Relaxed Genetic Programming

[Submitted to track: Genetic Programming]

ABSTRACT

A study on the performance of solutions generated by **G**enetic **P**rogramming (**GP**) when the training set is *relaxed* (in order to allow for a wider definition of the desired solution) is presented. This performance is assessed through 2 important features of a solution: its *generalization error* and its *bloat*, a common problem of **GP** individuals. Some encouraging results are presented: we show how even a small degree of relaxation improves the generalization error of the best solutions; we also show how the variation of this parameter reduces the bloat of the solutions generated.

General Terms

Genetic Programming

Keywords

Genetic Programming, Bloat, Generalization Error

1. INTRODUCTION

Darwinian evolution and Mendelian genetics are the inspiration for the field of **G**enetic **P**rogramming (**GP**); **GP** solves complex problems by evolving populations of computer programs, being particularly relevant to approach non-analytical, multiobjectives and (practically) infinite solution space dimension problems[1]. **GP** has proved its utility for supporting human analysis of problems in a variety of domains: computational molecular biology [4], cellular automata, sorting networks, analogical electrical circuits [2], among them.

In this paper, we present a study of 2 aspects of the **GP** method: the *generalization error* and the problem of *bloat*.

The **generalization error** of a machine learning model is a function that estimates how far a candidate solution is from a target solution, over the *"entire"* set of possible data that could be generated by this target solution. It indicates the capacity of the candidate to generalize, based only on a few examples.

Bloat is generally defined as *an excess of code growth caused by the genetic operators in search of better solutions, without a corresponding improvement in fitness*. It is a serious problem in **GP**, often leading to the stagnation of the evolutionary process [1]. Several approaches have been proposed for dealing with this problem; a brief overview is presented in the next section.

1.1 Popular approaches to bloat control: tree depth/resource limits

The traditional approach to maintaining code growth under control is to impose a tree depth limit on the individuals accepted into the population [3]. This approach raises several practical questions on the practical use of **GP** as a general method: what is the value of that limit? Is it problem-dependent? (if so, can we devise a method to automatically *tune* the algorithm?) How do we ensure that the limit is high enough, in order to be sure to keep the best solutions in our pool? These and other questions have been widely studied by researchers in the field, and several other techniques have been used with various degrees of success (see, for example, refs. [10, 7, 5, 6] and references within), but none was ever as popular as the traditional depth limits.

Silva *et al.* [9] took a slightly different path to replace the traditional tree depth limits. It is based on a single limit imposed on the amount of resources available to the whole **GP** population, where resources are the tree nodes or other elements in non tree-based **GP** (e.g. code lines). The authors claim that the resource-limited technique removes the disadvantages of using depth limits at the individual level, while introducing automatic population resizing, a natural side-effect of using an approach at the population level. Their results show that the replacement of individual depth limits by a population resource limit can be done without impairing performance, thus validating this first and important step towards a new approach to improving the efficiency of **GP**.

1.2 Hypothesis and motivation

Part of the authors research involves the use of **GP** for modelling the behavior of human subjects when confronted with a particular classification task: the **GP** technique was used for performing regression on a mathematical function mim-

[1]For details on **GP**, we invite interested readers to consult [3] or [1]

icking the answers given by the subjects. Because of the inherent variability in human responses, the data set for the experiment could contain several *correct* answers for each row of entry data. So, instead of having

$$\{x_1, x_2, \cdots, x_n\} \rightarrow y$$

we do have

$$\{x_1, x_2, \cdots, x_n\} \in [y_{min}, y_{max}]$$

So, in a sense, our data set has been *relaxed*: instead of asking the **GP** algorithm to obtain the *perfect* answer y, we want to obtain anything in the range $[y_{min}, y_{max}]$

Our research question is now: does this *relaxation* improves the generalization error of the **GP**? What effect does it have on the bloat behavior of the algorithm?

2. EXPERIMENTAL SETTING

2.1 The problem

A simple problem was used for the experiments: symbolic regression of the quartic polynomial $Q(x) = x^4 + x^3 + x^2 + x$, with 201 equidistant points in the interval -10 to +10. A training set of 20 points (10% of the total) was randomly chosen; the other 181 points were assigned to the test set; these two sets are presented in Fig. 1.

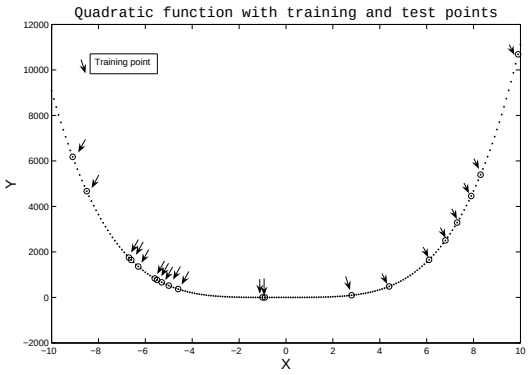

Figure 1: Training set (arrows) and test set

The values shown as *training points* in Fig. 1 are displayed in Table 1.

Table 1: Training points

Points			
x	y	x	y
-9.1000	6178	-8.5000	4670
-6.7000	1753	-6.6000	1647
-6.3000	1359	-5.6000	834
-5.5000	773	-5.3000	663
-5.0000	520	-4.6000	367
-1.0000	0	-0.9000	0
2.8000	94	4.4000	484
6.1000	1655	6.8000	2506
7.3000	3289	7.9000	4458
8.3000	5395	9.9000	10684

We prepared 9 training sets that differ in the amount of relaxation allowed on the target values; these values are shown in Table 2, and are based on the actual width of the values taken by Q in the interval studied ($x \in [-10, +10] \Rightarrow Q(x) \in [-0.3264, 11110]$)

Table 2: Relaxation values

Interval width			
Percentage	Absolute	Percentage	Absolute
0.125%	13.89	0.25%	27.78
0.5%	55	0.75%	83.33
1%	110	5%	555
10%	1110	15%	1666.07
20%	2222.07		

The effect of these intervals is to modify the shape of the fitness function used for the evaluation of solutions; a *perfect* solution for each of the x in Table 1 is relaxed by the absolute value in Table 2. For example: for relaxation set at 0.125%, a perfect value for $x = -9.1$ is $y = 6178 \pm 13.89$, so $y \in [6164.11, 6191.89]$ The 0.125%-*relaxation values* are shown in Table 3

Table 3: Training points relaxed at 0.125%

Points			
x	$y \in$	x	$y \in$
-9.1000	$[6164.11, 6191.89]$	-8.5000	$[4656.11, 4683.89]$
-6.7000	$[1739.11, 1766.89]$	-6.6000	$[1633.11, 1660.89]$
-6.3000	$[1345.11, 1372.89]$	-5.6000	$[820.11, 867.89]$
-5.5000	$[759.11, 786.89]$	-5.3000	$[649.11, 676.89]$
-5.0000	$[506.11, 533.89]$	-4.6000	$[353.11, 380.89]$
-1.0000	$[-14.11, 13.89]$	-0.9000	$[-14.11, 13.89]$
2.8000	$[80.11, 107.89]$	4.4000	$[470.11, 497.89]$
6.1000	$[1641.11, 1668.89]$	6.8000	$[2492.11, 2519.89]$
7.3000	$[3275.11, 3302.89]$	7.9000	$[4444.11, 4471.89]$
8.3000	$[5381.11, 5408.89]$	9.9000	$[10670.11, 10697.89]$

Similar tables are generated for each of the 9 relaxation intervals; their use is presented in the next section (2.2)

2.2 The algorithm

An initial population of 400 individuals (randomly fully initialized with a maximum depth of 6) was evolved for 50 generations. The details for the **GP** algorithm are given here:

- **Genetic operators**: standard tree crossover and tree mutation (details in [3]).
- **The function set** F = $\{+, -, \times, \div, log, exp, sqrt\}$[2]
- **The terminal set** T = $\{x$ (constant), $r \in \mathbb{R} \wedge r \in [0, 1], n \in \mathbb{N} \wedge n \in [1, 10]\}$
- **Selection** for reproduction used the lexicographic parsimony pressure tournament [5] and selection for survival used no elitism.

[2]The *log* and *srqt* function were *protected*: *log* of a negative number is equal to 0, and square root of a negative number is also 0

- **The fitness function** was defined in order to take account of the fact that, whenever a candidate solution belongs to the relaxation interval defined, it is considered *perfect*. This is expressed by Eq. 2

  1. A candidate solution $\overline{\mathcal{S}}$ has 20 approximations for the training points: $\overline{\mathcal{S}} = \{\overline{s_1}, \overline{s_2}, \cdots, \overline{s_{20}}\}$
  2. The perfect solution $\widehat{\mathcal{S}}$ is made by 20 intervals to approximate:

$$\widehat{\mathcal{S}} = \{\overbrace{[\widehat{s_1}^{min}, \widehat{s_1}^{max}]}^{\widehat{s_1}}, \cdots, \overbrace{[\widehat{s_{20}}^{min}, \widehat{s_{20}}^{max}]}^{\widehat{s_{20}}}\}$$

  3. The distance between an approximation to a point and an interval is defined as in Eq. 1

$$d(\overline{s_i}, \widehat{s_i}) = \begin{cases} 0 & \overline{s_i} \in \widehat{s_i} \\ |\overline{s_i} - (\frac{\widehat{s_1}^{max} - \widehat{s_1}^{min}}{2})| & \text{otherwise} \end{cases} \quad (1)$$

  4. The fitness of $\overline{\mathcal{S}}$ is:

$$f(\overline{\mathcal{S}}, \widehat{\mathcal{S}}) = \sum_{i=1}^{20} d(\overline{s_i}, \widehat{s_i}) \quad (2)$$

3. RESULTS

A total of 40 runs were performed with each of the intervals. We used **GPLAB**, a **GP** toolbox for **MATLAB** [8], modified in order to allow optimization by intervals. The population of candidate solutions was optimized with the training set, and then the generalization error $\mathcal{G}_e$ was calculated on the validation set $\mathcal{S}^G$ (181 points). This error is defined as the sum of the error on each of the points in the set (see Eq. 3). The *resources used* by the best solutions were also computed.

$$\mathcal{G}_e(\overline{\mathcal{S}}, \mathcal{S}^G) = \sum_{i=1}^{181} d(\overline{s_i}, s_i^G) \quad (3)$$

This equation stresses the fact that the relaxation is done **only** in the training set, not in the generalization set.

3.1 Generalization error

Results for the generalization error are shown in Fig. 2 (the **0%** relaxation is, of course, equivalent to approximate the exact values on the training set) as well as in Fig. 3.

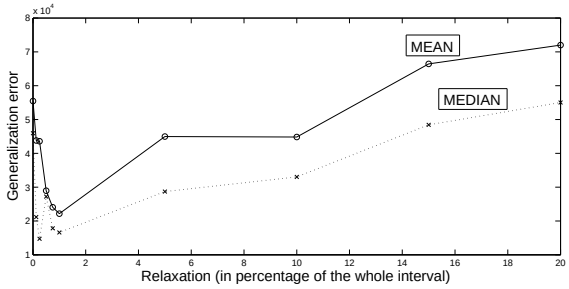

Figure 2: Generalization Error

The generalization error drops from the 0% mark, hitting a low value at 1% relaxation. It is interesting to see that the values for the average error are under the values for the 0% until the 10% mark. So even a small relaxation causes the generalization error to drop substantially.

These results seem to confirm our work hypothesis that there is an advantage in using a little relaxation in order to allow the algorithm some latitude.

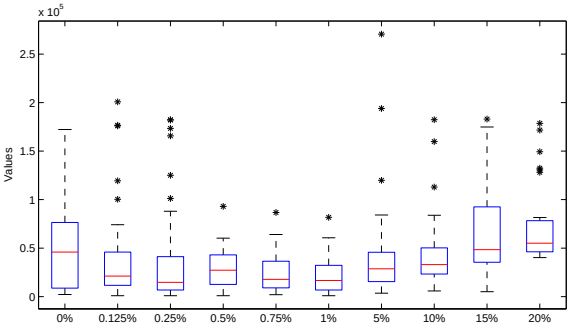

Figure 3: Boxplot generalization error

3.2 Resources used

The resources used by the candidate solutions are expressed in terms of *number of nodes* in the tree. Fig. 4 presents the resources used on average for the 40 runs by the best solution, expressed along the generations of the run.

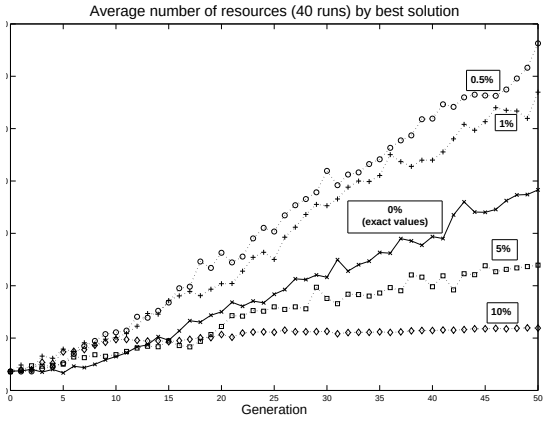

Figure 4: Resources used

Fig. 4 presents the results for 0.5%, 1%, 5%, 10% relaxation, along with exact values (relaxation = 0%); we chose not to show the results for relaxations 0.125%, 0.25% and 0.75%, as the values were very similar to that obtained for 0.5%. Likewise, results for 20% relaxation are similar to that of 10% relaxation .

We can observe 2 distinct categories: for relaxation values up to 1%, the number of resources used are greater than that of a no relaxation. For values greater than 1% (in our experiment, 5, 10, and 20%), the best solutions actually

used up *less* resources than the solutions generated with the exact training set.

It is interesting to note the results for 10% relaxation, as it shows that the amount of resources used remains constant after a small number of generations. This result shouldn't surprise us, as it is intuitively logic that if a solution has to approximate a rather wide target, it will not results in a *complicated* **GP** tree. This observation remains to be supported by more experimentation.

4. DISCUSSION AND FUTURE WORK

The results for this preliminary study show very encouraging signs: the generalization error of certain solutions was actually lower than the error for the exact set and the bloat phenomenon, an important concern for **GP** solutions, was also less important. While it shouldn't surprise us that, for more relaxed sets, the solutions would be less *bloated*, it is remarkable that the generalization error could decrease with less restrictions on the training set.

However, some remarks have to be made; our results show that the generalization error was best than the error for the exact valued-solution for relaxation $\leq 1\%$ (Fig. 2), and at the same time, the bloat values for the results *larger* than 1% were better than the best valued-solution (Fig. 4). So there seems to be a compromise between the *quality* (in terms of approximation) of the solutions and their *quality*, in terms of bloat. More measures with relaxation in the range $[1\%, 10\%]$ would be important in order to quantify this compromise, if there is any.

It is also important to note that the values presented in Fig. 2, for the generalization error, have to be weighted with the statistics shown in the boxplot of Fig. 3; we graphically see that the values between each relaxation interval are not statistically different. So, in order to have a clear picture of the real improvement on error, we should repeat the experiments and try to extract meaningful statistical conclusions. However, the potential of this simple relaxation approach could mean the definition of a new control method for bloat that, at the same time, would improve the performance of the **GP** algorithm.

5. REFERENCES

[1] Wolfang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic Programming An Introduction*. Morgan Kauffman Publishers, San Francisco, 1998. An interesting introduction to genetic programming.

[2] John R. Koza, David Andre, Forrest H Bennett III, and Martin A. Keane. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using genetic programming. In J.R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 132–140. The MIT Press, Stanford University. Cambridge, MA, 1996.

[3] J.R. Koza. *Genetic Programming-On the programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, London, 1992.

[4] J.R. Koza. Evolution of a computer program for classifying protein segments as transmembrane domains using genetic programming. In Russ Altman, Douglas Brutlag, Peter Karp, Richard Lathrop, and David Searls, editors, *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology*, pages 244–252. AAAI Press, Menlo Park, CA, 1994.

[5] Sean Luke and L. Panait. Lexicographic parsimony pressure. In W.B. Langdon and et al, editors, *Proceedings of GECCO-2002*, pages 829–836. Morgan Kauffmann, San Francisco, CA, 2002.

[6] L. Panait and Sean Luke. Alternative bloat control methods. In Kalyanmoy Deb and et al, editors, *Proceedings of GECCO-2004*, pages 630–641. Springer, Berlin, 2004.

[7] Ricardo Poli. A simple but theoretically-motivated method to control bloat in genetic programming. In Ryan C. and et al, editors, *Proceedings of EuroGP-2003*, pages 204–217. Springer, Berlin, 2003.

[8] Sara Silva. Gplab-a genetic programming toolbox for matlab, 2004. http://gplab.sourceforge.net.

[9] Sara Silva, Pedro J.N. Silva, and Ernesto Costa. Resource-limited genetic programming: Replacing tree depth limits. In Bernardete Ribeiro and et al, editors, *Adaptive and Natural Computing Algorithms*, pages 243–246, Coimbra, Portugal, 2005. Springer. http://icannga05.dei.uc.pt/.

[10] T Soule and J.A. Foster. Effects on code growth and parsimony pressure on populations in genetic programming. *Evolutionary Computation*, 6(4):293–309, 1999.