GPLAB – A Genetic Programming Toolbox for MATLAB

Sara Silva [1] and Jonas Almeida [1,2]

[1] Biomathematics Group, Instituto de Tecnologia Química e Biológica,
Universidade Nova de Lisboa, PO Box 127, 2780-156 Oeiras, Portugal
sara@itqb.unl.pt
[2] Dept Biometry & Epidemiology, Medical University of South Carolina,
135 Cannon St, Suite 303, PO Box 250835, Charleston SC 29425, USA
almeidaj@musc.edu

Abstract

This paper presents GPLAB, a genetic programming toolbox for MATLAB. Besides most of the features traditionally used in genetic programming, it also implements two techniques aimed at controlling the well known bloat problem, as well as a modified version of a previously published method for automatically adapting the genetic operator probabilities in runtime, which makes it possible to use the toolbox as a test bench for new genetic operators. Combining a highly modular and adaptable structure with automatic parameterization techniques, the toolbox suits all kinds of users, from the layman who wants to use it as a "black box", to the advanced researcher who wants to build and test new functionalities. The toolbox and its documentation are freely available for download at `http://www.itqb.unl.pt:1111/gplab/`.

1 Introduction

Genetic programming is the most recent paradigm of evolutionary computation [1, 6]. It solves complex problems by evolving populations of computer programs, using Darwinian evolution and Mendelian genetics as inspiration. Until recently, there were a number of public domain genetic algorithm toolboxes for MATLAB [3, 5], but none specific for genetic programming. GPLAB was developed to provide such a free toolbox that can be used and further developed by others. Accordingly, a careful programming discipline was adopted to ensure code reusability and easy maintenance. The result was a versatile, generalist and extendable tool, able to accommodate a wide variety of usages. It was tested on different MATLAB versions and computer platforms, namely version 5.2 (R11) for Windows and version 5.3 (R13) for Windows and Linux. GPLAB and its documentation are released under GNU General Public License, and freely available for download at `http://www.itqb.unl.pt:1111/gplab/`.

2 Operational structure

The architecture of GPLAB follows a highly modular and parameterized structure, which different users may use at various levels of depth and insight. What follows is a visual description of this structure, along with brief explanations of some operation details and control parameters. A summary of three usage profiles appropriate for the different types of users can be found in the user's manual accompanying the toolbox. Some demo functions are available.

Figure 1 shows the operational structure of GPLAB. There are three main operation modules, namely SET VARS, GEN POP, and GENERATION, each representing an interaction point with the user. Inside each main module the submodules are executed from top to bottom, the same happening inside INITIAL PROBS and ADAPT PROBS. The description of these two can be found in Sect. 3.3. Any module with a question mark can be skipped, depending on the parameter indi-

cated above it. Each module may use one or more parameters and one or more user functions. User functions implement alternative or collective procedures for realizing a module, and they behave as plug and play devices. Please refer to the user's manual for details and references on these.

GENPOP. This module generates the initial population (INIT POP) and calculates its fitness (FITNESS). The individuals in GPLAB are tree structures created using one of the available initialization methods: Grow, Full, or Ramped Half-and-Half. The functions available to build the trees include the if-then-else statement and some protected functions, plus any MATLAB function that verifies closure. The terminals include a random number generator and all the variables necessary, created in runtime.

Fitness is, by default, the sum of absolute differences between the obtained and expected results in all fitness cases. The lower the fitness value, the better the individual. This is the standard for symbolic regression problems ("regfitness" in Fig. 1), but GPLAB accepts any other plug and play function to calculate fitness, like the function for artificial ant problems, also provided ("antfitness").

GEN POP is called by the user. It starts by requesting some parameter initializations to SET VARS, and finishes by passing the execution to GENERATION. If the user only requests the creation of the initial generation, GENERATION is not used.

GENERATION. This module creates a new generation of individuals by applying the genetic operators to the previous population (OPERATOR). Standard tree crossover and tree mutation are the two genetic operators available as plug and play functions. They must have a pool of parents to choose from, created by a SAMPLING method, which may or may not base its choice on the EXPECTED number of offspring of each individual. Three sampling methods (Roulette, SUS, Tournament) and three methods for calculating the expected number of offspring (Absolute, Rank85, Rank89) are available as user functions, and any combination of the two can be used. A fourth sampling method ("lexictour" in Fig. 1) is one of GPLAB's special features, described in Sect. 3.

The genetic operators create new individuals until a new population is filled, a number determined by the generation gap. This parameter can be set to any positive integer, causing GPLAB to be able to operate either in generational mode (when generation gap equals the population size), any level of steady-state mode (generation gap lower than population size), or what can be called a batch mode (generation gap higher than population size). In GPLAB there are no strict frontiers between modes, and any set of parameters can be used with any generation gap value.

Calculating fitness is followed by the SURVIVAL module, where the individuals that enter the new generation are chosen according to the elitism level parameter. The GENERATION module repeats itself until the stop condition is fulfilled, or when the maximum generation is reached. Several stop conditions can be used simultaneously. This module can be called either by the user or by GEN POP.

SET VARS. This module either initializes the parameters with the default values or updates them with the user settings. Besides the parameters directly related to the execution of the algorithm, other parameters affect the output of its results. Please refer to the user's manual for a complete description. SET VARS can be called either by the user or by a request for parameter initialization from GEN POP.

3 Special features

This section describes some of the features implemented in GPLAB and not usually found in other genetic programming software packages, discussing their possible implications and presenting a few examples. All the plots presented were generated by toolbox functions.

3.1 Controlling bloat

In genetic programming, code growth is a healthy result of genetic operators in search of better solutions. Unfortunately, it also permits the appearance of pieces of redundant code, called introns, which increase the size of programs without improving its fitness. Besides consuming precious

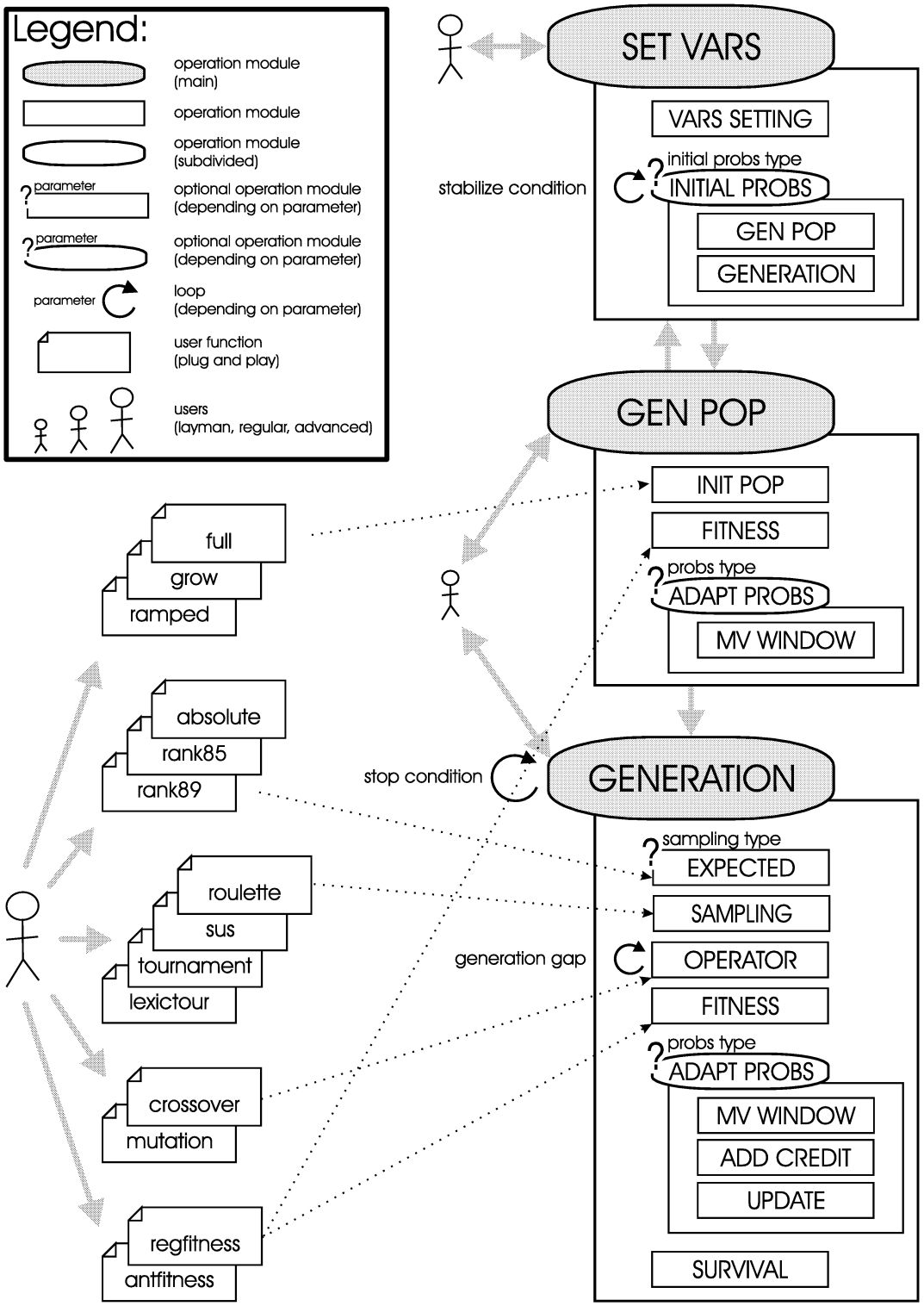

Figure 1: Operational structure of GPLAB. The layman interacts with the toolbox using default parameter values; the regular user tries different settings; the advanced user builds new plug and play functions. The three usage profiles are described in the user's manual.

time in an already computationally intensive process, introns may start growing rapidly, a situation known as bloat [2, 10]. Several techniques have been used in the attempt to control bloat (reviews in [7, 9]), and GPLAB implements two of the most recently successful, lexicographic parsimony pressure [7] and dynamic maximum tree depth [8]. Each one of them can be used exclusively, but the conjunction of both has shown to be more efficient in the battle against bloat [8].

Lexicographic parsimony pressure. This is a multiobjective technique that optimizes both fitness and size at the same time, without assigning a new fitness value. Instead, it uses a modified tournament selection operator ("lexictour" in Fig. 1) that treats fitness as the primary objective and tree size as the secondary objective, in a lexicographic ordering. This technique has shown to be very effective in problems where many different individuals have the same fitness [7], meaning where code growth is mainly caused by introns.

Dynamic maximum tree depth. This technique introduces a dynamic limit to the maximum depth of the individuals allowed in the population. It is similar to the traditional Koza-style strict limit commonly imposed on tree depth [6], but does not replace it – both dynamic and strict limits are used in conjunction. The dynamic limit should be initially set with a low value, but at least as high as the depth of the randomly created initial trees. Once increased, it will not be lowered again. If and when the dynamic limit reaches the same value as the strict Koza limit, both limits become one and the same. This technique has been tested in two different problems, where it has surpassed lexicographic parsimony pressure and shown to effectively avoid excessive code growth caused by either introns or exons. The combination of both techniques produced even better results [8].

3.2 Accuracy versus Complexity

The dynamic maximum tree depth parameter may be used for another purpose besides controlling bloat. In real world applications, one may not be interested or able to invest a large amount of time in achieving the best possible solution, particularly in approximation problems. Instead, one may only consider a solution to be acceptable if it is sufficiently simple to be comprehended, even if its accuracy is known to be worse than the accuracy of other more complex solutions. Choosing less stringent stop conditions is not enough to ensure that the resulting solution will be acceptable, as it cannot predict its complexity. GPLAB responds to this by returning, not only the best individual found during the run, but the series of all individuals that have once been considered to be the best during the run. In this series the user can find several solutions of different levels of complexity, and choose which one corresponds better to the desired ratio between accuracy and complexity. It is important to choose a low value for the initial dynamic maximum tree depth, to force the algorithm to look for simpler solutions before adopting more complex ones. When the dynamic maximum level is initially set to a high value, the algorithm does find complex solutions first, and hardly ever discards them in favor of simpler ones. Figure 2 illustrates a run performed with initial dynamic maximum tree depth 5 on a symbolic regression problem (approximation of the exponential function).

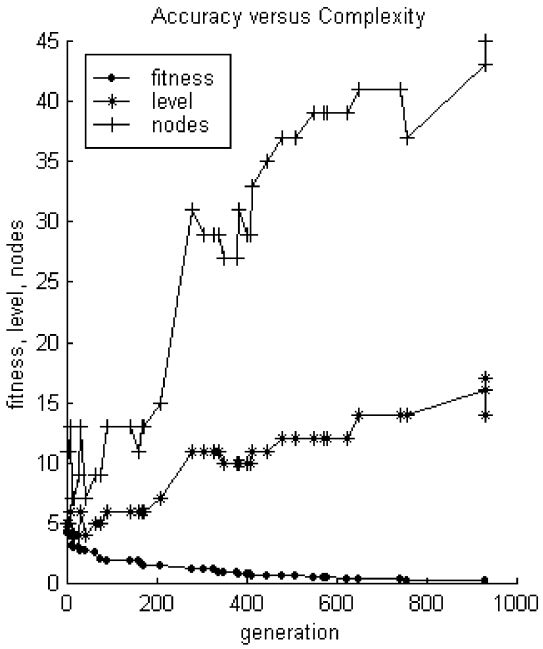

Figure 2: Evolution of fitness, level (tree depth) and number of nodes of the best individual found so far. Is it worth waiting for a more accurate and more complex solution?

3.3 Adapting operator probabilities

Due to the extremely wide range of problems that can potentially be solved by genetic programming, there seems to be no such thing as the ideal set of parameter values to use. Some of the most important and most difficult to set are precisely the ones directly related to the heart of the evolutionary process: the probabilities of occurrence of the genetic operators. GPLAB implements a modified version of a previously published method for automatically adapting these in runtime [4].

The adaptation procedure is represented in Fig. 1 as the module ADAPT PROBS. It keeps track of a certain number of past individuals, by means of a moving window (MV WINDOW), and adds credit to the genetic operators, proportional to the relative quality of their offspring when compared to the previous population (ADD CREDIT). The operators that created their ancestors also receive credit, in a lower amount, passed back until a certain number of backward generations is reached. Only the individuals inside the moving window participate in the credit distribution. From time to time the operator probabilities are updated to reflect how well they have performed during the last interval (UPDATE). The initial operator probabilities can also be automatically set before the run starts, by repeatedly simulating modules GEN POP and GENERATION, which adapts the operator probabilities until a certain stabilization condition is verified (module INITIAL PROBS). The operator probabilities may drop to null values, knowing that whenever an operator reaches the adaptation time without having produced any offspring during the last interval, its credit will be twice the amount of the best operator. This boost will give the operator a new opportunity to prove itself useful. If not, its probability will rapidly drop again.

Figure 3 exemplifies the active role of the adaptation procedure in presence of destructive genetic operators. The initial operator probabilities are all equal, not subject to the initial adaptation procedure. When the two destructive operators "destroy all" and "destroy less" are used in conjunction with the regular genetic operators "crossover" and "mutation" on the Even-3 Parity problem, the pattern shown in Fig. 3 is generated. Both destructive operators pick one parent and produce one offspring by replacing the tree with a numeric constant (-100 in "destroy all", -10 in "destroy less"). Both are dramatically destructive, but "destroy all" is so much worse that it even masks the fact that "destroy less" is also a very bad operator. It immediately drops to the bottom, and all the probability boosts it receives are to no avail.

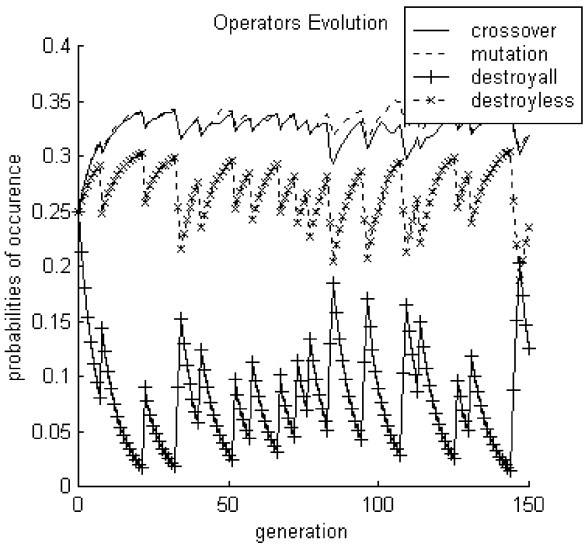

Figure 3: Operator probabilities evolution in presence of destructive operators

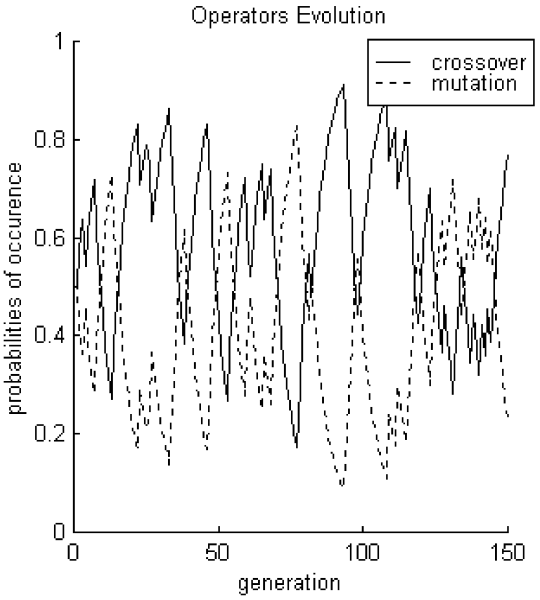

Figure 4: Alternation pattern between the probabilities of crossover and mutation

It is important to remove the genetic operators that do not recover after successive boosts, as they may be preventing an important relationship between the remaining operators. When both destructive operators are removed from the example above, an elegant pattern of alternation between crossover and mutation arises (shown in Fig. 4), as if the positive contribution of one could not be possible without the previous contribution of the other. It is beyond the scope of this report to analyze the conditions that lead to such patterns, and the plots presented are no more than mere illustrations of using the toolbox as a test bench for genetic operators.

4 Future Work

The GPLAB toolbox is not a final product. On the contrary, it was built to serve as a test bench for new genetic programming functionalities, some of which will inevitably be integrated in future versions of the toolbox. Any improvements to GPLAB will be made available as new versions or simply as new user function modules ready to be plugged in the operational structure.

Launched on July 8, 2003, the GPLAB web page already counts several hundreds of visits, and dozens of e-mails have been received by the authors reporting an interest in future updates and suggestions for improving efficiency, including a proposition to parallelize the algorithm. It seems like GPLAB may be filling a gap in evolutionary computation tools for MATLAB.

Acknowledgements

This work was partially supported by grants QLK2-CT-2000-01020 (EURIS) from the European Commission and POCTI/1999/BSE/34794 (SAPIENS) from Fundação para a Ciência e a Tecnologia, Portugal.

References

[1] Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic programming – an introduction, San Francisco, CA. Morgan Kaufmann (1998)

[2] Banzhaf, W., Langdon, W.B.: Some considerations on the reason for bloat. Genetic Programming and Evolvable Machines, 3. Kluwer Academic Publishers (2002) 81–91

[3] Chipperfield, A.J., Fleming, P.J., Pohlheim, H., Fonseca, C.M.: A genetic algorithm toolbox for MATLAB. In Proceedings of the International Conference on Systems Engineering (1994) 200–207

[4] Davis, L.: Adapting operator probabilities in genetic algorithms. In Schaffer, J.D., editor, Proceedings of the Third International Conference on Genetic Algorithms, San Mateo, CA. Morgan Kaufmann (1989) 61–69

[5] Houck, C.R., Joines, J.A., Kay, M.G.: A genetic algorithm for function optimization: a MATLAB implementation. NCSU-IE Technical Report 95-09, North Carolina State University, Raleigh, NC (1995)

[6] Koza, J.R.: Genetic programming – on the programming of computers by means of natural selection, Cambridge, MA. MIT Press (1992)

[7] Luke, S., Panait, L.: Lexicographic parsimony pressure. In Langdon, W.B., Cantú-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R. Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., *et al.*, editors, Proceedings of GECCO-2002. Morgan Kaufmann (2002) 829–836

[8] Silva, S., Almeida, J.: Dynamic maximum tree depth – a simple technique for avoiding bloat in tree-based GP. In Cantú-Paz, E., Foster, J.A., Deb, K., Davis, L.D., Roy, R., O'Reilly, U.-M., Beyer, H.-G., Standish, R., Kendall, G., Wilson, S., *et al.*, editors, Proceedings of GECCO-2003. Springer Verlag (2003) 1776–1787

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

[10] Soule, T., Heckendorn, R.B.: An analysis of the causes of code growth in genetic programming. Genetic Programming and Evolvable Machines, 3 (2002) 283–309