Measuring Bloat, Overfitting and Functional Complexity in Genetic Programming

Leonardo Vanneschi[1,2]
vanneschi@disco.unimib.it

Mauro Castelli[1]
castelli@disco.unimib.it

Sara Silva[2,3]
sara@kdbio.inesc-id.pt

[1]Department of Informatics, Systems and Communication (D.I.S.Co.),
University of Milano-Bicocca, Milan, Italy
[2]INESC-ID Lisboa, KDBIO group, Lisbon, Portugal
[3]CISUC, ECOS group, University of Coimbra, Portugal

ABSTRACT

Recent contributions clearly show that eliminating bloat in a genetic programming system does not necessarily eliminate overfitting and vice-versa. This fact seems to contradict a common agreement of many researchers known as the minimum description length principle, which states that the best model is the one that minimizes the amount of information needed to encode it. Another common agreement is that overfitting should be, in some sense, related to the functional complexity of the model. The goal of this paper is to define three measures to respectively quantify bloat, overfitting and functional complexity of solutions and show their suitability on a set of test problems including a simple bidimensional symbolic regression test function and two real-life multidimensional regression problems. The experimental results are encouraging and should pave the way to further investigation. Advantages and drawbacks of the proposed measures are discussed, and ways to improve them are suggested. In the future, these measures should be useful to study and better understand the relationship between bloat, overfitting and functional complexity of solutions.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming

General Terms

Algorithms, Performance

Keywords

Genetic Programming, Bloat, Overfitting, Complexity

1. INTRODUCTION

Genetic Programming (GP) [19] traditionally uses representations of variable size. Programs tend to grow larger during the evolutionary process, as a natural result of genetic operators in search of better solutions. However, it has been shown that beyond a certain program length the distribution of fitness converges to a limit [9]. When there is an excess of code growth without a corresponding improvement in fitness, we are in the presence of *bloat* (see [22] for a review of the numerous theories explaining this phenomenon that have been proposed in the literature).

Several GP techniques that do not bloat (like for instance Cartesian GP [12]) or that counteract bloat (such as Tarpeian [16], multi-objective [3] or double tournament [10] methods and Dynamic Limits [22] among many others) have been introduced so far. In this work, we are particularly interested in a recently defined bloat free GP method called Operator Equalisation (OpEq) [6, 23, 24, 25] based on the crossover bias theory [17, 4, 5, 18], a new theory explaining the emergence of bloat. OpEq allows an accurate control of the distribution of program lengths inside the population. The original idea [6] was shortly followed by a self adapting version that automatically calculates and dynamically updates the target distribution along the evolution. This variant has been called DynOpEq [23, 24, 25]. In [24] the authors pointed out a large disparity of behaviors between standard GP (StdGP) and DynOpEq in terms of bloat and overfitting.

A common agreement of many researchers is the so called minimum description length principle (see for instance [20]), which states that the best model is the one that minimizes the amount of information needed to encode it. Simpler solutions are thought to be more robust and generalize better [21], while complex solutions are more likely to incorporate specific information from the training set, thus overfitting it. However, as mentioned in [15], this argumentation should be taken with care as too much emphasis on minimizing complexity can prevent the discovery of more complex yet more accurate solutions. A superficial interpretation of the minimum description length may be that bloat and overfitting should be two related phenomena: bloated programs could in fact use too much information, thus overfitting training data. Nevertheless, the observations in [24, 25] seem to contradict this idea.

The goal of this paper is to define three measures to respectively quantify the amount of bloat, the amount of overfitting and the functional complexity of the solutions in a GP system. We introduce these measures for the first time, pointing out their main advantages and drawbacks and trying to propose ways to improve them. These measures should be used to study and better understand the relationship between bloat, overfitting and functional complexity of solutions in the future.

Given that StdGP and DynOpEq have such different behaviors in terms of bloat and overfitting, and given that, as pointed out in [24, 25], DynOpEq seems to be able to produce much simpler (although

GECCO'10, July 7–11, 2010, Portland, Oregon, USA.

not qualitatively worse) solutions than StdGP, in this paper we study the evolution of the proposed three measures along StdGP and DynOpEq runs. The test problems we use are one simple bidimensional symbolic regression test function taken from [7] and two complex multidimensional real-life applications taken from [1].

This paper is structured as follows: in Sections 2, 3 and 4 we introduce our new measures of bloat, overfitting and functional complexity, respectively. Section 5 presents the DynOpEq algorithm. In Section 6 we describe the test problems used. In Section 7 we specify the parameter settings used in our experiments and we report and discuss the obtained results. Finally, Section 8 concludes the paper and offers hints for future research.

2. MEASURE OF BLOAT

The most accepted definition of bloat is "program growth without (significant) return in terms of fitness" (definition taken from [19], page 101). This definition is clearly a very fuzzy one and in many bibliographic references GP problems are analyzed in terms of "presence" or "absence" of bloat, without any numerical quantification. This is the case, among many other references, of [24] where the authors state "StdGP bloats" and "DynOpEq does not bloat", or of [19] where the authors use informal expressions like "bloat is more marked" (page 41), "reduce bloat" (page 78) or "bloat can ... happen" (page 106). We believe that these expressions are really too informal to permit a deep study of bloat and we introduce a simple measure to quantify the "amount of bloat of a GP run" with a single number. To the best of our knowledge, such a measure has never been defined so far.

According to the proposed measure, in case of minimization problems (i.e. problems were a small fitness value is better than a large one) the "amount of bloat" at generation g is defined as:

$$bloat(g) = \frac{(\overline{\delta}(g) - \overline{\delta}(0))/\overline{\delta}(0)}{(\overline{f}(0) - \overline{f}(g))/\overline{f}(0)} \quad (1)$$

where $\overline{\delta}(g)$ is the average program length[1] in the population at generation g and $\overline{f}(g)$ is the average fitness (calculated using training data) in the population at generation g.

We can informally say that $bloat(g)$ is an expression of the relationship between the average length growth and the average fitness improvement up to generation g compared to the respective values at generation zero. If we assume that at generation zero we have no bloat[2], then this is a very intuitive formalization of the definition of bloat. Furthermore, if we assume that, for a given $g > 0$, $\overline{\delta}(g)$ is larger than $\overline{\delta}(0)$ and (for minimization problems) $\overline{f}(g)$ is smaller than $\overline{f}(0)$, which is what normally happens in an usual GP run, then we have that $bloat(g)$ often assumes positive values[3]. Finally, if g is large, then we usually have that small improvements in fitness correspond to large improvements in length. For this reason, we have normalized the numerator and denominator of $bloat(g)$ by $\overline{\delta}(0)$ and $\overline{f}(0)$. This normalization mitigates the effect of the two different orders of magnitude of these improvements.

[1] Following [8], with the expression *length of a program* we mean the total number of nodes in its tree-like representation.

[2] If we assume that the definition of bloat is "program *growth* without (significant) return in terms of fitness", then it comes natural to assert that at generation zero programs did not *grow* yet, and thus there can be no bloat. Furthermore, it is worth pointing out that at generation zero, i.e. after the initialization phase, usually programs have a strong size limitation given by the initialization method used. In this work we use Ramped Half and Half initialization [8] with a depth limit equal to 6.

[3] This has a known exception for symbolic regression problems, where the average population size usually shrinks in the very first generations, before increasing progressively in the rest of the run. For this reason, we did not force this constraint to necessarily hold, thus assuming that $bloat(g)$ can also have negative values.

3. MEASURE OF OVERFITTING

Following [13] (page 67), overfitting can be defined as follows: "Given a hypothesis space H, a hypothesis $h \in H$ is said to *overfit* the training data if there exists some alternative hypothesis $h' \in H$, such that h has smaller error than h' over the training samples, but h' has smaller error than h over the entire distribution of instances". Under this perspective, defining a measure for overfitting is clearly a hard task, given that it requires the exploration of all the hypothesis space H, looking for such a hypothesis h'. Nevertheless, we believe that a good indication of the "amount of overfitting" of a GP system can be given by the relationship between fitness on the training set (training fitness from now on) and the fitness on the test set (test fitness from now on). Thus, at a given generation `g`, we quantify overfitting by the `overfit(g)` value calculated by the pseudocode in Figure 1, where: `btp` stands for "best test point" and it represents the best test fitness reached until the current generation, excluding the generations (usually at the beginning of the run) where the best individual on the training set has a better test than training fitness; `tbtp` stands for "training at best test point", i.e. the training fitness of the individual that has test fitness equal to `btp`; `training_fit(g)` is a function that returns the best training fitness in the population at generation `g`; `test_fit(g)` returns the test fitness of the best individual on the training set at generation `g`.

```
overfit(0) = 0
btp = test_fit(0)
tbtp = training_fit(0)

for each generation g>0
  if (training_fit(g) > test_fit(g))
     overfit(g) = 0
  else
     if (test_fit(g) < btp)
       overfit(g) = 0
       btp = test_fit(g)
       tbtp = training_fit(g)
     else
       overfit(g) =
         |training_fit(g)-test_fit(g)| - |tbtp-btp|
     endif
  endif
endfor
```

Figure 1: Pseudocode for calculating the overfitting measure presented in Section 3, in case of minimization problems.

In synthesis, the ideas that inspire this algorithm are: if, at a given generation `g`, test fitness is better than training fitness, then there is no overfitting (`overfit(g)=0`); if test fitness is better than the best test point, then there is no overfitting (`overfit(g)=0`); otherwise overfitting is quantified by the difference of the distance between training and test fitness at generation `g` and the distance between training and test fitness at the generation where the best test point has been found. Given that we use elitism in our GP runs, training fitness is constantly improving, or in the worst case it stays constant. For this reason: $|\texttt{training\_fit(g)-test\_fit(g)}| \geq |\texttt{tbtp-btp}|$. Thus we

have that, for all generations g, `overfit(g)` ≥ 0, but this may not be the case in the absence of elitism.

We are aware that this definition has the clear and strong limitation that it dramatically depends on how training and test data are chosen and we believe that a more sophisticated definition, where training and test data are alternated in a crossvalidation-like way is suitable. Nevertheless, this paper represents a first and preliminary study of this measure and thus we prefer to keep things simple and we adopt the algorithm in Figure 1.

We also point out that using the same idea as in equation (1) for measuring the relationship between training and test fitness (i.e. comparing the values at the current generation with the corresponding ones at generation zero) is definitely unsuitable. In fact, at generation zero the population has been created by the (typically random) initialization algorithm and the solutions in the population have not undergone any evolution yet. For this reason, there is typically no reason why training fitness should be better than test fitness at generation zero. On the other hand, these two values are typically quite similar and being the test set often smaller than the training set, it is even more likely that the average training fitness is worse than the average test fitness at generation zero. For this reason, using a formula like equation (1) to quantify overfitting would produce a measure that is more affected by the values of training and test fitness at generation zero than by the course of the learning process. As a consequence of the fact that training and test fitness values are very similar to each other at generation zero (or test fitness is even better than training fitness), a measure like equation (1) would very often produce very high overfitting values, even when it is clearly not the case.

4. MEASURE OF COMPLEXITY

Previous and Related Work. The relationship between code growth and functional complexity in GP has been only lightly investigated so far. In [26], the authors define two measures of complexity for the GP individuals: a genotypic measure and a phenotypic one. While the genotypic measure is related to counting the number of nodes of a tree and its subtrees, the phenotypic one, called *order of nonlinearity*, is related to functional complexity and consists in calculating the degree of the Chebyshev polynomial approximation of the function. The authors then use these two measures as further criteria (besides fitness) in a multi-objective system based on Pareto optimization, and they show that this system is able to counteract bloat and overfitting.

In this paper, complexity is used differently: the proposed measure will not be used as an optimization criterium, but just reported for the current best individual (in the training set) along the evolution, in order to discover its relationship with bloat and/or overfitting. Furthermore, we do not wish to find the approximating polynomial of the function, but rather wish to work directly on the function itself. For this reason, we propose a measure inspired by the concept of curvature as an indicator of the functional complexity.

Curvature. Informally, the curvature [2] of a function is the amount by which its geometric representation deviates from being *flat*, or *straight*. For a plain curve in 3 dimensions, given parametrically as $c(t) = (x(t), y(t))$, the curvature is defined as [2]:

$$k = \frac{x'y'' - y'x''}{(x'^2 + y'^2)^{3/2}}$$

Suppose we want to solve a tridimensional symbolic regression problem using GP, and suppose we want to report the average curvature of the individuals in the population at each generation. This task is computationally very hard. In fact it implies the calculation of simple and second derivatives of the function along two dimensions. In case the coding function of a GP individual is not derivable, the exact calculus may even be impossible, and an approximation would require the previously mentioned [26] calculation of the approximating polynomial of that function. Furthermore, the dimension of the feature space of usual symbolic regression problems is by far larger than three, as it is the case of the real-life applications used as test problems in this paper and described in Section 6. The theory of generalized curvature measures to multidimensional spaces has a long history, that has recently been gathered in [14]. Nevertheless, given its sophistication, this subject is outside the scope of this paper.

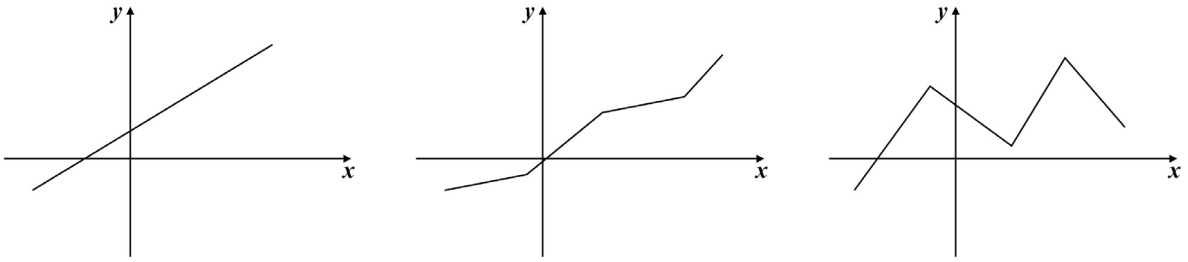
On the other hand, in this paper we define an indicator of functional complexity that is inspired by our intuition of the concept of curvature and that has the advantage of being calculable in polynomial time with the number of fitness cases. The intuition of this indicator is very simple, even though its formal definition may seem a bit complicated. For this reason, we first present this measure from an intuitive viewpoint, then we give a simple example of its calculation and finally we give its formal definition.

Intuition. Consider the three simple bidimensional functions in Figure 2. We want to define a measure to quantify the fact that the function represented in Figure 2(c) is more complex than the one represented in Figure 2(b), which is more complex than the one represented in Figure 2(a). In fact, all the three functions are polylines, but in Figure 2(a) all the segments forming the polyline have the same slope, while in Figure 2(b) the segments have different slopes, even though all the slopes have the same sign; finally, segments in Figure 2(c) have different slopes of different signs. In other words, we wish to express the complexity of a function by counting the number of different slopes and by assigning a higher weight to inversions in the slope sign. It is worth pointing out that such a measure has formally absolutely no relationship with the curvature measure. It is only inspired by our informal intuition of the concept of curvature.

Considering bidimensional regression problems, one can imagine the graphical representation of a GP individual as a polyline, by plotting the points that have fitness cases on the abscissas and the corresponding values of the function coding the GP individual as ordinates and joining those points by segments. Thus, in case of bidimensional problems, all we have to do to calculate our measure is to sort all fitness cases (from the smaller value to the larger one) and to consider the values assumed by the GP individual on those fitness cases. Then we calculate the slope of each segment joining these points. Let $(s_1, s_2, s_3, s_4, s_5, ...)$ be those slopes. The measure is simply calculated as: $|s_1 - s_2| + |s_2 - s_3| + |s_3 - s_4| + |s_4 - s_5| + ...$ In this way, if the slopes are all identical, the value of the measure is zero (minimal possible complexity) and if the signs of the slopes of all the consecutive segments change, the contribution of each segment is maximal (because all slopes' absolute values are summed). On the other hand, if two consecutive segments have different slopes with the same sign, their contribution is the subtraction of their respective absolute values (i.e. a contribution larger than zero, but smaller than in the case where the slope sign changes).

In case of multidimensional spaces of features, we simply consider the projection of the function coding the GP individual on each single dimension. We execute the same calculus as above for each dimension separately and then we calculate the average.

Example. Let the fitness cases of a bidimensional regression problem be $(0, \pi/2, 3/2\pi, 2\pi)$ and let $g(x) = sin(x)$ be a GP individual. Then the points of the polyline are:



(a) (b) (c)

Figure 2: The graphical representation of three simple functions used to explain our complexity measure from an intuitive viewpoint. According to our measure, the function represented in plot (a) must be less complex than the function represented in plot (b) and the function represented in plot (b) must be less complex than the function represented in plot (c) (see the text for further explanation).

$\{(0,0),(\pi/2,1),(3/2\pi,-1),(2\pi,0)\}$. In this case, we must calculate the slope of the segment that joins the points $(0,0)$ and $(\pi/2,1)$, the one of the segment that joins the points $(\pi/2,1)$ and $(3/2\pi,-1)$ and the one of the segment that joins the points $(3/2\pi,-1)$ and $(2\pi,0)$. Let s_1, s_2 and s_3 be these three slopes respectively. We have: $s_1 = 1/(\pi/2) = 2/\pi$, $s_2 = -2/(3/2\pi - \pi/2) = -2/\pi$ and $s_3 = 1/(2\pi - 3/2\pi) = 2/\pi$. The value of our measure is: $|s_1 - s_2| + |s_2 - s_3| = 8/\pi$.

Formal Definition. Let $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n)$ be the vector of fitness cases of a regression problem, where for each $i = 1,2,...,n$: $\mathbf{x}_i = (x_{i1}, x_{i2}, ..., x_{im})$ is an m-dimensional vector of floating point numbers (i.e. the feature space is m-dimensional). Let $g : \mathbb{R}^m \rightarrow \mathbb{R}$ be the function coding a GP individual and let $g_i = g(\mathbf{x}_i)$ be the value assumed by g on the i^{th} fitness case. Given a $j = 1,2,...,m$, let $\mathbf{p}_j = (x_{1j}, x_{2j}, ..., x_{nj})$ be the vector containing all the values of feature j in $\mathbf{X}$. Now let $\mathbf{q}_j = (y_{1j}, y_{2j}, ..., y_{nj})$ be a vector that contains the same values as $\mathbf{p}_j$, except that they are sorted (i.e. $\mathbf{q}_j$ is a permutation of $\mathbf{p}_j$, such that the values in $\mathbf{q}_j$ are sorted). Let $\phi : \{1,2,...,n\} \rightarrow \{1,2,...,n\}$ be a function that, applied to an index in $\mathbf{q}_j$ returns the position of the corresponding element in $\mathbf{p}_j$. In other words, for all $k,h = 1,2,...,n$: $y_{kj} = x_{hj}$ if and only if $\phi(k) = h$. Then we define:

$$pc_j = \begin{cases} \sum_{i=1}^{n-2} \left| \frac{g_{\phi(i+1)} - g_{\phi(i)}}{y_{(i+1)j} - y_{ij}} - \frac{g_{\phi(i+2)} - g_{\phi(i+1)}}{y_{(i+2)j} - y_{(i+1)j}} \right|, & \text{if } n \geq 3 \\ 0, & \text{otherwise} \end{cases}$$

where pc_j stands for *partial complexity* on the j^{th} dimension. Finally, we define our complexity measure as the average of all the partial complexities on all the dimensions of the feature space, i.e.:

$$complexity = \left(\sum_{j=1}^{m} pc_j \right) / m$$

5. DYNOPEQ

Developed alongside the crossover bias theory [17, 4, 5, 18], Operator Equalisation (OpEq) is a recent technique to control bloat that allows an accurate control of the program length distribution inside a population during a GP run. To better explain how it works, we use the concept of a histogram. Each bar of the histogram can be imagined as a bin containing those programs whose length is within a certain interval. The width of the bar determines the range of lengths that fall into this bin, and the height specifies the number of programs allowed within. We call the former *bin width* and the latter *bin capacity*. All bins are the same width, placed adjacently with no overlapping. Each length value, l, belongs to one and only one bin b, identified as follows:

$$b = \left\lfloor \frac{l-1}{bin\_width} \right\rfloor + 1 \quad (2)$$

For instance, if $bin\_width = 5$, bin 1 will hold programs of lengths 1,..,5, bin 2 will hold programs of lengths 6,..,10, etc. The set of bins represents the distribution of program lengths in the population. OpEq biases the population towards a desired target distribution by accepting or rejecting each newly created individual into the population (and into its corresponding bin). The original implementation of OpEq [6], where the user was required to specify the target distribution and maximum program length, rapidly evolved to a self adapting implementation called DynOpEq [23, 24, 25], where both these elements are automatically set and dynamically updated to provide the best setting for each stage of the evolutionary process. There are two tasks involved in DynOpEq: calculating the target (in practical terms, defining the capacity of each bin) and making the population follow it (making sure the individuals in the population fit the set of bins).

Calculating the Target Distribution. In DynOpEq the dynamic target length distribution simply follows fitness. For each bin, the average fitness of the individuals within is calculated, and the target is directly proportional to these values. Bins with higher average fitness will have higher capacity, because that is where search is proving to be more successful. Formalizing, the capacity, or target number of individuals, for each bin b, is calculated as:

$$bin\_capacity_b = round\left(n \times (\bar{f}_b / \sum_i \bar{f}_i) \right)$$

where $\bar{f}_i$ is the average fitness in the bin with index i, $\bar{f}_b$ is the average fitness of the individuals in b, and n is the number of individuals in the population.

Initially based on the first randomly created population, the target is updated at each generation, always based on the fitness measurements of the current population. This creates a fast moving bias towards the areas of the search space where the fittest programs are, avoiding the small unfit individuals resulting from the crossover bias, as well as the excessively large individuals that do not provide better fitness than the smaller ones already found.

Following the Target Distribution. In DynOpEq every newly created individual must be validated before eventually entering the population. In particular, DynOpEq rejects the individuals that do not fit the target: individuals from the population are selected for mating and the application of genetic operators allow us to create

new individuals, as in standard GP. After that, the length of each new individual is measured, and its corresponding bin is identified using Equation (2). If this bin already exists and is not full (meaning that its capacity is higher than the number of individuals already there), the new individual is immediately accepted. If the bin still does not exist (meaning it lies outside the current target boundaries) the fitness of the individual is measured and, in case we are in the presence of the new best-of-run (the individual with best training fitness found so far), the bin is created to accept the new individual, becoming immediately full. Any other non existing bins between the new bin and the target boundaries also become available with capacity for only one individual each. The criterion of creating new bins whenever needed to accommodate the new best-of-run individual is inspired by the Dynamic Limits [22] bloat control technique.

When the intended bin exists but is already at its full capacity, or when the intended bin does not exist and the new individual is not the best-of-run, DynOpEq evaluates the individual and, if we are in the presence of the new best-of-bin (meaning the individual has better fitness than any other already in that bin), the bin is forced to increase its capacity and accept the individual. Otherwise, the individual is rejected. Permitting the addition of individuals beyond the bin capacity allows a clever overriding of the target distribution, by further biasing the population towards the lengths where the search is having a higher degree of success. In the second case, when the bin does not exist and the individual is not the best-of-run, rejection always occurs[4].

6. TEST PROBLEMS

The first test problem we use in this paper is taken from [7]. Maintaining the same terminology as in [7], we have used test function $F_7(x) = log(x)$. As in [7] we have used a training set equal to $x = [1 : 1 : 100]$ and a test set equal to $x = [1 : 0.1 : 100]$.

We are aware that this test function is bidimensional (as almost all the ones considered in [7]) and thus it does not represent real-life applications (typically characterized by many features and thus multidimensional). For this reason, we also consider two real-life applications characterized by a large dimensionality of the feature space. These two applications consist in predicting the value of two pharmacokinetic parameters of a set of candidate drug compounds on the basis of their molecular structure. The first pharmacokinetic parameter we consider is human oral bioavailability (indicated with %F from now on) and the second one is median lethal dose (indicated with LD50 from now on), also informally called toxicity.

%F is the parameter that measures the percentage of the initial orally submitted drug dose that effectively reaches the systemic blood circulation after the passage from the liver. LD50 refers to the amount of compound required to kill 50% of the test organisms (cavies). For a more detailed discussion of these two pharmacokinetic parameter the reader is referred to [1].

The datasets we have used are the same as in [1]: the %F dataset consists in a matrix composed by 260 rows (instances) and 242 columns (features). Each row is a vector of molecular descriptor values identifying a drug; each column represents a molecular descriptor, except the last one, that contains the known target values of %F. The LD50 dataset consists in a matrix composed by 234 rows (instances) and 627 columns (features). Also in this case, each row is a vector of molecular descriptors identifying a drug and each column represents a molecular descriptor except the last one, that contains the known values of LD50. These datasets can be downloaded from *http://personal.disco.unimib.it/Vanneschi/bioavailability.txt* and *http://personal.disco.unimib.it/Vanneschi/toxicity.txt*, respectively.

For both these datasets training and test sets have been obtained by random splitting: at each different GP run, 70% of the molecules have been randomly selected with uniform probability and inserted into the training set, while the remaining 30% formed the test set.

7. EXPERIMENTS

Experimental Setting. A total of 30 runs were performed both with StdGP and DynOpEq. All the runs used populations of 500 individuals allowed to evolve for 100 generations. Tree initialization was performed with the Ramped Half-and-Half method [8] with a maximum initial depth of 6. The function set contained the four binary operators $+$, $-$, $\times$, and $/$, protected as in [8]. The terminal set contained one floating point variable for the F_7 test function, 241 floating point variables for the %F dataset and 626 floating point variables for the LD50 dataset. In all these test problems no random constants were added to the terminal set. Because the cardinalities of the function and terminal sets were so different, we have imposed a balanced choice between functions and terminals when selecting a random node. Fitness was calculated as the root mean squared error between outputs and targets. Selection for reproduction used Lexicographic Parsimony Pressure [11] tournaments of size 10. Very similar to a regular tournament, the lexicographic tournament chooses smaller individuals when their fitness is the same. The reproduction (replication) rate was 0.1, meaning that each selected parent has a 10% chance of being copied to the next generation instead of being engaged in breeding. Standard tree mutation and standard crossover (with uniform selection of crossover and mutation points) were used with probabilities of 0.1 and 0.9, respectively. The new random branch created for mutation has maximum depth equal to 6. Selection for survival used elitism (i.e. unchanged copy of the best individual in the next population). Regarding the parameters specific to each technique, StdGP used a fixed maximum depth of 17, and DynOpEq used a bin width of 1.

Experimental Results. Experimental results are shown in Figures 3, 4 and 5 for the F_7 test function, and the %F and LD50 test problems, respectively. In all these figures plots (a) report the best training fitness and the test fitness of the best individual on the training set vs. generations. These plots should give an intuition of the generalization ability of the solutions found by the different GP models and their amount of overfitting. In plots (b) we report some unconventional curves: best training fitness vs. average program length. Depending on how fast the fitness improves with the increase of program length, the lines in the plot may point downward (south), or they may point to the right (east). Lines pointing south represent a rapidly improving fitness with little or no code growth. Lines pointing east represent a slowly improving fitness with strong code growth. These plots should help us to make inferences about bloat: for a method that does not bloat, but at the same time is able to improve fitness, these lines should point as south as possible. Plots (c) report the bloat measure described in Section 2 vs. generations. Plots (d) report the overfitting measure described in Section 3 vs. generations[5]. Finally plots (e) report the complexity measure described in Section 4 vs. generations. In all cases, median values over the 30 runs are reported. When compar-

[4]We point out that there is an obvious computational overhead in evaluating so many individuals that end up being rejected. This subject has been extensively addressed in previous work [23] where the main conclusion was that most rejections happen in the beginning of the run and refer to very small individuals.

[5]For an easier comparison of overfitting between the three problems, each y axis was drawn so that its maximum value is roughly one fifth of the maximum y axis value of the respective plot (a).

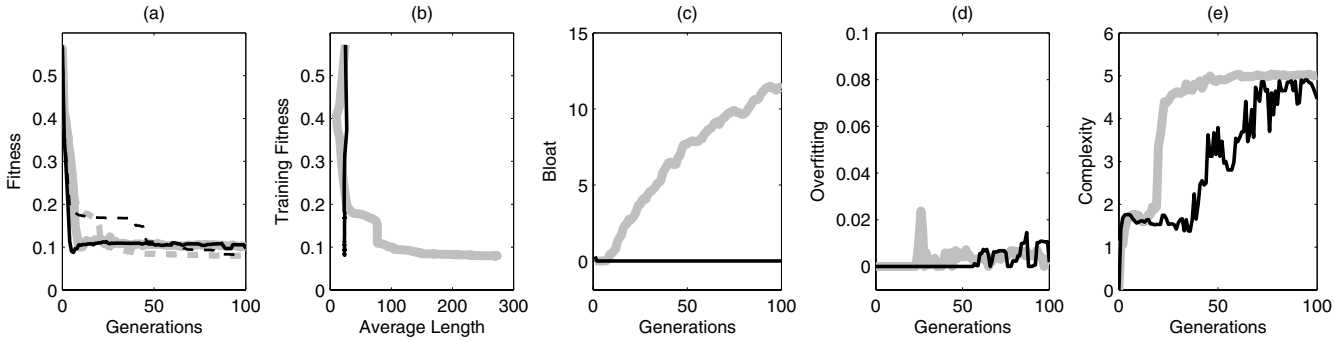


Figure 3: Test Function: $F_7(x) = log(x)$**. Plot (a): best training and test fitness vs. generations. Plot (b): best training fitness vs. average program length. Plot (c): bloat measure described in Section 2 vs. generations. Plot (d): overfitting measure described in Section 3 vs. generations. Plot (e): complexity measure described in Section 4 vs. generations. In all cases, median values over 30 independent runs are reported. Legend: grey thick lines = StdGP; black thin lines = DynOpEq. (In Plot (a): solid lines = test fitness; dashed lines = training fitness).**

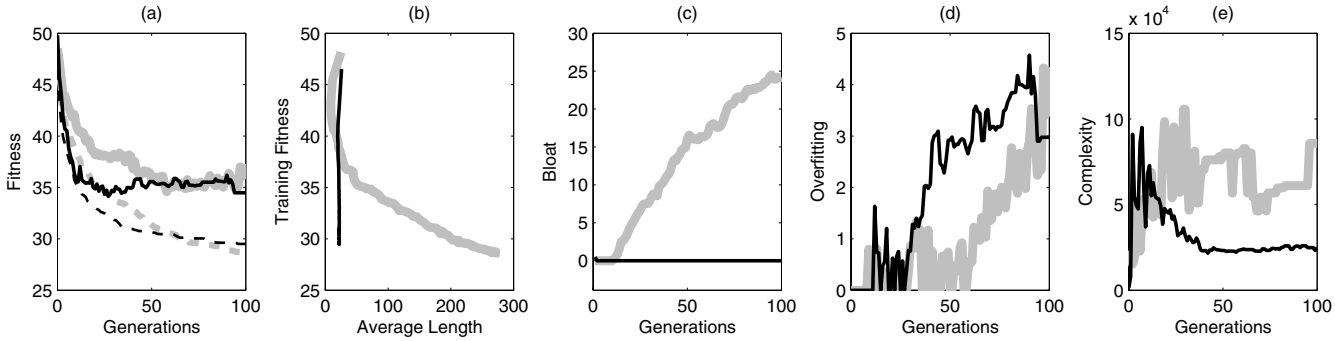


Figure 4: Test Function: *bioavailability (%F) regression***. All the rest as in Figure 3.**

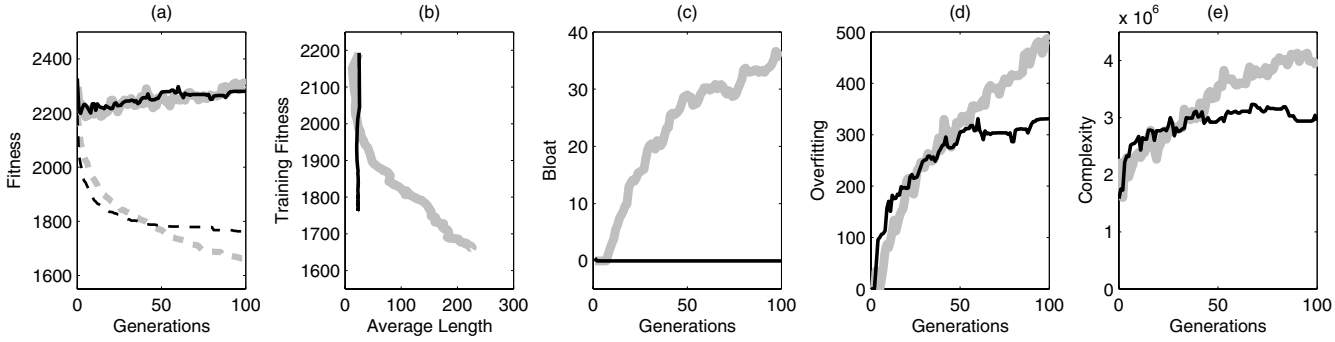


Figure 5: Test Function: *toxicity (LD50) regression***. All the rest as in Figure 3.**

ing curves, we use the term "correlation" only from a visual, not statistical, point of view.

Let us begin by analyzing the results of the F_7 test function (Figure 3). Plot 3(a) shows that DynOpEq does not overfit in the first part of the run, because test fitness is better than training fitness, while it slightly overfits in the second part, where training fitness improves and test fitness remains approximately constant. StdGP shows a similar behavior, with a slightly more marked overfitting in the first part of the run. It is interesting to point out that the overfitting measure (plot 3(d)) captures this fact, given that for both methods it stays constantly equal to zero, except for some small oscillations in the second part of the run and for some larger oscillations for StdGP in the first part. If we observe plot 3(e), we can also see that the functional complexity of the individuals found by StdGP and DynOpEq is more or less the same at the end of the run, but StdGP increases its values sooner and faster than DynOpEQ. There is also a clear correlation between complexity and the best training fitness (plot 3(a)). Plot 3(b) clearly shows that StdGP bloats, while DynOpEq does not bloat. In fact, as training fitness keeps improving, the average length in the StdGP population increases, while it remains approximately constant in the DynOpEq population. Consistently, after a small variation in the first five generations, the bloat measure (plot 3(c)) is constantly equal to zero for DynOpEq, while it steadily grows for StdGP.

Now let us focus on the results obtained on the %F dataset (Figure 4). Plot 4(a) shows that DynOpEq starts overfitting earlier than StdGP. Now observe the behavior of the overfitting measure (plot 4(d)). The measure has larger values for DynOpEq from the beginning of the run, but they are met by the StdGP values towards the end of the run. This is partially captured by the complexity measure (plot 4(e)), where the values for DynOpEq increase sooner than the values for StdGP. Despite the overfitting, the complexity values soon decline and remain surprisingly low for DynOpEq. Plot 4(b) clearly shows that StdGP bloats while DynOpEq does not bloat. Consistently, the bloat measure remains constantly equal to zero (except for a small oscillation in the first three generations) for DynOpEq, while it steadily keeps growing during the whole evolution for StdGP.

Finally, let us focus on the results obtained on the LD50 dataset (Figure 5). Plot 5(a) shows that both methods overfit, but StdGP overfits more than DynOpEq, in particular in the second part of the run. In fact, while training fitness keeps improving for both methods during the whole evolution, test fitness has a minimum and then, for both methods, it keeps deteriorating. This minimum is around generation 10 for both methods, but this deterioration is clearly stronger for StdGP. In fact the fitness on the test set keeps deteriorating until the end of the evolution for StdGP, while it remains approximately constant after generation 60 for DynOpEq. The behavior of the overfitting measure (plot 5(d)) is consistent with this. It keeps growing during the whole evolution for StdGP, while for DynOpEq it grows in the first phase of the evolution and then stabilizes on a more or less constant value, lower than the values of StdGP. Also the complexity measure behaves consistently (plot 5(e)). In fact, at the end of the evolution DynOpEq starts producing less complex individuals than StdGP. Plot 5(b) clearly shows that also for this test problem StdGP bloats while DynOpEq does not bloat. Consistently, the bloat measure (plot 5(c)) is always equal to zero for DynOpEq, while it steadily keeps growing during the whole evolution for StdGP.

Discussion. Even though promising, the results that we have obtained do not have to be emphasized. In fact, we are perfectly aware that each proposed measure has drawbacks that have to be overcome if we really want to use them to make strong inferences about bloat, overfitting, functional complexity and their mutual relationship. For instance:

(1) The bloat measure compares the fitness and the length of programs at a given generation with the fitness and the length of programs at generation zero. This is a good starting point, given that we can assume that we have no bloat at generation zero (as described above). Nevertheless, the initialization algorithm (that defines the population at generation zero) has biases both on program length and on fitness. These biases clearly affect the measure and should be taken into account. Or otherwise, a measure that does not take generation zero as a reference deserves to be defined and investigated.

(2) As already discussed above, the overfitting measure clearly depends on how training and test sets have been chosen. Given that at each GP run we have used a different (random) partitioning of the dataset into training and test sets, it may happen that exactly the same population has two different values of the overfitting measure in two different runs. As a good starting point, we have decided to perform 30 independent runs of each experiment, and we have reported median values, hoping that this mitigates the bias of the measure given by its dependence on the training/test partitioning. Furthermore, we also have to point out that having exactly the same population in two different runs is a very unlikely event. Nevertheless the problem exists and it is important. New versions of this measure deserve to be defined, for instance alternating training and test data in a crossvalidation-like way.

Last but not least (3), the functional complexity measure has formally *no relationship* with curvature. It is an empirical indicator, that captures our idea of curvature only from an intuitive viewpoint. It is a good starting point, because compared to other (more formal) measures it has the advantage of being simple and computationally cheap. Furthermore, the number of slope changes and (even more) the number of slope sign changes are clearly related to the empirical intuition of the concept of functional complexity. Nevertheless, we suspect that the lack of formality of this measure may cause problems in the future and we believe that more formal measures, hopefully still computationally cheap, deserve investigation.

8. CONCLUSIONS AND FUTURE WORK

The goal of this paper was to define measures for bloat, overfitting and functional complexity in GP evolution. These measures, or their future improvements, should be helpful to investigate and better understand the relationship between these three phenomena in the future. Even though we recognize that, having been studied for the first time in this contribution, these measures suffer of several drawbacks and limitations, the results we have obtained are encouraging and should pave the way to further investigation. In particular, we have shown that, on a simple symbolic regression bidimensional test function and on two real-life multidimensional regression problems, the proposed quantification of bloat and overfitting nicely represent the intuitive perception that GP users have of these two phenomena. The complexity measure has shown a curious and still unexplained correlation with training fitness and overfitting.

Future activity includes new definitions of these measures aimed at overcoming their most severe drawbacks discussed in the previous section. We also intend to statistically formalize the correlation between the three measures, and possibly identify other elements of the evolutionary process that may help us explain the relationship between them. Since the only two GP techniques used in this paper represent what may currently be the worst (StdGP) and the best (DynOpEq) in terms of bloat control, we plan to validate the measures with other techniques that neither bloat as much as StdGP

neither completely eliminate it as DynOpEq. Finally, we plan to use these measures in different domains beyond regression problems (e.g. in boolean problems or path recognition problems like the artificial ant) and to test them on other complex applications.

Acknowledgments. The authors acknowledge project PTDC/EIA-CCO/103363/2008 from Fundação para a Ciência e a Tecnologia, Portugal.

9. REFERENCES

[1] F. Archetti, S. Lanzeni, E. Messina, and L. Vanneschi. Genetic programming for human oral bioavailability of drugs. In M. Keijzer, et al., editors, *GECCO 2006: Proc. of the 8th annual conference on Genetic and evolutionary computation*, volume 1, pages 255–262. ACM Press, 2006.

[2] J. Casey. *Exploiting Curvature*. Wiesbaden, Germany: Vieweg, 1996.

[3] E. D. de Jong, R. A. Watson, and J. B. Pollack. Reducing bloat and promoting diversity using multi-objective methods. In L. Spector, et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 11–18, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[4] S. Dignum and R. Poli. Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat. In D. Thierens, et al., editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1588–1595, London, 7-11 July 2007. ACM Press.

[5] S. Dignum and R. Poli. Crossover, sampling, bloat and the harmful effects of size limits. In M. O'Neill, et al., editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 158–169, Naples, 26-28 Mar. 2008. Springer.

[6] S. Dignum and R. Poli. Operator equalisation and bloat free GP. In M. O'Neill, et al., editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 110–121, Naples, 26-28 Mar. 2008. Springer.

[7] M. Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In C. Ryan, et al., editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 70–82, Essex, 14-16 Apr. 2003. Springer-Verlag.

[8] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[9] W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer, Berlin, Heidelberg, New York, Berlin, 2002.

[10] S. Luke and L. Panait. Fighting bloat with nonparametric parsimony pressure. In J. J. Merelo-Guervos, et al., editors, *Parallel Problem Solving from Nature - PPSN VII*, number 2439 in Lecture Notes in Computer Science, LNCS, pages 411–421, Granada, Spain, 7-11 Sept. 2002. Springer-Verlag.

[11] S. Luke and L. Panait. Lexicographic parsimony pressure. In W. B. Langdon, et al., editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 829–836, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[12] J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, et al., editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.

[13] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[14] J.-M. Morvan. *Generalized Curvatures*. Springer. Series: Geometry and Computing, Vol. 2, 2008.

[15] P.Domingos. The role of Occam's razor in knowledge discovery. *Data Mining and Knowledge Discovery*, 3(4):409–425, 1999.

[16] R. Poli. A simple but theoretically-motivated method to control bloat in genetic programming. In C. Ryan, et al., editors, *Genetic Programming, Proceedings of the 6th European Conference, EuroGP 2003*, volume 2610 of *LNCS*, pages 200–210, Essex, 14-16 Apr. 2003. Springer, Berlin, Heidelberg, New York.

[17] R. Poli, W. B. Langdon, and S. Dignum. On the limiting distribution of program sizes in tree-based genetic programming. In M. Ebner, et al., editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 193–204, Valencia, Spain, 11 - 13 Apr. 2007. Springer.

[18] R. Poli, N. F. McPhee, and L. Vanneschi. The impact of population size on code growth in GP: analysis and empirical validation. In M. Keijzer, et al., editors, *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1275–1282, Atlanta, GA, USA, 12-16 July 2008. ACM.

[19] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[20] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[21] J. Rosca. Generality versus size in genetic programming. In J. R. Koza, et al., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 381–387, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[22] S. Silva and E. Costa. Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. *Genetic Programming and Evolvable Machines*, 10(2):141–179, 2009.

[23] S. Silva and S. Dignum. Extending operator equalisation: Fitness based self adaptive length distribution for bloat free GP. In L. Vanneschi, et al., editors, *Proceedings of the 12th European Conference on Genetic Programming, EuroGP2009*, pages 159–170. Springer, 2009.

[24] S. Silva and L. Vanneschi. Operator equalisation, bloat and overfitting: a study on human oral bioavailability prediction. In G. Raidl, et al., editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1115–1122, Montreal, 8-12 July 2009. ACM.

[25] L. Vanneschi and S. Silva. Using operator equalisation for prediction of drug toxicity with genetic programming. In L. S. Lopes, et al., editors, *EPIA*, volume 5816 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2009.

[26] E. J. Vladislavleva, G. F. Smits, and D. den Hertog. Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. *IEEE Transactions on Evolutionary Computation*, 13(2):333–349, 2009.