmark problems for both symbolic regression and classification. The results show that a *neat*-GP based search can outperform a standard GP search, based on test performance and especially with regards to solution size and depth. These results agree with those reported in [29], with the added advantage that the bloat control method does not incur in any additional computational cost exhibited by other state-of-the-art bloat control methods [23,26].

The remainder of this paper proceeds as follows. Section 2 provides a comprehensive overview on both the bloat phenomenon and the NEAT algorithm, discussing the theoretical causes of bloat, state-of-the-art bloat control methods and how bloat relates to NEAT. The proposed *neat*-GP algorithm is presented in Section 3, discussing different possible variants and detailing important algorithm features. The experimental work is presented in Section 4, discussing system setup, benchmarking and results. Finally, a summary and concluding remarks are outlined in Section 5.

2. Background

This section presents a comprehensive discussion of the most relevant background topics related to the current research paper.

2.1. Bloat

In what follows, the bloat phenomenon in GP is presented, focusing on theoretical aspects and state-of-the-art bloat control methods; a more complete survey on this topic can be found in [25,26].

2.1.1. Bloat theory and bloat control methods

The most well-established explanation of bloat is the fitness-causes-bloat theory (FCBT), originally developed by Langdon and Poli [11]. The FCBT assumes the following common features in a GP search: (a) there is a many-to-one mapping from syntactic space to fitness space; and (b) for a particular fitness value (e.g., the optimum), there are exponentially more large programs than there are small programs with the same fitness. Hence, if a particular fitness value is desired, there is a tendency toward larger, or bloated, programs during a GP search, simply because there are more of them within the search space. Indeed, stating that the search for fitness is the main cause of bloat is by now uncontroversial, since it is basically the underlying factor in all major bloat theories [25]. Moreover, recent works suggest that a GP search that does not consider fitness explicitly can in fact avoid bloat altogether, by searching for novelty instead of solution quality [13,30].

Currently, one of the most useful bloat theories is the crossover bias theory (CBT) [20]. Focusing on canonical GP, the CBT states that bloat is produced by the effect that subtree crossover has on the distribution of program size. While the average size of trees is not affected, the size distribution is skewed in a particular way, producing a large number of small trees. For most fairly challenging problems, small trees will have a relatively low fitness. This in consequence will bias the selection operator

toward favoring larger programs, causing an increase in the average tree size within the population, effectively bootstrapping the bloating phenomenon.

2.1.2. Operator equalization

Given the insights provided by the CBT, Dignum and Poli [4] proposed the Operator Equalisation (OE) bloat control method, that focuses on explicitly controlling the distribution of program sizes at each generation. OE has produced impressive results in several benchmark and real-world problems, and has led to the development of a family of related methods [26,27]. Despite the success of OE, there are several practical concerns with it. For instance, it relies on an expensive computational process that generates, evaluates and in many cases rejects program trees that do not fit the desired target distribution. If we consider that one of the main practical reasons for bloat control is to reduce the inherent computational costs in a GP search, it seems counterproductive to use a bloat control strategy that might increase the total computational effort. Moreover, recent results suggest that some of the underlying assumptions in OE may not be justified. We discuss these issues in the following paragraphs.

Given the shortcomings of the original OE algorithms, Silva attempted to develop a stripped down version of OE [23]. The goal was to construct a simpler algorithm with a lower computational cost, while at the same time maintaining the bloat control features of standard OE. In particular, Silva combined the brood recombination operator [1] and the dynamic limits survival strategy [25]. However, experimental results showed that the proposed approximation of the OE method failed at controlling bloat. The reason for this appears to lie in a contradiction between OE and the CBT. Since the latter suggests that small individuals are harmful to the evolutionary process, OE methods tend to promote target size distributions that exclude such individuals from the population. On the other hand, while OE methods are designed to promote such distributions, in practice OE does not seem to actually fit them. In fact, Silva showed that OE tends to produce uniform or flat distributions of program sizes, with a roughly similar proportion of trees of different sizes, from small to large [23]. Based on these results, Silva proposed the Flat-OE method, where a flat target distribution is sought, such that the range of the distribution remains constant throughout the search. Empirical results suggest that Flat-OE can control bloat while not compromising the quality of the evolved solutions.

Nonetheless, the original strategy developed in [23] seems promising; i.e., determine the underlying properties of OE and develop an approximate algorithm that satisfies these properties, while at the same time reducing the computational overhead induced by the bloat control mechanism. It seems that the proposed mechanisms failed (brood recombination and dynamic limits), due to the fact that the underlying assumptions behind what OE does during a GP search were wrong; i.e., OE does not eliminate small program trees from the evolving population. Therefore, the current work follows a similar general strategy and develops an approximate version of OE by eliminating what appears to be the cause of bloat by adapting the main features of NEAT into the GP paradigm. However, instead of attempting to reproduce the original OE algorithm, this work is inspired on the insights gained from Flat-OE [27].

2.2. NeuroEvolution of augmenting topologies

The goal of this paper is to naturally handle bloat by implementing a GP algorithm that can maintain a close to uniform distribution of program sizes. In particular, the proposal is to adapt the NEAT algorithm [28] to the general GP paradigm, which was originally developed to evolve neural networks (NN). Nonetheless, it is possible to argue that NEAT should be considered as a specialized GP variant, since it evolves a variable length arrays (genotype) that represent NNs, a particular computational model. However, unlike GP, NEAT does not explicitly consider a syntactic search space or symbolic representation for the evolved solutions. Moreover, since NEAT employs a graph representation for NNs with variable topology and size, it can also be affected by bloat. The main components in NEAT, which are reviewed next, promote diversity in both solution size and shape, which is precisely the strategy followed in [23]. This section provides an introduction to NEAT, contextualizing it within the broader GP paradigm, and presents experimental evidence, which suggests that NEAT can perform a bloat-free search if it is configured properly.

2.2.1. The main components in NEAT

Stanley and Miikkulainen [28] developed NEAT as a variable length evolutionary algorithm that explicitly encodes the topology and connection weights of a NN. The algorithm can be broken down into the following main components.

The first component is a variable length list representation that encodes graph structures. The genome is conceptually divided into two segments: the first contains node genes that specify the set of input, output and hidden nodes of the network; while the second segment contains the set of connection or synaptic genes, that specify the input and output node of each connection and its respective weight. To allow search operations that are coherent between NNets of different sizes, NEAT includes a historical marking for each synaptic connection that uniquely defines each connection introduced into the evolving population. In this way, when crossover is performed between two parent networks, that can have different topologies, the connection genes are first aligned based on these historical markings, allowing for the identification of the shared structure between both parents (nodes and connections). Matching genes between two parents are randomly inherited from either parent, but any disjoint or excess genes are inherited from the fittest parent. Moreover, the connection weights are also inherited after a crossover operation. NEAT does not evolve the activation functions in the NNs, these are set a priori for all network nodes. Besides the crossover operation described above, other search operators include a weight mutation, and structural mutations that can either add nodes or add connections. This set of search operators is unusual, in the sense that they always produce offspring of equal or larger size than their parents, a configuration that should induce code growth.

The second component is that the initial population in NEAT only contains NNs that share the same minimal topology, in most cases this is a fully connected feedforward network with no hidden neurons and randomly generated connection weights. This encourages an incremental evolution of solution complexity; i.e., an incremental evolution of size and topology. Fundamentally, NEAT assumes that the best way to start an evolutionary search is by using a simple/small network, and progressively builds more complex networks as the search progresses. If the problem can be solved by a parsimonious network then the search might be able to find it early on during the search, and only if it fails to achieve this, the search will progress toward larger and more complex topologies.

Finally, NEAT incorporates a scheme that protects topological innovation during the search. Consider that at the beginning of a run, all the individuals share the same initial (minimal) structure, while the search focuses on improving the connection weights in the networks. The search operators progressively add structural elements (nodes and/or connections) to the base topology, each time a new structural element is generated its weight is set randomly. This could cause a problem, because a randomly generated connection with a randomly set weight, could have a destructive effect on fitness. However, it is reasonable to assume that an increase in structural complexity might be required to solve difficult problems, hence the need to protect these structural innovations so they are not discarded by selection as soon as they appear. NEAT accomplishes this by using speciation based on topological similarities and fitness sharing [7], where the fitness of each individual is penalized based on its similarity with other individuals within the population. The key element is the use of a problem specific distance measure, denoted by δ. In this case it is based on the topological similarities between two networks expressed by

$$\delta = \frac{c_1 \cdot G + c_2 \cdot D}{N} + c_3 \cdot \overline{W}\,, \tag{1}$$

where D is the number of disjoint genes, G is the number of excess genes between them, $\overline{W}$ is the average weight difference of matching genes, N is the number of genes in the larger genome, and c_x are weight coefficients. Speciation not only penalizes the selection probability of individuals, it also serves as a constraint on parent selection for crossover, by posing an interspecies crossover rate to a relatively low value of 5%. Thus, crossover is performed between individuals of the same species, which in turn generates topologically similar offspring compared to their parents.

In summary, NEAT is a variable length evolutionary algorithm that evolves graph structures which represent NNs. From this general perspective, we state that NEAT is a special form of GP, where evolved solutions express instances from a narrow class of functions or programs, those that can be represented as NNs. However, the results obtained with NEAT can provide useful insights to the GP paradigm as a whole [29].

2.2.2. Bloat in NEAT

It can be argued that grouping NNs based on topological structure is not the best way to promote a diverse set of functional behaviors [31], since networks with different topologies can produce the same results, and *vice versa*. Nonetheless, speciation based on topological similarities can produce a diverse population of network topologies; i.e., a diverse population of network shapes and sizes. This is a key feature that inspired the present work, in that NEAT provides a promising approximation to the main Flat-OE strategy. Therefore, we could expect NEAT to search without producing bloat. Indeed, it has been previously hypothesized that NEAT may intrinsically control bloat.[1]

Our previous work confirmed that NEAT can be executed bloat-free, but only if it is configured appropriately [29]; the main results are briefly summarized here. Since NEAT is a fairly complex and intricate algorithm, in our previous work [29] we used the freely available Java implementation of NEAT[2] which closely follows the original NEAT algorithm [28].[3] NEAT was tested on two standard benchmark problems, the XOR problem and the 3-bit parity problem, which are both distributed with the Java library, executing 30 independent runs for each of these two problems. Performance is analyzed using both fitness and solution size given by the number of network nodes.

However, a re-parametrization of NEAT referred to as bloat-free NEAT (BF-NEAT) produces more promising results in terms of bloat, as shown in Figs. 1 and 2 for the XOR and Parity problems, respectively. BF-NEAT is configured in such a way that new individuals have a higher probability of survival, overall elitism is increased, and all species are protected regardless of their 'age' or historic performance. In these experiments, BF-NEAT, which is shown in Figs. 1(b) and 2(b), is able to maintain smaller programs during the search, the distribution is not skewed toward larger sizes, contrary to what is shown for NEAT in Figs. 1(a) and 2(a). BF-NEAT encourages the formation of populations with different program sizes, without excluding programs based on their size, which is consistent with the published results regarding Flat-OE. Moreover, BF-NEAT does not show any substantial decrease in performance relative to NEAT, as discussed in detail in [29]. The important lesson to take from these results is to confirm that the general NEAT strategy can, under certain conditions, control bloat during a search by implicitly shaping the program size distribution.

[1] To the authors' knowledge, the only explicit, yet informal, discussion of this issue is given in the official NEAT website http://www.cs.ucf.edu/~kstanley/neat.html.

[2] Source: http://nn.cs.utexas.edu/?jneat.

[3] While there are many open implementations of NEAT available, many of them are modified or simplified variants of the original algorithm, and it is not straightforward to determine what consequences these modifications, no matter how slight, can have on the search dynamics.

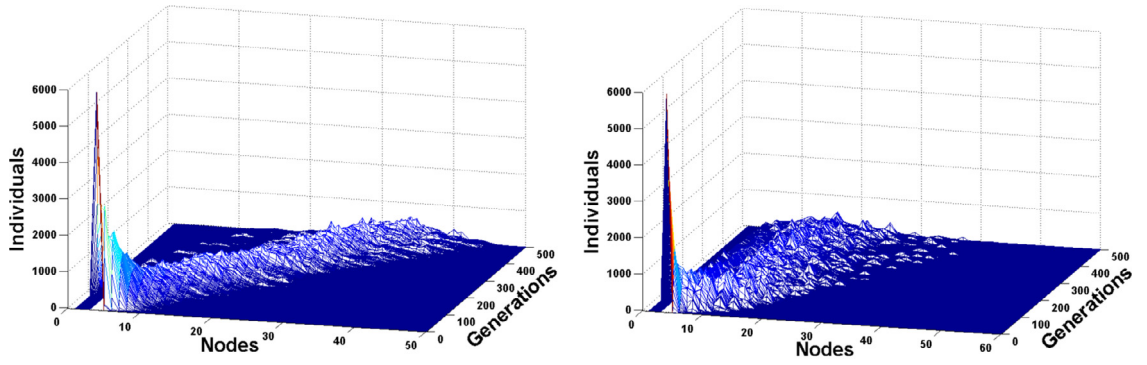


Fig. 1. Evolution of the size distribution of individuals based on number of nodes across generations on the XOR problem for: (a) NEAT and (b) BF-NEAT.

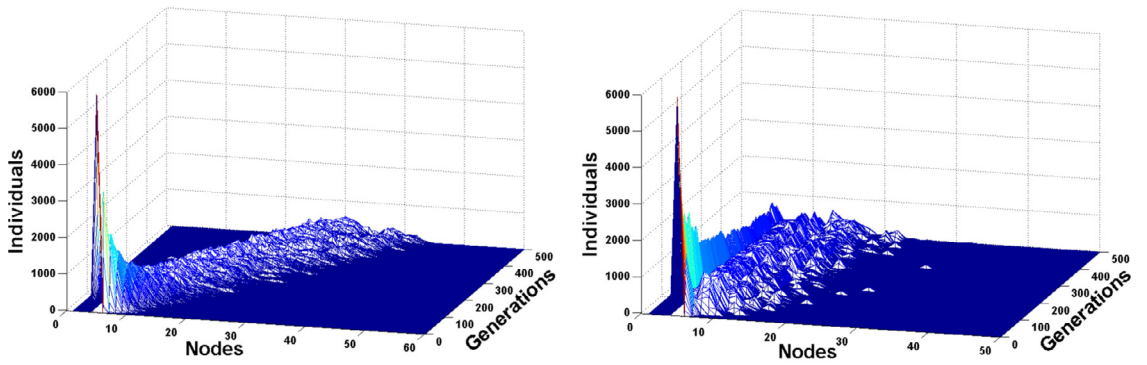


Fig. 2. Evolution of the size distribution of individuals based on number of nodes across generations on the Parity problem for: (a) NEAT and (b) BF-NEAT.

3. *neat* Genetic Programming

The goal of this work is to develop a bloat-free GP search, inspired by the insights of Flat-OE and built around the basic features of NEAT, that implicitly shapes the program size distribution. The proposed method is called *neat*-GP, taking the two evolutionary paradigms on which the proposed algorithm is based on and which serve as namesakes. Note that the name *neat*-GP should not be taken as an acronym, since what is evolved are program trees and not network structures. The name is just intended to convey the inspiration of the algorithm in a light-hearted manner.

The remainder of this section presents a detailed description of *neat*-GP, describing how the general NEAT methodology was ported to the GP domain. The following section presents an extensive experimental evaluation of the algorithm.

3.1. Overview

It is important to note that *neat*-GP is not designed to be an exact reproduction of NEAT, instead only the more general and important design principles are integrated into the canonical GP algorithm. Indeed, the goal was to identify and adopt the main design principles in NEAT that could allow the search to run bloat-free.

Therefore, the following main aspects of NEAT are considered:

1. Initialization and Speciation: NEAT starts the search with a random population of small/simple solutions, and progressively builds and protects solution complexity, while also maintaining and promoting diversity using speciation. To perform speciation a dissimilarity measure between trees is required (see Eq. (1)) as well as a process that determines species membership. Then, based on species membership, fitness sharing is used to penalize individuals from densely populated species.
2. Genetic Operators: NEAT uses several different search operators, specifically designed for the specialized solution representation. Moreover, NEAT enforces a strict policy for crossover, where interspecies crossover is highly discouraged, to ensure that offspring will be able to replace their parents within the same species. Additionally, selection and survival enforce a strong selective pressure. The general principles are included in *neat*-GP, with some simplifications, and a crossover operator is proposed that is similar to the original NEAT crossover designed for NNs.

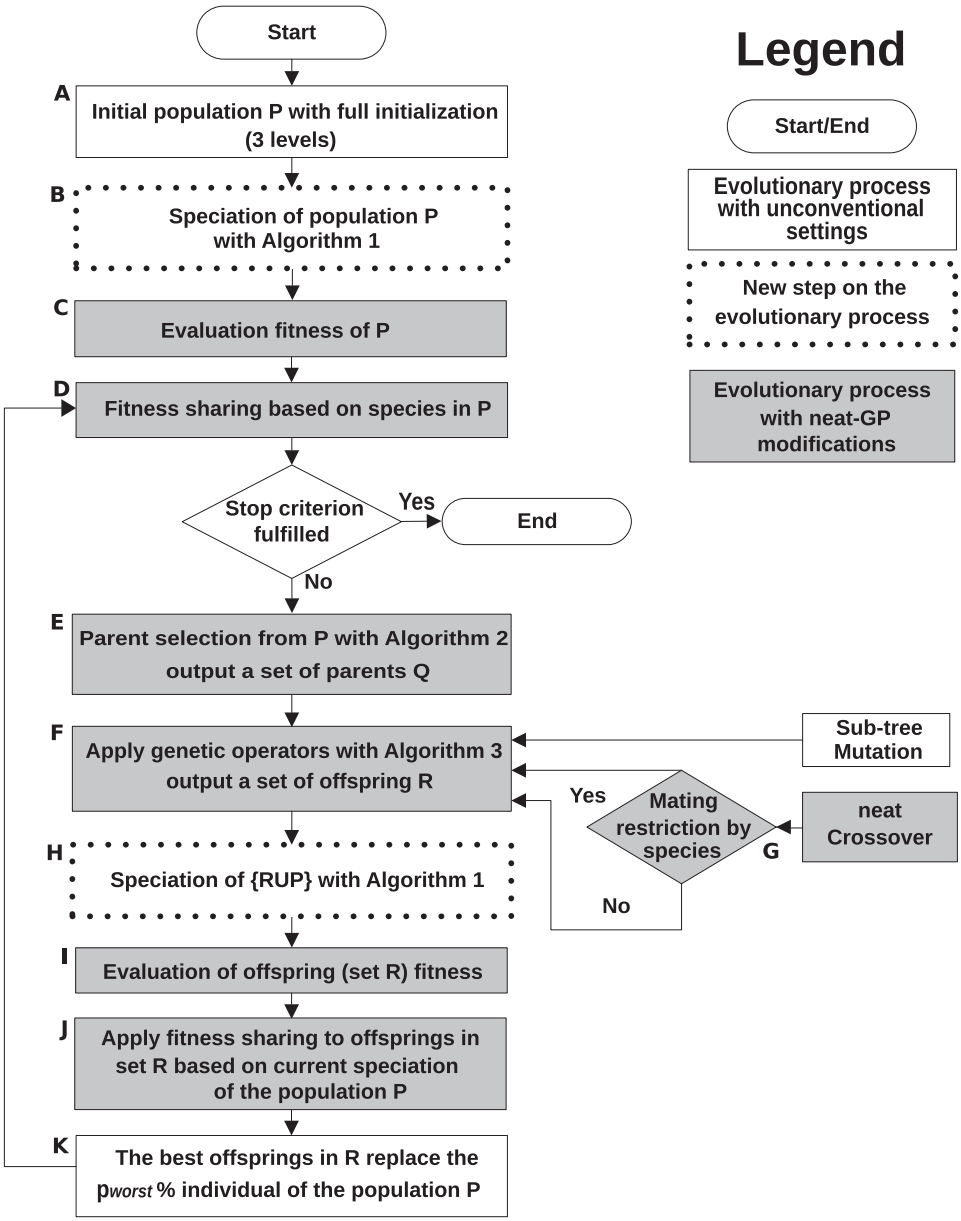


Fig. 3. General flow diagram of the *neat*-GP algorithm.

Fig. 3 presents a flowchart of the basic *neat*-GP algorithm. In general the flowchart describes a basic EA or GP, with initialization, fitness evaluation, selection and survival. However the figure presents several types of processes, denoted by different blocks in the diagram; these are: (1) standard GP processes (in some cases with unconventional settings), shown as white blocks with solid contours; (2) a new process based on the NEAT algorithm, shown as white blocks with dotted contours; and (3) a standard GP process with a unique implementation based on NEAT, shown as grey blocks with solid contours. Here we give a general description of the overall algorithm and then provide a detailed description of each block and algorithms in the following subsections.

First, the algorithm starts with a randomly generated population, starting with small full trees (Block A of Fig. 3). Second, speciation is performed on the entire population, based on the size of each solution, as described in Algorithm 1 (Block B). Third, fitness evaluation is performed and fitness sharing is applied, to protect individuals within small species and to penalize those from larger species (Blocks C and D). Fourth, if the stopping criterion has not been satisfied, then parent selection is performed using Algorithm 2 (Block E). Fifth, offspring are generated using the parents selected in the previous step and Algorithm 3 (Block F). Moreover, following NEAT, a new crossover operation is proposed called neat-crossover used along with standard subtree mutation (Block G). Sixth, the offspring are inserted into the population and Algorithm 1 is used to perform speciation, while their fitness is assigned using fitness sharing based on the current population (Blocks H, I and J). Seventh, an elitist survival

Algorithm 1: Speciation algorithm used in *neat*-GP.

```
Data: Set of individuals P | ∃T_i ∈ P where T_i.species = NULL
Result: Set of individuals P | ∄T_i ∈ P where T_i.species = NULL
for ∀T_i ∈ P do
    if T_i.species = NULL then
        for ∀T_j ∈ P \ {T_i} do
            if δ_T(T_i, T_j) ≤ h then
                T_i.species ← T_j.species;
                break;
            end
        end
        if T_i.species = NULL then
            create new species S_u;
            T_i.species ← S_u;
        end
    end
end
```

Algorithm 2: Parent selection algorithm used in *neat*-GP.

Data: Set of n individuals $P \mid \nexists T_i \in P$ where $T_i.species = NULL$
Data: Average fitness of the population P: $\bar{f} = \frac{1}{n}\sum_{i=1}^{n} f(T_i)$
Data: p_{worst}% individuals to replace
Result: Set Q of parents
$Q \leftarrow \emptyset$;
$T_i.descendants \leftarrow \lfloor \frac{\bar{f}}{f(T_i)} \rfloor \; \forall T_i \in P$;
$Q \leftarrow P$;
for *each species* S_j *of* Q **do**
  Eliminate the p_{worst}% individuals that belong to S_j from Q;
end
Order the remaining individuals in Q based on the penalized fitness f'

strategy is used and the algorithm in order to replace the p_{worst}% solutions in the current population with the best offspring (Block K). Finally, the algorithm iterates into the following generation by applying fitness sharing to the new population and evaluating the termination criterion. In what follows each algorithm and process is described in detail, for simplicity each is described using pseudocode with set notation and C++ style OOP instructions.

3.2. Initialization and speciation

This subsection summarizes Blocks A–D and H–J in the *neat*-GP flowchart presented in Fig. 3.

3.2.1. Initial population

Probably the simplest component to reproduce from the original NEAT algorithm is the minimal complexity of the initial population. For *neat*-GP, this is done using the full initialization method and a small initial depth, in this work set to 3 levels, where the root node is regarded as depth 1. In this way, all individuals in the initial population share the same shape and size, producing only one species in the initial population, as is carried out in NEAT.

3.2.2. Tree dissimilarity measure

The NEAT measure defined in Eq. (1) is not directly applicable to a tree representation. However, the same general principles are desired, determine the shared topological structure $S_{i,j}$ between two trees T_i and T_j, similar to what is done in one-point or homologous crossover [21]; see Fig. 4. To determine the shared structure we identify the overlapping upper tree from both trees, following the approach described in [34]. Moreover, n_{T_x} and d_{T_x} refer to the number of nodes and depth of a GP tree T_x, respectively. Note that $S_{i,j}$ is also a tree, with a particular size $n_{S_{i,j}}$ and depth $d_{S_{i,j}}$. Then, the dissimilarity between two trees T_i and T_j is given by

$$\delta_T(T_i, T_j) = \beta \frac{N_{i,j} - 2n_{S_{i,j}}}{N_{i,j} - 2} + (1-\beta)\frac{D_{i,j} - 2d_{S_{i,j}}}{D_{i,j} - 2}\,, \tag{2}$$

Algorithm 3: Genetic operators applied in *neat*-GP.

Data: Set Q of parents to generate k offspring
Data: Mutation and crossover probabilities (p_m, pc), number of offspring n
Result: Set of offspring R of size n
$R \leftarrow \emptyset, i \leftarrow 0$;
while $i < n$ **do**
  $eflag \leftarrow uRandom(\{0, 1\})$;
  if $eflag$ **then**
    $T_1 \leftarrow Q.best()$, where $Q.best()$ returns the first (best) individual in Q;
  else
    $T_1 \leftarrow Q.rand()$, where $Q.rand()$ returns a random individual in Q;
  end
  $oflag \leftarrow Random(p_m, p_c)$;
  if $oflag$ = *Mutation* **then**
    $T' \leftarrow mutate(T_1)$;
    $R \leftarrow R \cup \{T'\}$;
    $T_1.descendants \leftarrow T_1.descendants - 1$;
    $i \leftarrow i + 1$;
  else
    if $\exists T \in Q \mid [T.species = S] \wedge [T \neq T_1]$ **then**
      $T_2 \leftarrow T \in Q \mid f'(T) \geq f'(T_u),\ \forall T_u.species = S$;
    else
      $T_2 \leftarrow Q.rand()$;
    end
    $T' \leftarrow$ neat-Crossover(T_1, T_2);
    $R \leftarrow R \cup \{T'\}$;
    $T_1.descendants \leftarrow T_1.descendants - 0.5$;
    $T_2.descendants \leftarrow T_2.descendants - 0.5$;
    $i \leftarrow i + 1$;
    if $T_2.descendants \leq 0$ **then**
      $Q \leftarrow Q \setminus \{T\}$;
    end
  end
  if $T_1.descendants \leq 0$ **then**
    $Q \leftarrow Q \setminus \{T\}$;
  end
end

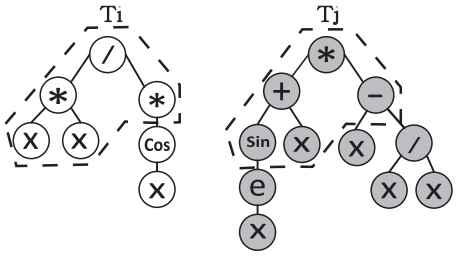


Fig. 4. The shared topological structure $S_{i,j}$ between two individuals T_i and T_j, depicted by a dashed line.

where $N_{i,j} = n_{T_i} + n_{T_j}$, $D_{i,j} = d_{T_i} + d_{T_j}$, and $\beta \in [0, 1]$ such that $\delta_T \in [0, 1)$.[4] On the right-hand side of Eq. (2), the first term measures the difference with respect to program size, while the second term measures the difference in depth. Thus, setting $\beta = 0.5$ gives an equal importance to program size and depth. This is similar to the original configuration reported by NEAT with the c_1 and c_2 parameters, and it is also the parametrization used in our previous work with BF-NEAT [29].

[4] When $n_{T_i} = n_{T_j} = d_{T_i} = d_{T_j} = 1$ then $\delta_T = 0$.

3.2.3. Defining species membership

neat-GP defines species membership using Algorithm 1 and Eq. (2), which is applied before fitness evaluation within the evolutionary loop. Briefly, each individual T_i that has not been assigned to a species is compared with each individual T_j in the current population that does belong to a species, one after another. When $\delta_T(T_i, T_j) \leq h$, with threshold h an algorithm parameter, then T_i is assigned to the species to which T_j belongs and no more comparisons are done. Another approach would be to compare T_i with every individual in the population, and then choose the species of the T_j that gave the minimum dissimilarity. However, we choose to use the first approach for two reasons. First, this helps limit the number of total comparisons that are performed on average, thus reducing computational cost. Second, it is the approach used in the original NEAT algorithm, with strong results [28]. Moreover, if $\delta_T(T_i, T_j) > h$ for all $j \neq i$, then a new species is created with T_i as its only member. In the end, this process divides the population into several species S_u, that can be understood as non-overlapping subsets of the population P, such that $P = S_1 \cup S_2 \cup \ldots S_n$ with n is the number of species.

3.2.4. Fitness sharing

After individuals have been grouped into species, fitness sharing is used among individuals of the same species. Basically, the fitness of a program is adjusted based on the total number of individuals in the species. For simplicity, all problems are treated as minimization tasks (minimize the error in symbolic regression or classification). Thus for an individual tree T_i, which is a member of species S_u and with fitness $f(T_i)$, an adjusted fitness f' is computed by

$$f'(T_i) = |S_u| f(T_i) \tag{3}$$

where $|S_u|$ is the number of individuals in species S_u. This penalizes individuals of densely populated species, and promotes a diverse population of program sizes. It is important to note that this strategy helps protect new program trees, produced by crossover or mutation, that might be introducing novel tree structures into the population but also exhibiting low fitness values. Moreover, an elitist criterion is included, such that the best individual from each species is not subject to fitness sharing, guaranteeing that the best individuals have a better chance of surviving and producing offspring.

3.3. Search operators

Like all other evolutionary algorithms, *neat*-GP relies on the standard set of search operators, which include selection, crossover and mutation. This subsection summarizes Blocks E, F and G of the *neat*-GP flowchart presented in Fig. 3.

3.3.1. Parent selection

This process performs the parent selection step of the evolutionary process, as described in Algorithm 2, taking as input the current (speciated) population P, the average population fitness and algorithm parameters, and producing as output a set of parents Q. First, the number of expected descendants of each individual T_i is computed proportional to its fitness $f(T_i)$. In this step, the original fitness f is used instead of the adjusted fitness f'. This is done because some individuals in the population were not penalized by Eq. (3) (i.e., the best individual in each species). If the number of descendants was computed based on f' instead, the algorithm would allocate a very large number of expected offspring to the best individuals in each species, imposing a high selective pressure and reducing diversity, possibly leading toward premature convergence.

Afterward, a copy of the current population P is made, we refer to it as Q. For each species S_j in Q the p_{worst}% individuals in the species are eliminated from Q. Then, the remaining individuals in Q are ordered based on their adjusted fitness f'. Set Q are used as parents to generate offspring that will repopulate P.

Some notes on the parent selection algorithm are worth mentioning. First, notice that $T_i.descendants$ sets the maximum number of times that an individual can be used to parent offspring. Second, the parameter p_{worst} determines the overlap between consecutive generations.

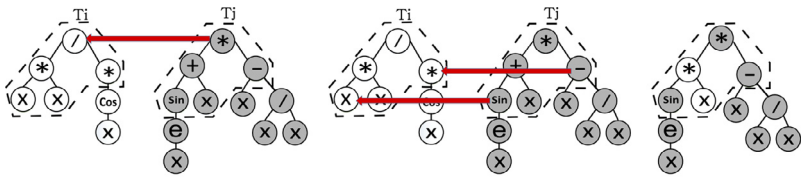
3.3.2. Apply genetic operators and generate offspring

The process by which the genetic operators are applied is summarized in Algorithm 3, taking as input the set of parents Q that will be used to generate a set of offspring R of size n. Basically, set Q is iterated until n individuals have been generated, determining which genetic operator to use based on the operator probabilities.

Individual trees T_i are selected from Q based on an equiprobable random decision of either selecting: (a) the best solution in Q or (b) a randomly chosen individual from Q. Afterward, a random decision is made to either apply mutation or crossover, based on their respective probabilities p_m and p_c. If mutation is chosen then the selected tree is mutated and inserted into the offspring set R. On the other hand, if crossover is chosen then the selected individual T_1 is considered as the first parent, while the second parent T_2 is selected based on two possibilities. First, we take species S to which T_1 belongs as reference, and then select T_2 as the individual that also belongs to species S and has the best fitness relative to all other individuals T_u that also belong to species S. If, on the other hand, no other individual in Q belongs to species S, then a random individual in Q is used as the second parent T_2.

After each selection event, the number of expected descendants for the selected individuals is reduced by 1 if it is used in a mutation and by 0.5 if it is used in crossover since crossover only generates one offspring. Finally, if the number of descendants for an individual T is equal or less then zero then T is removed from Q and cannot be chosen again as a parent.

For mutation, *neat*-GP uses standard subtree mutation, while for crossover two different operators are tested. Standard subtree crossover and we propose a NEAT-like crossover for GP trees, depicted in Fig. 5 and referred to as NEAT-Crossover. The



(a) Swap internal nodes (b) Swap tree-branches (c) Offspring

Fig. 5. The proposed NEAT-Crossover for GP, where the dotted line denotes the shared topological structure $S_{i,j}$ between parents T_i and T_j. (a) The swap of a single internal node within $S_{i,j}$. (b) The exchange of branches at the boundary of $S_{i,j}$. (c) The resulting offspring from this example.

Table 1
Algorithms used in the experimental evaluations.

Method	Description
GP	Standard GP search used as a control method.
FlatOE	Flat Operator Equalization method [23].
neat-GP	The full *neat*-GP algorithm.
neat-GP-SC	*neat*-GP subtree crossover is used instead of NEAT-Crossover.
neat-GP-Spe	*neat*-GP without mating restrictions; the decision in block G is set to *NO*.
neat-GP-Sel	*neat*-GP with tournament selection; block E uses tournament selection to construct *Q*.
neat-GP-FS	*neat*-GP without fitness sharing; blocks *D* and *J* are omitted.

proposed operator is similar to the one-point crossover and homologous crossover previously used in other GP systems [21]. NEAT-Crossover first identifies the common region $S_{i,j}$ between two parent trees T_i and T_j. Then, genetic material is taken with equal probability from each parent in the following way. Nodes with equal arity are taken randomly from each parent, as well as tree branches rooted at leaf nodes of $S_{i,j}$, as depicted in Fig. 5.

3.4. Survival and replacement

Finally, this section summarizes the survival and replacement strategy in Block K of the *neat*-GP flowchart presented in Fig. 3. Summarizing, the initial population P was speciated with Algorithm 1, and the population is evaluated and fitness sharing is applied. Afterward, a set of parents Q was constructed with Algorithm 2, to generate a set R of offspring using Algorithm 3. Speciation is then performed with Algorithm 1 using $P \cup R$ as input, to assign species membership to the newly generated offspring, then their fitness is assigned and fitness sharing is performed. Only the best individuals in R are included into the new population, to replace the $p_{worst}\%$ individuals from the previous generation. Finally, fitness sharing is reapplied to the population, in particular to the surviving individuals from the previous generation, to account for the fact that species membership has been modified when the new offspring were inserted into the population.

4. Experimental work

4.1. Experimental setup

The proposed *neat*-GP algorithm is implemented using the Matlab GPLab toolbox developed by Silva and Almeida [24]. The GPLab-based implementation is freely available at our team's homepage http://www.tree-lab.org/,[5] along with an implementation that can be run over the DEAP framework for Python [5].

Different variants of the algorithm are tested, to illustrate the effect that each component has on performance, regarding test fitness and bloat; these variants are: the full *neat*-GP algorithm as described in the preceding section; *neat*-GP-SC that uses standard subtree crossover instead of the proposed NEAT-Crossover; *neat*-GP-Spe that omits the mating restriction due to speciation; *neat*-GP-Sel that uses standard tournament selection instead of the proposed selection algorithm; and finally *neat*-GP-FS which does not employ fitness sharing. Additionally, a standard tree-based GP is used as a control method and the Flat-OE method is also included for comparison; all the tested algorithms are summarized in Table 1. For clarity regarding the *neat*-GP variants, Table 1 uses the flowchart of Fig. 3 as reference, and states which blocks are omitted or modified.

Two sets of problems are used to test the referred algorithms. First, nine symbolic regression problems are chosen based on the suggestions made in [14,15,17,32,35]; they are summarized in Table 2. Second, five real-world classification problems are used, taken from the well-known UCI machine learning repository [2]. The classification problems are summarized in Table 3.

For all problems, the search parameters for GP are given in Table 4, and all *neat*-GP variants use the parameters in Table 5. It is important to note that beside the algorithmic differences between GP and *neat*-GP, all other shared parameter values are the same except for the maximum depth of the initial population. As stated before, NEAT suggests that the best approach is to

[5] http://www.tree-lab.org/index.php/resources-2/downloads/open-source-tools.

Table 2
Symbolic regression benchmark problems.

No	Problem	Function	Fitness/test cases	Function set
1	*Koza*-1	$x^4 + x^3 + x^2 + x$	20 random $\subseteq [-1, 1]$	*Table 4*
2	*Nguyen*-3	$x^5 + x^4 + x^3 + x^2 + x$	20 random $\subseteq [-1, 1]$	*Table 4*
3	*Nguyen*-5	$\sin(x^2) * \cos(x) - 1$	20 random $\subseteq [-1, 1]$	*Table 4*
4	*Nguyen*-7	$\ln(x + 1) + \ln(x^2 + 1)$	20 random $\subseteq [0, 2]$	*Table 4*
5	*Nguyen*-10	$2\sin(x) * \cos(y)$	100 random $\subseteq [-1, 1] \times [-1, 1]$	*Table 4*
6	*Keijzer*-6	$\sum_i^x \frac{1}{i}$	$E[1, 50, 1]$, $E[1, 120, 1]$	*Keijzer*
7	*Korns*-12	$2 - 2.1\cos(9.8x)\sin(1.3w)$	$U[-50, 50, 10000]$	*Korns*
8	*Vladislavleva*-1	$\frac{e^{-(x-1)^2}}{1.2+(y-2.5)^2}$	$U[0.3, 4, 100]$	*Vladislavleva-B*
9	*Pagie*-1	$\frac{1}{1+x^{-4}} + \frac{1}{1+y^{-4}}$	$E[-5, 5, 0.4]$	*Koza*

Table 3
Real-world classification problems for standard GP and *neat*-GP.

Problem	Classes	Features	Samples
UCI2 (Breast cancer Wisconsin)	2	8	441
UCI16 (Ionosphere)	2	32	350
UCI20 (Parkinson's)	2	22	195
UCI21 (Pima Indians Diabetes)	2	8	768
UCI22 (Sonarall)	2	60	208

Table 4
Parameters used in benchmark problems with standard GP.

Parameter	Description
Population size	500 for regression 200 for classification
Generations	100 generations for regression 200 for classification
Initialization	*Ramped Half-and-Half*, with 6 levels of maximum depth
Operator probabilities	Crossover $p_c = 0.7$, Mutation $p_\mu = 0.3$
Function set (regression)	$\{+, -, \times, \div, \sin, \cos, \exp, \log\}$, *Keijzer*, *Korns*, *Vladislavleva-B* and *Koza*.
Function set (classification)	$\{+, -, \times, \div, \sin, \cos, \exp, \sqrt[y]{x}, |x|, \text{if}\}$
Terminal set (regression)	x, 1 for single variable problems and x, y for bivariable problem
Terminal set (classification)	Problem features
Initial dynamic depth	6 levels
Hard maximum depth	20 levels
Selection	Tournament selection of size 3
Elitism	Best individual always survives

Table 5
Parameters used in benchmark problems with *neat*-GP.

Parameter	Description
Population size	500 for regression 200 for classification
Generations	100 generations for regression 200 for classification
Initialization	*Full initialization*, with 3 levels of maximum depth
Operator probabilities	NEAT-Crossover $p_c = 0.7$, Mutation $p_\mu = 0.3$
Function set (regression)	$\{+, -, \times, \div, \sin, \cos, \exp, \log\}$, *Keijzer*, *Korns*, *Vladislavleva-B* and *Koza*.
Function set (classification)	$\{+, -, \times, \div, \sin, \cos, \exp, \sqrt[y]{x}, |x|, \text{if}\}$
Terminal set (regression)	x, 1 for single variable problems and x, y for bivariable problem
Terminal set (classification)	problem features
Initial dynamic depth	3 levels
Hard maximum depth	20 levels
Selection	Eliminate the worst individuals of each species by the factor of $p_{worst}\%$
Elitism	Do not penalize the best individual of each species
Survival threshold	0.5
Specie threshold value	$h = 0.15$ with $\alpha = 0.5$

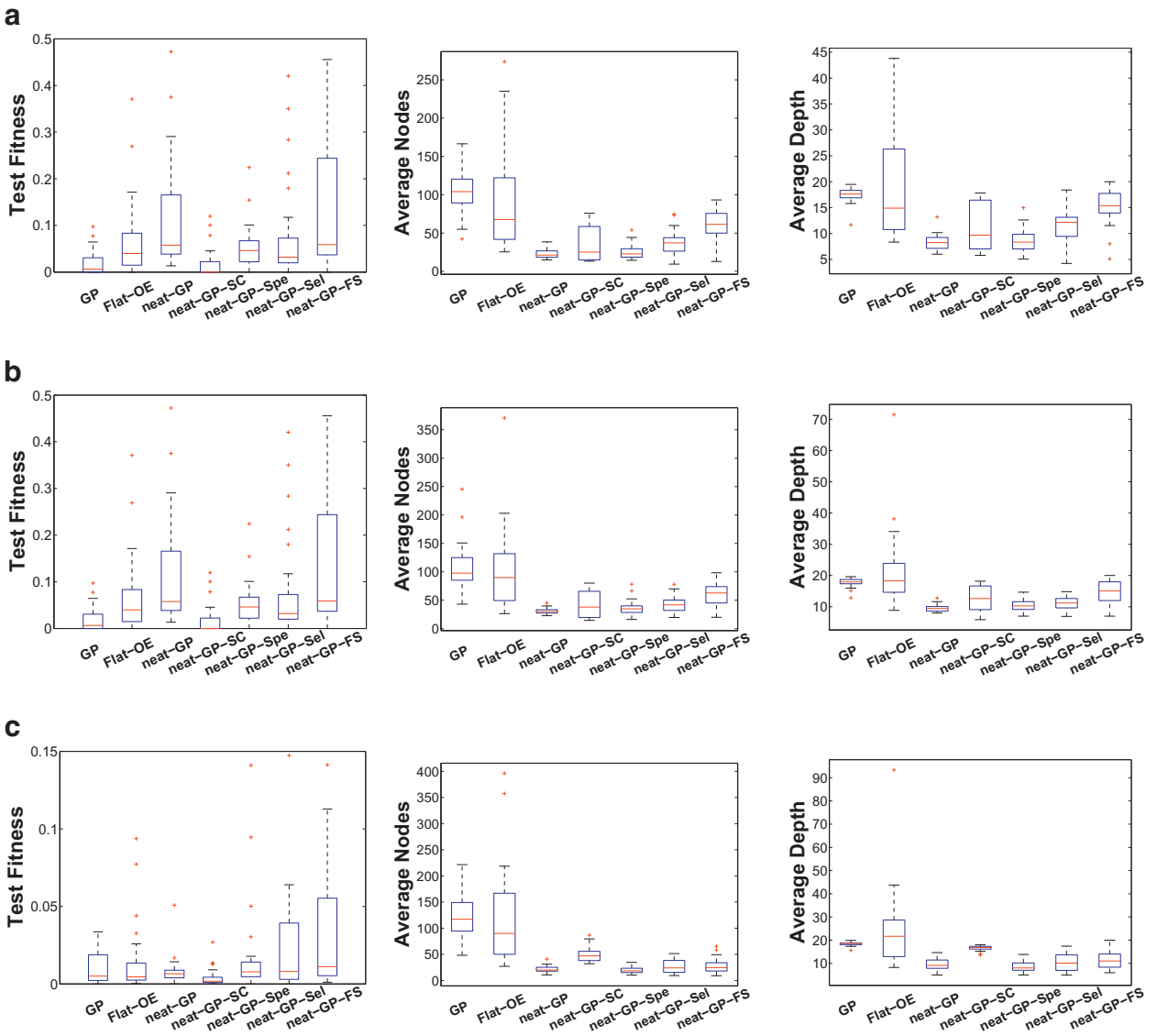


Fig. 6. Box plots for symbolic regression problems, that show: Test Fitness (first column), Nodes (second column) and Depth (third column). Each row is for a different problem: (a) Koza-1, (b) Nguyen-3 and (c) Nguyen-5.

start with small or simple solutions in the initial population, while most GP works use an initial depth between 5 and 7 levels. In this work, several exploratory experiments for GP were carried out using a maximum of 3-levels for initial depth, this produced poor performance, worse for GP on all the problems reported here while bloat was also not reduced. For these reasons, and to maintain the results as concise as possible, results are only presented for the configurations given in Tables 4 and 5.

Flat-OE is implemented based on the Dynamic OE algorithm [26], using the same parameters specified in Table 4, however tournament size is set to 10 for best performance, and the bin size is 1 (used to force the flat distribution). In Flat-OE much of the computation time is used to generate individuals of a particular size such that the flat distribution is maintained. Therefore, instead of using a fixed number of generations we set a maximum number of individuals generated by the search. To maintain a fair comparison, after performing 30 runs of *neat*-GP on each problem we took the largest number of total individuals generated by the search in any run of each problem domain. For regression problems the largest number was 50,000 and for classification it was 40,000, after rounding to the nearest thousand. These values are used to determine the maximum number of evaluated individuals for Flat-OE in each domain.

For symbolic regression fitness is computed as the root mean square error between predicted and expected outputs, and for classification the total error is used. Thirty independent runs are performed for each problem, with random training and testing sets in each run. The algorithms are compared based on the test error of the best solution found, the average size and the average depth of the individuals in the population, and the size of the best solution found. Results are presented as median values over

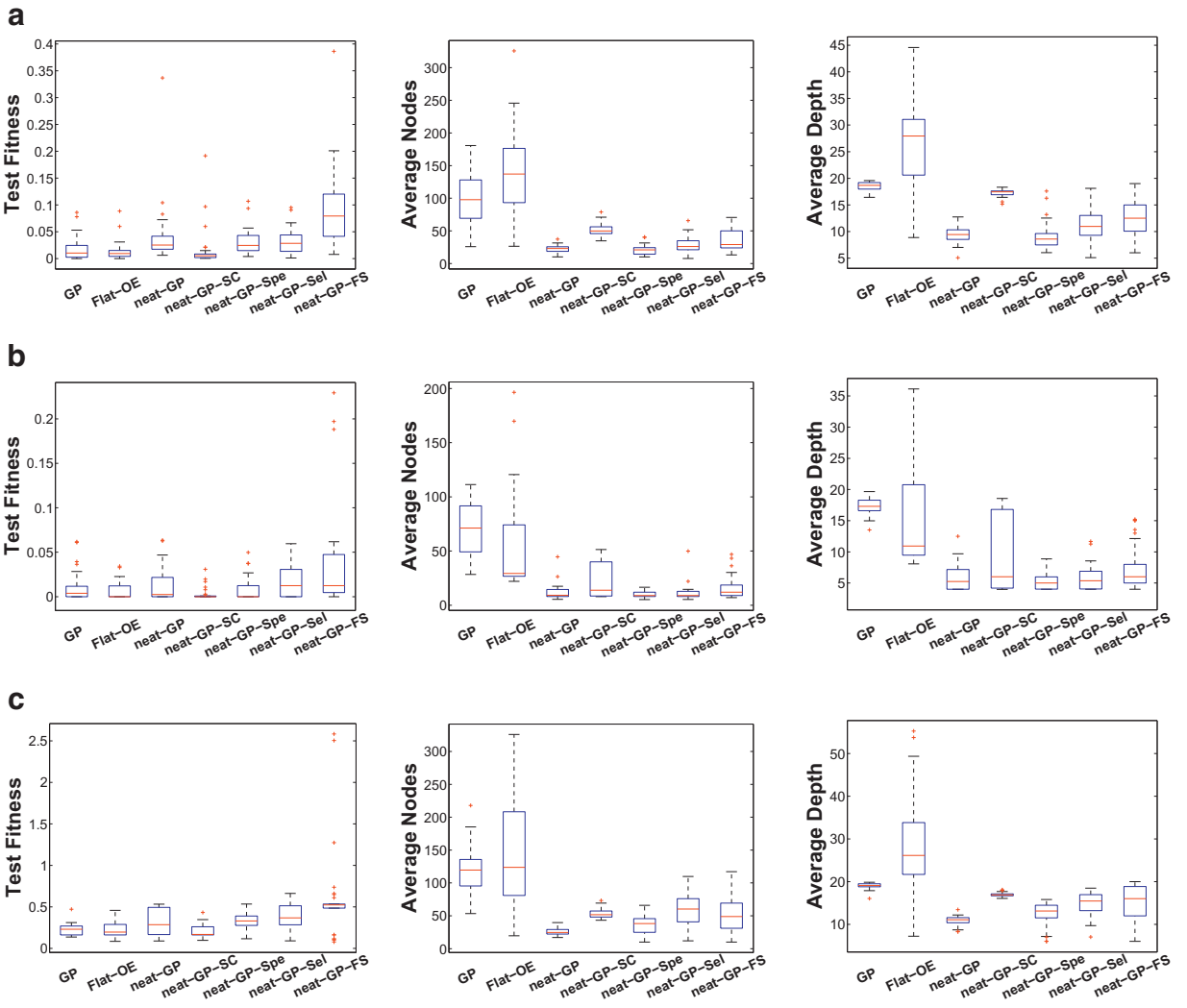


Fig. 7. Box plots for symbolic regression problems, that show: Test Fitness (first column), Nodes (second column) and Depth (third column). Each row is for a different problem: (a) Nguyen-7, (b) Nguyen-10 and (c) Keijzer-6.

all runs. Statistical comparisons are carried out using a $1 \times N$ formulation, where a single control method (GP) is compared with N algorithms. We use the Friedman test and the Bonferroni–Dunn correction of the p-values for each comparison, as suggested by Derrac et al. [3]. For each problem domain two tables are used to summarize the median performance values and the p-values resultant from the statistical tests.

4.2. Results: symbolic regression

Figs. 6–8 show the results for the nine symbolic regression benchmarks, showing box plots for all thirty runs measuring performance based on: (1) value of test fitness for the best solution;[6] (2) average population size based on number of nodes; and (3) average depth. Table 6 presents the comparison based on the median performance of each algorithm, and Table 7 presents the p-values of the statistical tests.

The results show some clear trends. Regarding fitness, it is reasonable to state that the goal of a bloat control method is to reduce the average size of the evolving population without incurring in a performance decrease. All *neat*-GP variants produce significantly smaller average tree sizes than GP, based on total nodes and tree depth. However, only *neat*-GP-SC achieves equal or better performance than GP on all problems based on test fitness. On the other hand, Flat-OE exhibits a larger performance

[6] Note that some boxplots show skewed distributions, where the minimum, first quartile and median are all 0; this is possible since the minimum possible error is always 0.

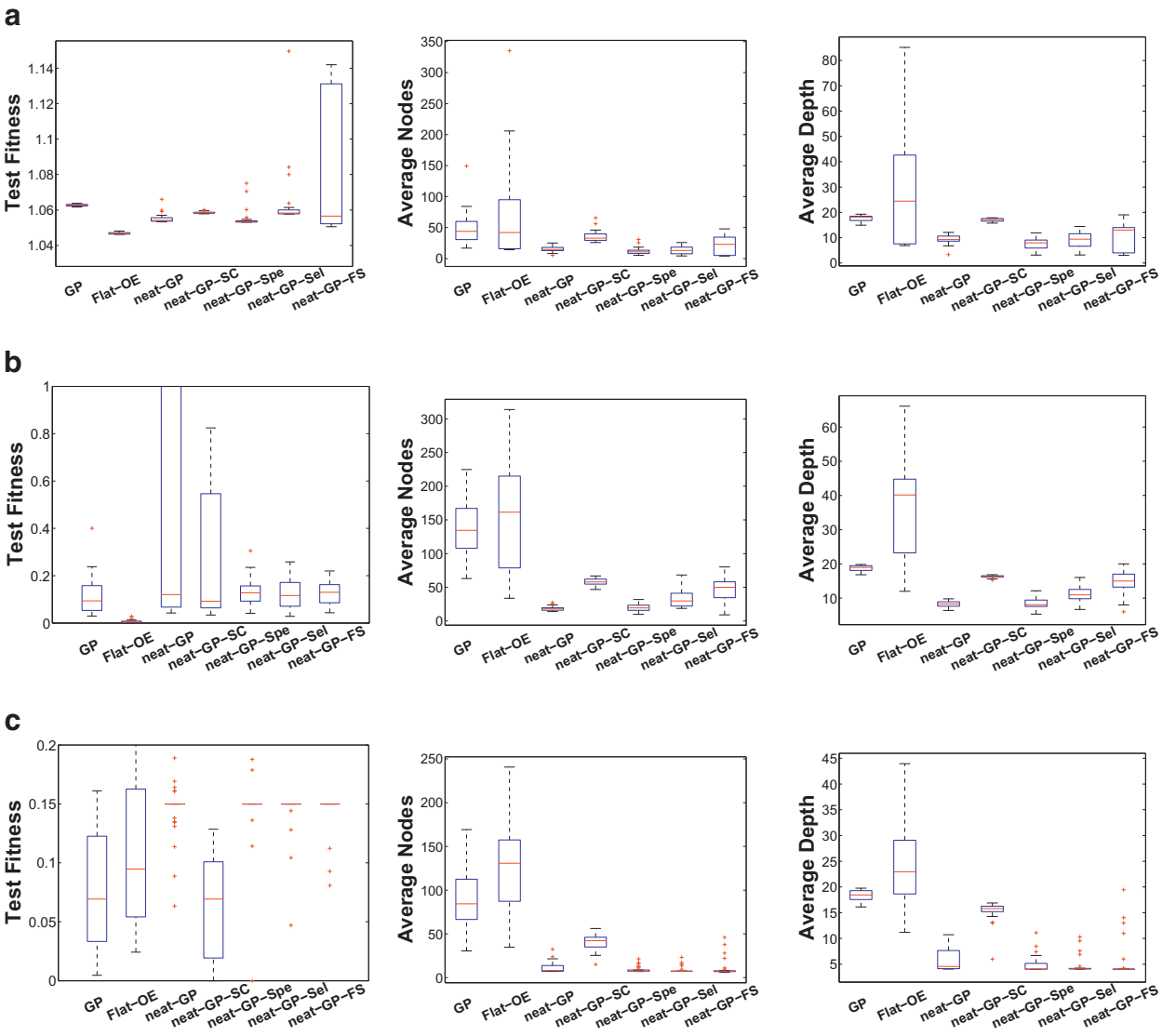


Fig. 8. Box plots for symbolic regression problems, that show: Test Fitness (first column), Nodes (second column) and Depth (third column). Each row is for a different problem: (a) Korns-12, (b) Vladislavleva-1 and (c) Pagie-1.

variant, in two cases outperforming GP (Korns-12 and Vladislavleva-1) and in two cases performing significantly worse (Koza-1 and Nguyen-3).

One of the main reasons for performing bloat control, is to help the search find good solutions that are also small and interpretable. Based on the size of the best solutions found we can see that *neat*-GP variants also produce smaller solutions. Focusing on *neat*-GP-SC that achieved the best test performance, it produces significantly smaller solutions than GP in 4 problems, and consistently outperforms Flat-OE. In some cases these differences are quite large, such as in Vladislavleva-1, Nguyen-5, Nguyen-10 and Pagie-1.

4.2.1. Search dynamics

To illustrate the search dynamics of the *neat*-GP algorithm, Figs. 9–13 show the program size distribution over all of the generations for five of the problems (Nguyen-3, Nguyen-5, Nguyen-7, Nguyen-10 and Vladislavleva-12). The plots are averages over all 30 runs, where the grayscale used is linearly proportional to the total number of programs of a particular size that are present at each generation, such that darker regions represent the presence of a large number of trees and lighter regions show a small number of trees.

Figs. 9 (a)–13(a) show that the behavior of GP is similar in all problems. The size of the programs quickly increases at the beginning of the search and it progressively moves toward larger trees as the search progresses. In most cases the search seems to generate multimodal distributions of program sizes at the end of the run. At some instances during the search the population does

Table 6
Comparison of the median values for each performance criterion on symbolic regression problems. Bold indicates the best (lowest) value and an asterisk (*) indicates that the null hypothesis is rejected at the $\alpha = 0.05$ confidence level.

Method	GP	Flat-OE	neat-GP	neat-GP-SC	neat-GP-Spe	neat-GP-Sel	neat-GP-FS
Koza-1							
Test	**0.0**	0.0185*	0.0779	**0.0**	0.0183	0.0418*	0.0753*
AVG size	104.1	67.7	**21.2***	25.0*	22.8*	37.2*	61.3*
AVG level	17.6	14.8	**8.2***	9.6*	8.3*	12.1*	15.3
Best size	**19.0**	90.5	30.5	21.5	27.0	43.0	65.0
Nguyen-3							
Test	0.0063	0.03988*	0.0576*	**0.0**	0.0461*	0.0320	0.0586*
AVG size	97.6	89.7	**30.1***	38.0*	34.8*	41.8*	62.7*
AVG level	17.9	18.2	**9.4***	12.6*	10.2*	11.2*	15.0
Best size	75.0	119.0*	50.0	**36.0**	39.0	48.0	65.5
Nguyen-5							
Test	0.0050	0.0046	0.0065	**0.0017**	0.0078	0.0080	0.0111*
AVG size	117.2	90.0	20.0*	47.1*	**18.2***	24.1*	24.5*
AVG level	18.6	21.7	9.1*	16.8*	**8.0***	10.1*	11.0*
Best size	100.5	133.5	23.5*	41.5*	**20.0***	25.5*	24.5*
Nguyen-7							
Test	0.0103	0.0094	0.0253*	**0.0052**	0.0245*	0.0285*	0.0793*
AVG size	97.8	137.1	23.1*	49.8*	**20.9***	26.2*	29.1*
AVG level	18.7	28.0*	9.4*	17.5*	**8.6***	11.0*	12.5*
Best size	111.0	155.0	28.0*	71.0	**25.5***	32.0*	30.0*
Nguyen-10							
Test	0.0037	**0.0**	0.0023	**0.0**	**0.0**	0.0124	0.0124
AVG size	71.2	29.3	9.2*	13.9*	**9.0***	**9.0***	12.0*
AVG level	17.3	10.9	5.2*	6.0*	**5.0***	5.4*	6.0*
Best size	45.5	26.5	12.0*	12.5	**9.0***	**9.0**	12.0
Keijzer-6							
Test	0.2301	0.1963	0.2855	0.1680	0.3306*	0.3676*	0.5234*
AVG size	119.5	123.8	24.6*	51.7*	38.2*	60.3*	48.8*
AVG level	19.1	26.2*	11.1*	16.9*	13.1*	15.5*	16.0*
Best size	125.5	192.0	38.0*	75.5*	42.5*	78.0*	51.0*
Korns-12							
Test	1.0627	**1.0467***	1.0541*	1.0585*	1.0533*	1.0582*	1.0565
AVG size	44.0	42.0	14.6*	32.9	**10.9***	13.0*	23.0*
AVG level	18.1	24.4	9.3*	17.2	**8.0***	9.4*	13.0*
Best size	55.0	58.5	18.0*	40.0	**11.5***	15.5*	22.5*
Pagie-1							
Test	**0.0692**	0.0947	0.1498*	**0.0692**	0.1498*	0.1498*	0.1498*
AVG size	84.3	130.8*	8.3*	42.3*	7.2*	7.3*	**7.0***
AVG level	18.5	23.0	4.6*	15.8*	4.1*	4.1*	**4.0***
Best size	85.0	151.0*	10.0*	40.5*	**7.0***	**7.0***	**7.0***
Vladislavleva-1							
Test	0.0935	**0.0042***	0.1202	0.0918	0.1277	0.1160	0.1297
AVG size	134.5	161.5	**18.5***	58.0*	19.3*	29.7*	50.0*
AVG level	19.0	40.1*	8.3*	16.4*	**8.1***	10.9*	15.0*
Best size	136.0	181.0	25.0*	67.0*	**24.5***	37.5*	53.0*

not contain programs of some sizes, as indicated by the light gray or white regions in the plots, particularly some intermediate sizes between the largest and smallest programs in the population. Such a distribution will bias the search toward the largest trees, as expected by the CBT (crossover-bias theory).

Most *neat*-GP variants show a different trend, they seem to concentrate the search within a particular range of program sizes, depicted by the regions that are consistently dark across all generations. In the initial generations all programs are small given the minimal initialization suggested by NEAT, then the search explores programs of a larger size, but after a small number of generations the runs tend to focus on what appears to be a problem-dependent range of sizes. In almost all plots of *neat*-GP variants the distribution of program sizes hits a limit after which growth no longer occurs. The only *neat*-GP variant that shows a consistent pattern of growth over most problems is *neat*-GP-FS, but at a smaller rate than GP. Moreover, in most *neat*-GP variants small program sizes are maintained throughout the search, this is particularly true for *neat*-GP-Spe and *neat*-GP-Sel. Probably the most restrictive variants are *neat*-GP-Sel and *neat*-GP-Spe, producing very small increases in program size, but these restrictions in growth are surely too severe given that these variants produce worse solutions than GP. Conversely, *neat*-GP-SC seems to consistently eliminate very small trees from the population in some problems, such as Nguyen-5, Nguyen-7 and Vladislavleva-1. It appears that small programs are eliminated given their poor performance, since *neat*-GP-SC achieved the best test fitness on

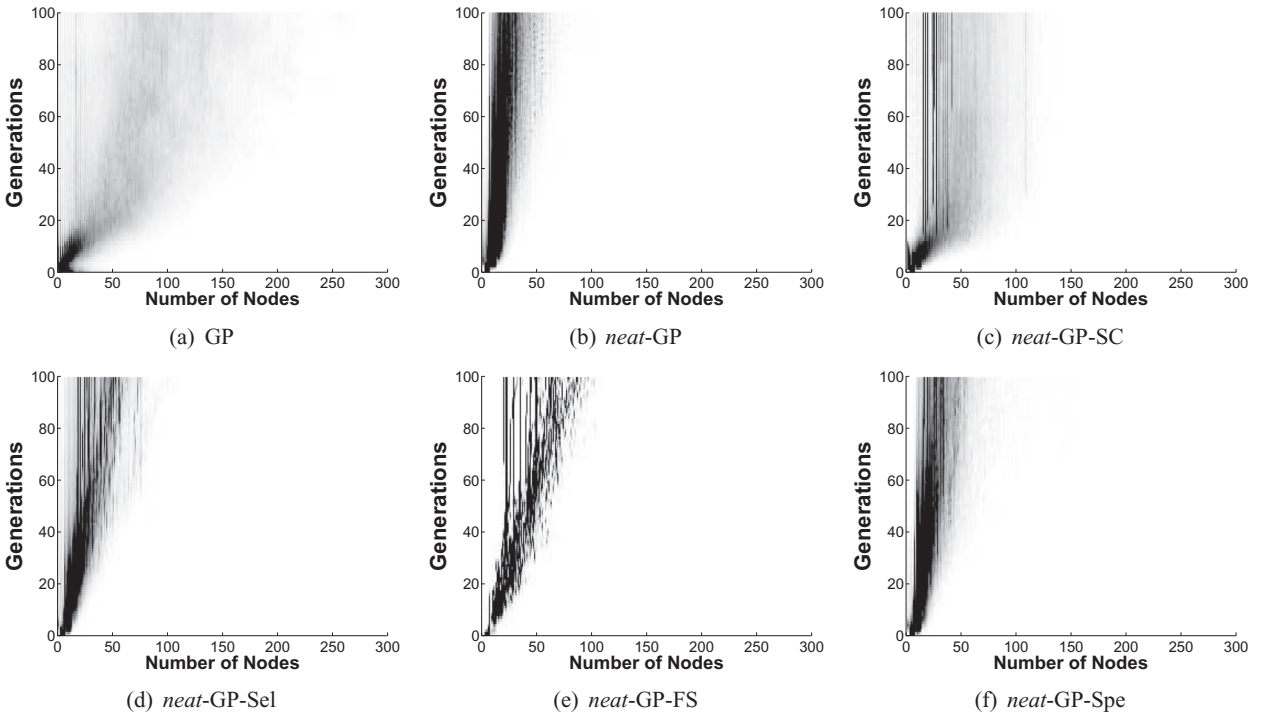
Table 7
Statistical results on symbolic regression problems, reporting the p-values of the Friedman test with Bonferroni–Dunn correction. An asterisk (*) confirms that the null hypothesis is rejected at the $\alpha = 0.05$ confidence level.

Method	Flat-OE	neat-GP	neat-GP-SC	neat-GP-Spe	neat-GP-Sel	neat-GP-FS
Koza-1						
Test	2.3623	0.0005*	3.7024	0.2972	0.0024*	0.0000*
AVG size	0.1707	0.0000*	0.0000*	0.0000*	0.0000*	0.0000*
AVG level	0.8647	0.0000*	0.0000*	0.0000*	0.0000*	0.1707
Best size	0.4073	0.4073	3.3822	1.6399	1.6399	0.1707
Nguyen-3						
Test	0.0025*	0.0003*	3.1896	0.0095*	0.0635	0.0060*
AVG size	2.7912	0.0000*	0.0000*	0.0000*	0.0000*	0.0060*
AVG level	6.0000	0.0000*	0.0000*	0.0000*	0.0000*	0.0635
Best size	0.0095*	1.5410	2.1189	1.6399	1.6399	6.0000
Nguyen-5						
Test	4.2900	2.7912	0.1707	3.4648	2.7912	0.0209*
AVG size	4.2900	0.0000*	0.0000*	0.0000*	0.0000*	0.0000*
AVG level	1.6399	0.0000*	0.0000*	0.0000*	0.0000*	0.0000*
Best size	0.4073	0.0000*	0.0003*	0.0000*	0.0000*	0.0000*
Nguyen-7						
Test	4.2900	0.0209*	0.8647	0.0060*	0.0060*	0.0000*
AVG size	0.1707	0.0000*	0.0003*	0.0000*	0.0000*	0.0000*
AVG level	0.0060*	0.0000*	0.0209*	0.0000*	0.0000*	0.0000*
Best size	0.1707	0.0000*	0.0635	0.0000*	0.0000*	0.0000*
Nguyen-10						
Test	1.3240	5.0089	0.1308	3.2310	1.4358	0.7834
AVG size	0.1707	0.0000*	0.0000*	0.0000*	0.0000*	0.0000*
AVG level	0.4073	0.0000*	0.0015*	0.0000*	0.0000*	0.0000*
Best size	4.1693	0.0233*	0.1668	0.0040*	0.0741	0.1400
Keijzer-6						
Test	4.2900	0.8647	2.7912	0.0209*	0.0015*	0.0209*
AVG size	1.6399	0.0000*	0.0000*	0.0000*	0.0000*	0.0000*
AVG level	0.0003*	0.0000*	0.0000*	0.0000*	0.0000*	0.0015*
Best size	0.1707	0.0000*	0.0000*	0.0000*	0.0015*	0.0000*
Korns-12						
Test	0.0000*	0.0000*	0.0000*	0.0000*	0.0003*	0.8647
AVG size	4.2900	0.0000*	0.8647	0.0000*	0.0000*	0.0003*
AVG level	1.6399	0.0000*	0.8647	0.0000*	0.0000*	0.0000
Best size	4.2900	0.0000*	0.8647	0.0000*	0.0000*	0.0000*
Pagie-1						
Test	0.0635	0.0000*	2.7912	0.0000*	0.0000*	0.0000*
AVG size	0.0060*	0.0000*	0.0000*	0.0000*	0.0000*	0.0000*
AVG level	0.0635	0.0000*	0.0000*	0.0000*	0.0000*	0.0000*
Best size	0.0015*	0.0000*	0.0003*	0.0000*	0.0000*	0.0000*
Vladislavleva-1						
Test	0.0000*	0.1707	2.7912	0.8647	1.6399	0.8647
AVG size	4.2900	0.0000*	0.0000*	0.0000*	0.0000*	0.0000*
AVG level	0.0003*	0.0000*	0.0000*	0.0000*	0.0000*	0.0003*
Best size	0.8647	0.0000*	0.0000*	0.0000*	0.0000*	0.0000*

most problems. The best compromise is achieved by *neat*-GP-SC, given its good bloat control performance and strong results compared with GP and Flat-OE.

4.3. Results: classification

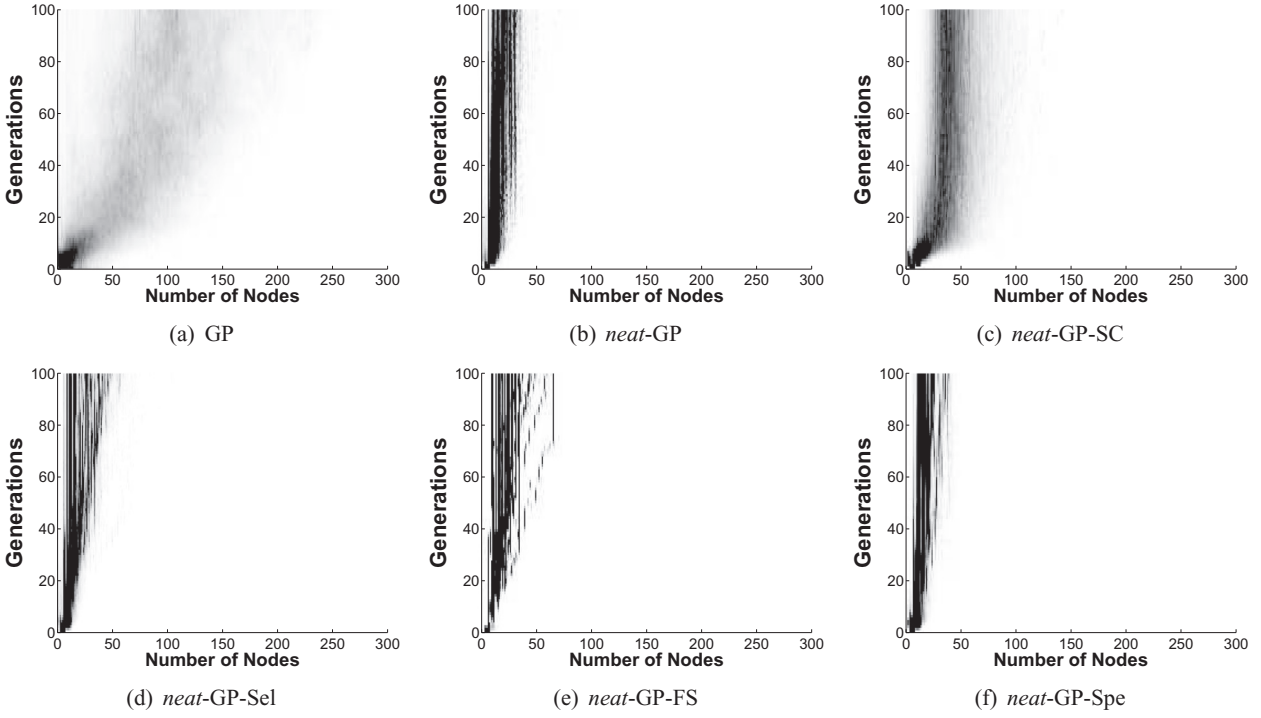
Given the better test performance of *neat*-GP and *neat*-GP-SC relative to all other *neat*-GP tested in symbolic regression, only these two methods are compared with GP and Flat-OE on the real world classification problems. Fig. 14 shows a box plot comparison using the same performance measures as before, except that test performance is given by the total classification error (percentage of misclassified samples). The numerical comparisons are presented in Table 8 and the p-values produced by the statistical tests are given in Table 9. For these problems *neat*-GP-SC achieves basically the same test error as GP, but it is surprisingly worse based on program size. This trend is also apparent for Flat-OE. It is reasonable to state that neither *neat*-GP-SC or Flat-OE can control bloat in this domain. On the other hand, the full *neat*-GP method also achieves equivalent test error relative to standard GP, but also induces a significant reduction in average program size and size of the best solution. Indeed, in all problems *neat*-GP produces the smallest solutions, and in some cases the differences are very large (UCI20 and UCI22).



(a) GP (b) *neat*-GP (c) *neat*-GP-SC

(d) *neat*-GP-Sel (e) *neat*-GP-FS (f) *neat*-GP-Spe

Fig. 9. Program size distribution averaged over all 30 runs for the Nguyen-3 benchmark.

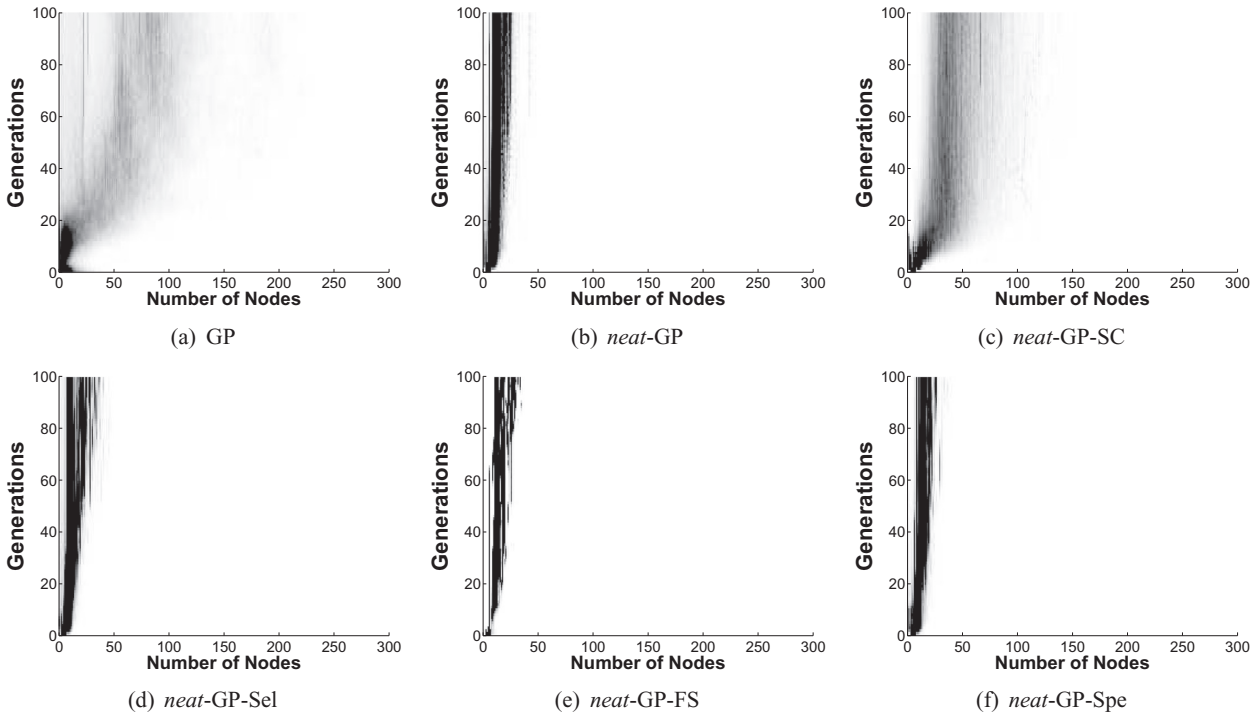


(a) GP (b) *neat*-GP (c) *neat*-GP-SC

(d) *neat*-GP-Sel (e) *neat*-GP-FS (f) *neat*-GP-Spe

Fig. 10. Program size distribution averaged over all 30 runs for the Nguyen-5 benchmark.

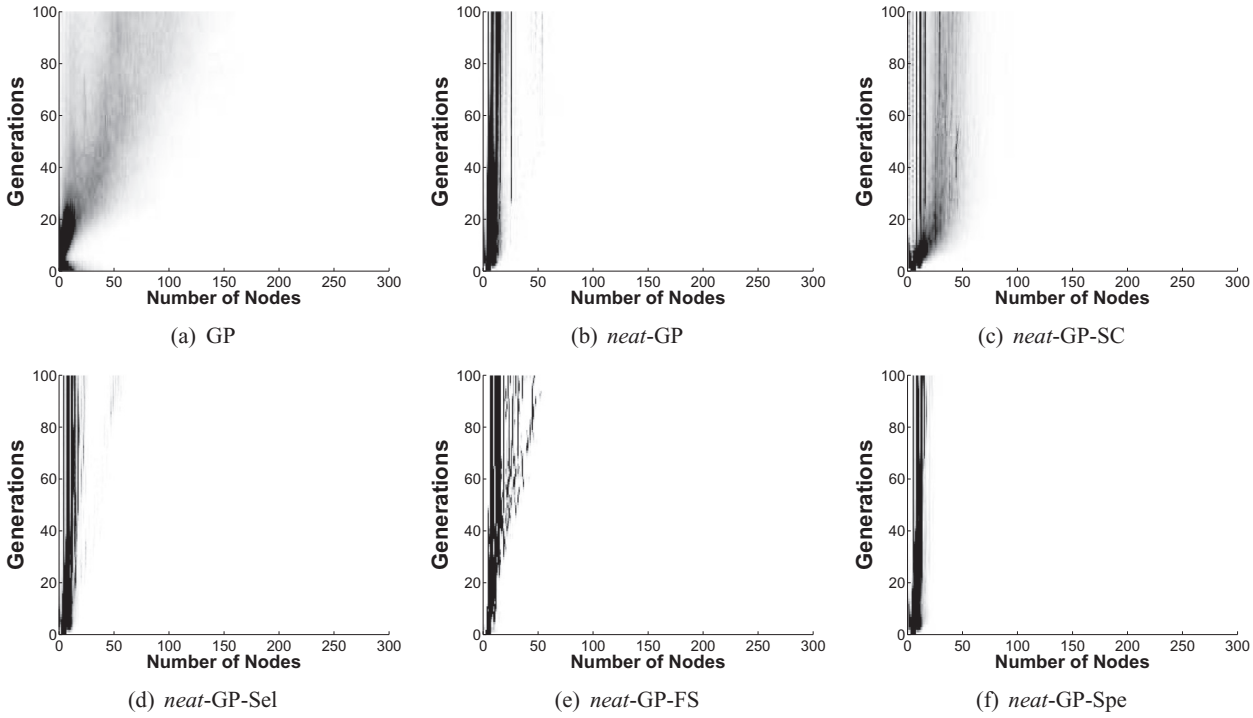
neat-GP shows a much better test error than *neat*-GP-SC, which is equivalent to standard GP, while substantially outperforming both methods in terms of program size and tree depth, effectively controlling bloat in all test problems. Flat-OE is very similar to standard GP, both in terms of classification error, size and depth, an unexpected result.



(a) GP (b) *neat*-GP (c) *neat*-GP-SC

(d) *neat*-GP-Sel (e) *neat*-GP-FS (f) *neat*-GP-Spe

Fig. 11. Program size distribution averaged over all 30 runs for the Nguyen-7 benchmark.

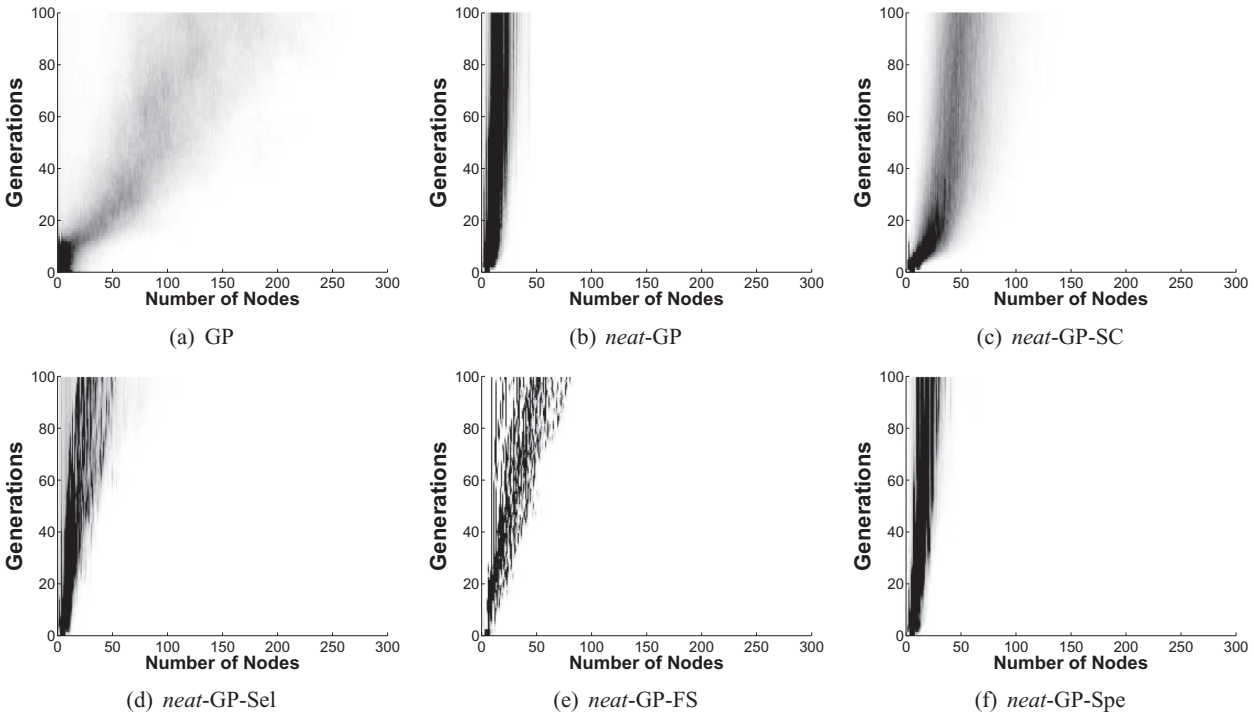


(a) GP (b) *neat*-GP (c) *neat*-GP-SC

(d) *neat*-GP-Sel (e) *neat*-GP-FS (f) *neat*-GP-Spe

Fig. 12. Program size distribution averaged over all 30 runs for the Nguyen-10 benchmark.

In fact, the bloat control of *neat*-GP is more clearly seen if we only consider the size of the best program found in each run; these results are summarized in Table 8. The decrease in program size is quite large in some cases, with median size decreasing from between 46% to as much as 88%, when compared with GP.



(a) GP (b) *neat*-GP (c) *neat*-GP-SC

(d) *neat*-GP-Sel (e) *neat*-GP-FS (f) *neat*-GP-Spe

Fig. 13. Program size distribution averaged over all 30 runs for the Vladislavleva-1 benchmark.

4.4. Computational efficiency

Finally, to assess the comparative computational costs of each method, we compare the methods based on a speedup ratio given by T_{neatGP}/T_{GP}, where T_{neatGP} represents the median run time of each *neat*-GP experiment, and T_{GP} is the median run time for GP. These tests were performed under equal conditions using an Intel dual-core PC with 4 GB of RAM and disabling all non-essential OS services before running Matlab without a GUI. In this test, *neat*-GP gives a 20 $\times$ speedup compared to standard GP, while the less efficient *neat*-GP-SC gives a 2 $\times$ speedup. Therefore, it can be said that *neat*-GP provides a substantial efficiency improvement with respect to a standard search, with no performance loss in classification tasks. While *neat*-GP-SC gives the same performance as GP on symbolic regression, with reduced bloating and no extra computational cost. This result is important, when compared with a state-of-the-art method such as OE or Flat-OE. It is reasonable to assume that the performance speedups are largely due to the total reduction in program sizes, since at each generation *neat*-GP has to process a substantially smaller number of total nodes, given the average program size relative to the standard GP search.

4.5. Discussion

The experimental results presented in the preceding subsection are clear, *neat*-GP is able to reduce code growth without decreasing GP performance and in some cases improving the quality of the evolved solutions. Moreover, the *neat*-GP variants tend to generate a distribution of program sizes that is consistent with Flat-OE. Indeed, we observe that some of the *neat*-GP variants, particularly *neat*-GP and *neat*-GP-SC can control code growth and focus the search within an almost constant range of program sizes. The range of program sizes varies with each problem and is not defined a priori, suggesting that *neat*-GP is able to automatically focus the search on promising regions of the search space.

The experimental work also considered the contribution made by each of the main components within *neat*-GP. In general, the most crucial components seem to be the selection mechanism, fitness sharing and mating restrictions based on species membership. In all three cases, tested by *neat*-GP-Sel, *neat*-GP-FS and *neat*-GP-Spe, when a component is removed and a standard approach is used instead, the quality of the solutions is compromised. While bloat can in fact be controlled by each variant, the performance on test data is reduced significantly, particularly for *neat*-GP-FS. These results suggest that all three mechanisms help improve search performance.

However, crossover, often considered as the main search operator, seems to be domain dependent. On the one hand, standard crossover seems to enhance performance on symbolic regression problems, as evidenced by the *neat*-GP-SC variant that outperforms both GP and Flat-OE. On the other hand, for classification problems the full *neat*-GP method clearly achieves the best results, matching GP performance based on test error and controlling bloat better than *neat*-GP-SC and Flat-OE. Hence, it seems that each crossover operator has a clear domain of competence, when combined with the proposed *neat*-GP search.

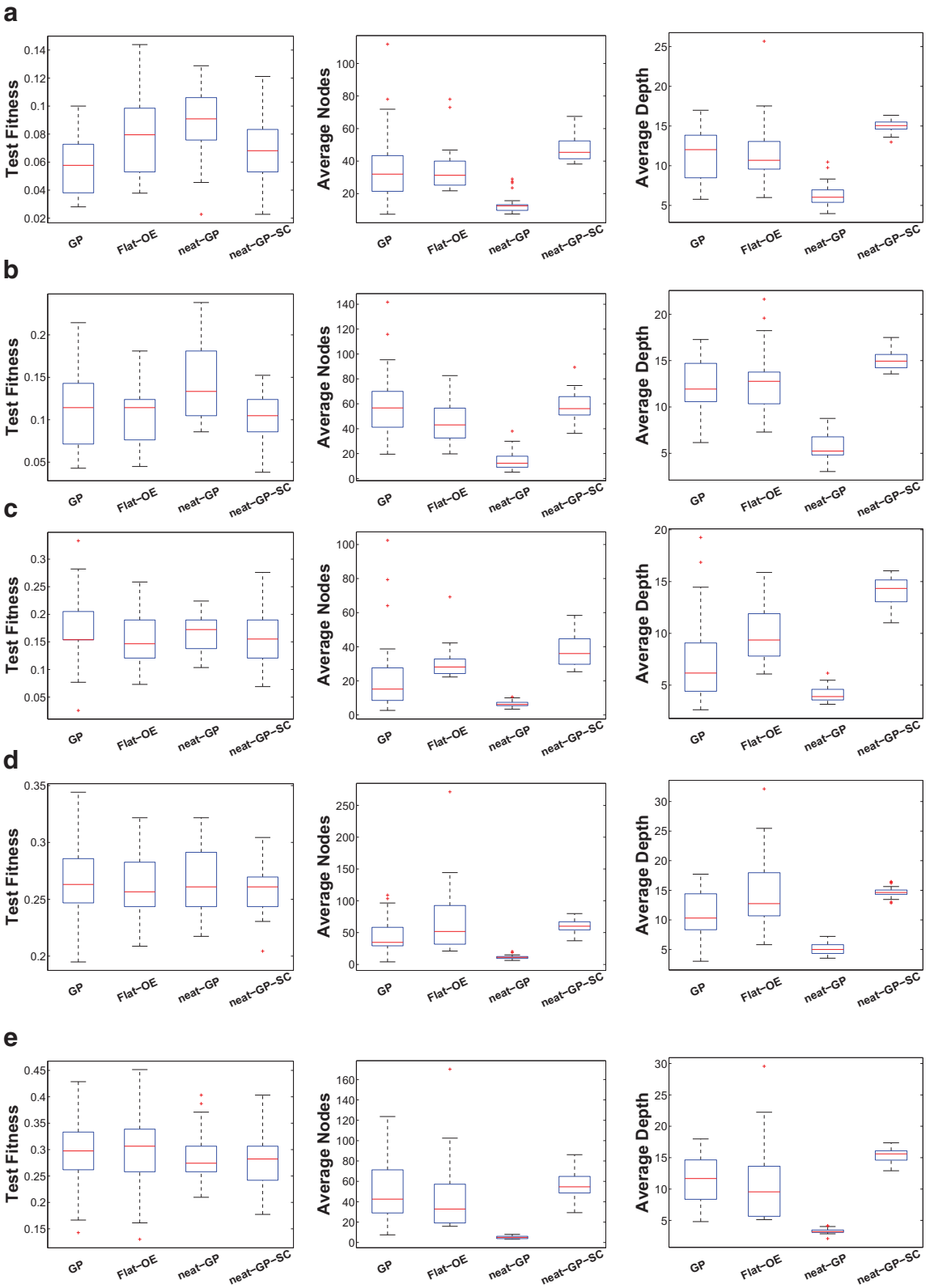


Fig. 14. Box plots for the classification problems, that show: Test Fitness (first column), Nodes (second column) and Depth (third column). Each row is for a different problem: (a) Breast cancer, (b) Ionosphere, (c) Parkinson's, (d) Pima and (e) Sonarall.

Table 8
Statistical results on classification problems, reporting the p-values of the Friedman test with Bonferroni–Dunn correction. Bold indicates the best (lowest) value and an asterisk confirms that the null hypothesis is rejected at the $\alpha = 0.05$ confidence level.

Method	GP	Flat-OE	neat-GP	neat-GP-SC
UCI2 (Breast cancer Wisconsin)				
Test	**0.0577**	0.0795*	0.0909*	0.0681*
AVG size	32.0	31.3	**12.5***	45.4*
AVG level	12.0	10.7	**6.0***	15.0*
Best size	41.5	49.0	**17.5***	61.5*
UCI16 (Ionosphere)				
Test	0.1142	0.1142	0.1333	**0.1047**
AVG size	56.7	43.0	**12.2***	56.1
AVG level	11.9	12.8	**5.2***	14.9*
Best size	59.0	56.0	**14.0***	68.0
UCI20 (Parkinson's)				
Test	0.1538	**0.1465**	0.1724	0.1551
AVG size	15.2	28.1	**6.2***	36.1*
AVG level	6.2	9.4	**3.9***	14.3*
Best size	22.0	32.5	**9.0***	48.0*
UCI21 (Pima Indians Diabetes)				
Test	0.2629	**0.2565**	0.2608	0.2608
AVG size	34.7	51.8	**11.1***	60.0*
AVG level	10.3	12.8*	**5.0***	14.6*
Best size	42.5	75.0	**19.0***	73.5*
UCI22 (Sonarall)				
Test	0.2976	0.3064	**0.2741**	0.2822
AVG size	42.6	32.8	**4.9***	54.7
AVG level	11.7	9.6	**3.2***	15.6*
Best size	50.0	43.5	**6.0***	74.5

Table 9
Comparison of the median values for each performance criterion on classification problems. An asterisk (*) indicates that the null hypothesis is rejected at the $\alpha = 0.05$ confidence level.

Method	Flat-OE	neat-GP	neat-GP-SC
UCI2 (Breast cancer Wisconsin)			
Test	0.0317*	0.0007*	0.0104*
AVG size	3.0	0.0000*	0.0104*
AVG level	3.0	0.0000*	0.0030*
Best size	0.2336	0.0000*	0.0001*
UCI16 (Ionosphere)			
Test	1.7324	0.1232	1.7324
AVG size	1.3956	0.0000*	1.3956
AVG level	2.1450	0.0000*	0.0030*
Best size	3.0	0.0000*	0.1232
UCI20 (Parkinson's)			
Test	0.8199	1.3956	0.2036
AVG size	0.0853	0.0104*	0.0030*
AVG level	0.0853	0.0317*	0.0001*
Best size	1.7324	0.0000*	0.0007*
UCI21 (Pima Indians Diabetes)			
Test	2.1450	3.0	0.8199
AVG size	0.2036	0.0000*	0.0030*
AVG level	0.0104*	0.0000*	0.0030*
Best size	0.0853	0.0000*	0.0030*
UCI22 (Sonarall)			
Test	2.1450	0.4323	1.3956
AVG size	1.3956	0.0000*	0.8199
AVG level	1.3956	0.0000*	0.0007*
Best size	3.0	0.0000*	0.0853

5. Conclusions and future work

This paper presents a new GP algorithm called *neat*-GP, that incorporates some of the main features of the NEAT algorithm and is able to implicitly shape the program size distribution during the search process. The method is loosely based on two well-known methods in evolutionary computation, Flat-OE and NEAT. It uses a similar overall strategy as the one proposed in Flat-OE, maintaining a diverse population of programs in terms of size throughout the search. However, while Flat-OE explicitly forces a flat distribution, *neat*-GP accomplishes this implicitly. To achieve this, *neat*-GP is partially implemented as a simplified version of the NEAT algorithm, which was originally developed for neuroevolution applications. Its main components are: (1) to seed the initial population only using simple and/or small individuals; (2) enforce diversity within the population through fitness sharing; and (3) to restrict the mating process based on speciation. Experimental results show strong performance for *neat*-GP on a total of 14 different problems, including nine symbolic regression benchmarks and five real-world classification tasks. *neat*-GP search can significantly reduce code growth, based on both average program size and size of the best solution, while basically matching or improving test fitness relative to standard GP. Indeed, *neat*-GP not only outperforms standard GP, but also achieves better performance and bloat control results than the original Flat-OE method.

An important result is the effect of crossover on *neat*-GP performance in each of the two problem domains considered in this work. For symbolic regression, it is clear that while all *neat*-GP variants can eliminate bloat, only when standard subtree crossover is used, performance does not degrade relative to standard GP. Moreover, *neat*-GP with subtree crossover improves performance on some problems. On the other hand, for classification the full *neat*-GP method, using the specialized NEAT-based crossover, achieves the best performance, considering both test error and the size of the evolved programs. Therefore, while *neat*-GP seems to be a good alternative for bloat control, the proper search operators should be chosen based on the problem domain.

An additional advantage of *neat*-GP is that it does not generate any additional computational overhead when compared with standard GP. In particular, for classification the same results are obtained with a 20 $\times$ speedup in total run time and a 2 $\times$ speedup in symbolic regression. This result is noteworthy, given that some of the most popular state-of-the-art methods can be more computationally costly compared to a standard GP search.

Several intriguing questions should be explored as future work related to *neat*-GP. It is of interest to test if the speciation process can be carried out in other ways. For instance, considering program syntax explicitly, or maybe considering semantic space [6,33] or behavioral space [30]. Another approach is to evaluate if diversity could be encouraged by other means, by using other popular methods such as the crowding distance or possibly incorporating the more unorthodox novelty search algorithm [13,30]. Finally, given the small solution size achieved by *neat*-GP, this could lead to more efficient implementations of memetic GP algorithms that incorporate local search strategies [36] that can become unreliable or inefficient when they are applied to extremely large trees.

Acknowledgments

Funding for this work was provided by CONACYT (Mexico) Basic Science Research Project no. 178323, DGEST (Mexico) Research Projects 5414.14-P and 5621.15-P, and FP7-PEOPLE-2013-IRSES project ACOBSEC financed by the European Commission with contract no. 612689. First author was supported by CONACYT scholarship no. 372126. The third author acknowledges funding provided by an ELEVATE Fellowship, the Irish Research Council's Career Development Fellowship co-funded by Marie Curie Actions, and thanks the TAO group at INRIA Saclay & LRI - Univ. Paris-Sud and CNRS, Orsay, France for hosting him during the outgoing phase of the ELEVATE Fellowship. The fourth author also acknowledges the support provided for this work by FCT funds (Portugal) under contract UID/Multi/04046/2013 and projects PTDC/EEI-CTP/2975/2012 (MaSSGP), PTDC/DTPFTO/1747/2012 (InteleGen) and EXPL/EMS-SIS/1954/2013 (CancerSys). Special thanks are also given to Perla S. Juárez-Smith, masters student at the Instituto Tecnológico de Tijuana, for her support in implementing the Python version of the *neat*-GP algorithm. Finally, the authors would like to thank all the reviewers for their useful comments that helped us to significantly improve our work.

References

[1] L. Altenberg, The evolution of evolvability in genetic programming, in: K.E. Kinnear (Ed.), Advances in Genetic Programming, MIT Press, Cambridge, MA, USA, 1994, pp. 47–74.
[2] K. Bache, M. Lichman, UCI machine learning repository, 2013, http://archive.ics.uci.edu/ml.
[3] J. Derrac, S. García, D. Molina, F. Herrera, A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms, Swarm Evol. Comput. 1 (1) (2011) 3–18.
[4] S. Dignum, R. Poli, Operator equalisation and bloat free GP, in: Proceedings of the 11th European Conference on Genetic Programming, EuroGP'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 110–121.
[5] F.-A. Fortin, F.-M. De Rainville, M.-A. G. Gardner, M. Parizeau, C. Gagné, DEAP: evolutionary algorithms made easy, J. Mach. Learn. Res. 13 (1) (2012) 2171–2175.
[6] E. Galvan-Lopez, B. Cody-Kenny, L. Trujillo, A. Kattan, Using semantics in the selection mechanism in genetic programming: a simple method for promoting semantic diversity, 2013 IEEE Congress on Evolutionary Computation (CEC) (2013) 2972–2979.
[7] D.E. Goldberg, J. Richardson, Genetic algorithms with sharing for multimodal function optimization, in: Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application, L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1987, pp. 41–49.
[8] J. Koza, Human-competitive results produced by genetic programming, Gen. Prog. Evol. Mach. 11 (3) (2010) 251–284.
[9] J.R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, Cambridge, MA, USA, 1992.
[10] W. Langdon, R. Poli, Foundations of Genetic Programming, Springer-Verlag New York, Inc., New York, NY, USA, 2002.
[11] W.B. Langdon, R. Poli, Fitness causes bloat, in: Proceedings of the Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing, Springer-Verlag, 1997, pp. 13–22.

[12] W.B. Langdon, R. Poli, Fitness causes bloat: mutation, in: Proceedings of the First European Workshop on Genetic Programming, EuroGP '98, Springer-Verlag, London, UK, UK, 1998, pp. 37–48.
[13] J. Lehman, K.O. Stanley, Abandoning objectives: evolution through the search for novelty alone, Evol. Comput. 19 (2) (2011) 189–223.
[14] Y. Martínez, E. Naredo, L. Trujillo, E.G. López, Searching for novel regression functions, in: Proceedings of the 2013 IEEE Congress on Evolutionary Computation (CEC), IEEE Press, 2013, pp. 16–23.
[15] Y. Martínez, L. Trujillo, E. Naredo, P. Legrand, A comparison of fitness-case sampling methods for symbolic regression with genetic programming, in: EVOLVE – A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation V, Springer International Publishing, 2014, pp. 201–212.
[16] T. McConaghy, FFX: fast, scalable, deterministic symbolic regression technology, in: R. Riolo, E. Vladislavleva, J.H. Moore (Eds.), Genetic Programming Theory and Practice IX. Genetic and Evolutionary Computation, Springer, Ann Arbor, USA, 2011, pp. 235–260.
[17] 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 Fourteenth International Conference on Genetic and Evolutionary Computation Conference, GECCO '12, ACM, New York, NY, USA, 2012, pp. 791–798.
[18] M. O'Neill, L. Vanneschi, S. Gustafson, W. Banzhaf, Open issues in genetic programming, Gen. Prog. Evol. Mach. 11 (3–4) (2010) 339–363.
[19] R. Poli, M. Graff, N.F. McPhee, Free lunches for function and program induction, in: I.I. Garibay (Ed.), Proceedings of the Tenth ACM SIGEVO Workshop on Foundations of Genetic Algorithms (FOGA), ACM, 2009, pp. 183–194.
[20] R. Poli, W.B. Langdon, S. Dignum, On the limiting distribution of program sizes in tree-based genetic programming, in: Proceedings of the 10th European Conference on Genetic Programming, EuroGP'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 193–204.
[21] R. Poli, W.B. Langdon, N.F. McPhee, A Field Guide to Genetic Programming, Lulu Enterprises, UK Ltd., 2008.
[22] R. Poli, L. Vanneschi, W.B. Langdon, N.F. Mcphee, Theoretical results in genetic programming: the next ten years? Gen. Prog. Evol. Mach. 11 (3–4) (2010) 285–320.
[23] S. Silva, Reassembling operator equalisation: a secret revealed, in: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, GECCO '11, ACM, New York, NY, USA, 2011, pp. 1395–1402.
[24] S. Silva, J. Almeida, GPLAB–a genetic programming toolbox for MATLAB, in: L. Gregersen (Ed.), Proceedings of the Nordic MATLAB Conference, 2003, pp. 273–278.
[25] S. Silva, E. Costa, Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories, Gene. Prog. Evol. Mach. 10 (2) (2009) 141–179.
[26] S. Silva, S. Dignum, L. Vanneschi, Operator equalisation for bloat free genetic programming and a survey of bloat control methods, Gen. Prog. Evol. Mach. 13 (2) (2012) 197–238.
[27] S. Silva, L. Vanneschi, The importance of being flat – studying the program length distributions of operator equalisation, in: R. Riolo, E. Vladislavleva, J.H. Moore (Eds.), Genetic Programming Theory and Practice IX. Genetic and Evolutionary Computation, Springer, New York, 2011, pp. 211–233.
[28] K.O. Stanley, R. Miikkulainen, Evolving neural networks through augmenting topologies, Evol. Comput. 10 (2) (2002) 99–127.
[29] L. Trujillo, L. Munoz, E. Naredo, Y. Martinez, NEAT, there's no bloat, in: M. Nicolau (Ed.), 17th European Conference on Genetic Programming, LNCS, 8599, Springer, 2014, pp. 174–185.
[30] L. Trujillo, E. Naredo, Y. Martínez, Preliminary study of bloat in genetic programming with behavior-based search, in: M. Emmerich (Ed.), EVOLVE - A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation IV, Advances in Intelligent Systems and Computing, 227, Springer International Publishing, 2013, pp. 293–305.
[31] L. Trujillo, G. Olague, E. Lutton, F. Fernndez de Vega, L. Dozal, E. Clemente, Speciation in behavioral space for evolutionary robotics, J. Intell. Rob. Syst. 64 (3–4) (2011) 323–351.
[32] N.Q. Uy, N.X. Hoai, M. O'Neill, R.I. Mckay, E. Galván-López, Semantically-based crossover in genetic programming: application to real-valued symbolic regression, Gen. Prog. Evol. Mach. 12 (2) (2011) 91–119.
[33] L. Vanneschi, M. Castelli, S. Silva, A survey of semantic methods in genetic programming, Gen. Prog. Evol. Mach. 15 (2) (2014) 195–214.
[34] L. Vanneschi, M. Tomassini, P. Collard, M. Clergue, Fitness distance correlation in structural mutation genetic programming, in: Proceedings of the 6th European Conference on Genetic Programming, EuroGP'03, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 455–464.
[35] D.R. White, J. McDermott, M. Castelli, L. Manzoni, B.W. Goldman, G. Kronberger, W. Jaskowski, U.-M. O'Reilly, S. Luke, Better GP benchmarks: community survey results and proposals, Gen. Prog. Evol. Mach. 14 (1) (2013) 3–29.
[36] E. Z-Flores, L. Trujillo, O. Schutze, P. Legrand, Evaluating the effects of local search in genetic programming, in: A.-A. Tantar (Ed.), EVOLVE – A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation V, Advances in Intelligent Systems and Computing, 288, Springer International Publishing, 2014, pp. 213–228.