Neurocomputing 180 (2016) 79–93

Contents lists available at ScienceDirect

Neurocomputing

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

Studying bloat control and maintenance of effective code in linear genetic programming for symbolic regression

Léo Françoso dal Piccol Sotto, Vinícius Veloso de Melo *

Institute of Science and Technology (ICT), Federal University of São Paulo (UNIFESP), São José dos Campos, SP, Brazil

ARTICLE INFO

Article history:
Received 31 March 2015
Received in revised form
2 October 2015
Accepted 5 October 2015
Available online 11 November 2015

Keywords:
Bloat control
Effective code
Symbolic regression
Linear genetic programming

ABSTRACT

Linear Genetic Programming (LGP) is an Evolutionary Computation algorithm, inspired in the Genetic Programming (GP) algorithm. Instead of using the standard tree representation of GP, LGP evolves a linear program, which causes a graph-based data flow with code reuse. LGP has been shown to outperform GP in several problems, including Symbolic Regression (SReg), and to produce simpler solutions. In this paper, we propose several LGP variants and compare them with a traditional LGP algorithm on a set of benchmark SReg functions from the literature. The main objectives of the variants were to both control bloat and privilege useful code in the population. Here we evaluate their effects during the evolution process and in the quality of the final solutions. Analysis of the results showed that bloat control and effective code maintenance worked, but they did not guarantee improvement in solution quality.

1. Introduction

Regression analysis is a statistical process of estimating relationships among variables and using that information to perform predictions. Usually, there is a set of independent explanatory variables and a single dependent response variable. Regression techniques are used to build a model that allows the correct/approximate prediction of the response variable using the explanatory variables as input. In fact, a model is assumed a priori, determined by the researcher, and the regression technique estimates the coefficients that minimize a certain criterion, for instance, the least-squared error. On the other hand, symbolic regression (SReg) consists of, without the a priori model, automatically finding a mathematical expression that adjusts as best as possible to these data [15]. That is, given the explanatory and response variables, an SReg technique has to find both the model and the coefficients that minimize it. Therefore, SReg is a harder problem than regular regression. Since one wishes to find the best model, a global optimization technique is supposed to be used, and Evolutionary Computation methods are widely employed for this task.

In order to solve SReg problems, Koza [15] proposed the Genetic Programming (GP) technique, an extension of the Genetic Algorithm (GA) technique [27]. While GA operates on a fixed-length array that is a codification of a problem's solution, GP manipulates a variable-length tree data-structure, that is a code to be used to solve the problem. Thus, techniques such as GP and similar approaches are proper for solving SReg problems and have been widely investigated [26,22,13,10,33,7].

In this work, we used the Linear Genetic Programming (LGP, [5]) technique for solving SReg problems. Instead of working with trees, LGP evolves a linear program in an array to solve a particular task, being the solution directly convertible to an imperative programming language. LGP has been used to efficiently solve various problems in the literature such as symbolic regression, classification [5,2], estimation models [14], and time series modeling [12]. Researches that compare LGP with basic GP [23,5] show that LGP outperforms GP in several tasks, including SReg. All those results suggest that LGP has a high potential as an efficient Genetic Programming variant.

A well-known issue in GP and similar techniques that employ variable-length representations is called bloat: an excessive code growth without a corresponding improvement in fitness [25,28,29]. Bloat may cause several problems: (1) solutions may grow to a point that fills the system memory; (2) huge solutions are slow to be evaluated; (3) spurious parts of the solution may hinder its improvement because modifications of those parts do not change the fitness; (4) bloated solutions are black-boxes. Therefore, bloat is a major and active topic of study in GP [32,1,31,17,4,18,24,9,28].

* Corresponding author.
E-mail addresses: leo.sotto352@gmail.com (L.F.d.P. Sotto), vinicius.melo@unifesp.br (V.V.d. Melo).

http://dx.doi.org/10.1016/j.neucom.2015.10.109
0925-2312/

Usually, researchers evaluate their bloat control methods in a few benchmark problems of distinct tasks, for instance, symbolic regression of benchmark functions, artificial ant [16], even-parity [15], and multiplexer [15]. On the other hand, the objective of the present work is to compare LGP variants, mostly in terms of selection mechanisms for bloat control, focusing on the SReg task. For such purpose, many distinct benchmark functions were selected. Here we proposed and developed a few variants of a basic LGP implementation, analyzed them from two points of view, and tried to establish a link between them to understand:

- their behavior in the evolution process, in terms of the proportion of effective code in the individuals;
- their effects in the quality of the solutions generated.

Those variants, explained in detail in a later section in this paper, include: (1) the usage of bloat control techniques to increase the presence of smaller (simpler) individuals in the population, (2) the usage of techniques to provide a higher percentage of effective code in the population, and (3) an operator that joins two successful individuals into one, intending to treat them as subfunctions. Furthermore, we applied the items 1 and 2 to 3, that is, using the operator that joins two solutions alongside with the techniques to both control bloat and increase the percentage of effective code. The study performed herein is an experimental one, trying to verify the usefulness of the proposed variants. A theory to understand and predict LGP behavior when using such variants is yet to be developed.

The remaining of this paper is structured as follows. Section 2 describes the Methodology of the work, presenting each variant in detail and its pseudocode. The experimental setup is introduced in Section 3. Results and the discussion are presented in Section 4. We end this paper with Conclusions and Future Works in Section 5.

2. Methods

In this section, we explain the basic LGP implementation we used and also present the variations of this basic implementation.

2.1. Linear genetic programming

Linear Genetic Programming (LGP) is an Evolutionary Computation (EC) technique that evolves a linear program, instead of a functional one as in GP, to solve a particular task. The linear representation is its main difference from standard tree-like GP [5]. The individuals in LGP are represented as a sequence of instructions, each using the results of previous instructions, constant values, or input values. A register stores the result of a particular instruction, and other instructions may use registers as arguments for an operator. Registers can be of three types: (1) *calculation registers* (which store results of instructions), (2) *constant registers* (constants predefined by the user), or (3) *input registers*, which are the program's input. An example of an LGP individual is shown in Fig. 1.

The use of such data structure has many consequences. As results of previous calculations can be used by more than one instruction, the connections among the instructions can be seen as a graph-based data flow. This structure helps to reduce the repetition of blocks of code that would be necessary for a tree structure. However, calculation registers may not be used by other instructions, leading to the emergence of *non-effective* code-blocks of code whose results are not used to change the output. Modifications performed on such code are named neutral variations that *were identified as being a main cause of code growth as well as an important motor of evolutionary progress allowing for neutral walks over the fitness landscape* [5].

```
r[4] = r[1] / 1.3;
r[1] = r[2] * -5.123;
r[0] = sqrt(10);
r[3] = r[1] + r[1];
r[1] = log(r[3]);
r[1] = r[3] >= r[0];
r[0] = abs(r[2]);
r[0] = if r[1] then r[0] else r[3];
```

Fig. 1. Example of an individual generated by LGP. The final result is stored in register r[0], as attributed by the last instruction.

As pointed out in [5], structurally non-effective code is easier to appear in individuals than semantic introns (code actually used by the program but that has no effect on the output, for instance, adding zero to the output). Therefore, they may quickly grow individuals to the maximum allowed size, resulting in an implicit parsimony pressure on the effective code. The large amount of non-effective code also implies that neutral variations are more likely to occur. Such individuals may improve and remain compact by reusing results from calculation registers. Also, non-effective code can be enabled (becoming effective) during the evolution resulting in a very distinct solution [3–5].

In a tree-based representation, it is not only harder to identify neutral code but also to remove or insert neutral code. In LGP, this is an easy task because the neutral code is not referenced/used by the output register. Therefore, the ancestors of the output register allow the identification of the registers that are being used and those not being used.

An algorithm for returning only the effective code of an individual is detailed in [5]. The general idea is to begin with the last instruction and move upwards in the list of instructions, identifying instructions that store their results into registers that are not used by instructions below. These instructions are non-effective, while the others are effective. For instance, in Fig. 1 the first instruction stores its result into r[4], which is not used below. Therefore, the first instruction is a non-effective code.

In the presence of non-effective code, traditional evolutionary operators used in tree-based GP may have much more destructive consequences, not only increasing both the bloat and the variation step-size [3,2] but also reducing convergence rates. Such consequences happen because LGP individuals may have just a small part of effective code, and the recombination with another individual is likely to add even more non-effective code than to break the current effective code. However, it cannot be guaranteed that effective code will remain effective after a big segment crossover and what will be the configuration of the same instructions in a different individual. A reason for such issue is that individuals use information from registers and store results into registers. Thus, in recombination, if the second individual does not use the registers employed by the first individual, then the first code becomes non-effective. The same can be said for the mutation procedure: when it replaces an argument that is currently a calculation register by a constant value, the code that stores its result into that calculation register may become non-effective. Therefore, the genetic operators have to be adapted for LGP in order do reduce those negative effects.

As in GP, the main evolutionary operators for LGP are crossover and mutation. Crossover is like in classical discrete Genetic Algorithms with variable-length individuals. Parents are chosen via tournament selection and parts of their code are exchanged (one-