Comparison of Linear Genetic Programming Variants for Symbolic Regression

Léo Françoso Dal Piccol Sotto
Institute of Science and Technology (ICT)
Federal University of São Paulo (UNIFESP)
São José dos Campos, SP, Brazil
leo.sotto352@gmail.com

Vinícius Veloso de Melo
Institute of Science and Technology (ICT)
Federal University of São Paulo (UNIFESP)
São José dos Campos, SP, Brazil
vinicius.melo@unifesp.br

ABSTRACT

In this paper, we compare a basic linear genetic programming (LGP) algorithm against several LGP variants, proposed by us, on two sets of symbolic regression benchmarks. We evaluated the influence of methods to control bloat, investigated these techniques focused in growth of effective code, and examined an operator to consider two successful individuals as modules to be integrated into a new individual. Results suggest that methods that deal with program size, percentage of effective code, and subfunctions, can improve the quality of the final solutions.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming, Program Synthesis, Program Modification; I.2.8 [**Problem Solving, Control Methods, and Search**]: Heuristic Methods; D.1.2 [**Programming Techniques**]: Automatic Programming

Keywords

Symbolic Regression; Linear Genetic Programming

1. INTRODUCTION

Linear genetic programming is a type of genetic programming that evolves linear programs, causing its data flow to form a graph [2]. In LGP, individuals are represented as a sequence of instructions, each using results of previous instructions, constant values, or input values, and storing its results to registers.

Works that have tested LGP in some instances of symbolic regression problems have reported that it outperforms basic TGP (tree-based GP) [2, 5]. In this work, the focus is to analyse how modified versions perform when compared to the basic implementation.

The baseline LGP implementation in this work is based only in macro and micro-mutations, taking into consideration the results obtained in [1], which was the setup that presented the best results. We named this algorithm **eff-mut**. Macro-mutations consist of inserting or deleting a whole instruction, while micro-mutations consist of changing a destination register, an argument or an operator within an instruction.

All variations compared in this work were configured with random generation of initial population, size of population and number of generations equal to 1000, initial size of individuals 20 and maximum size 200, tournament of size 10 with elite of size 1, conditional reproduction (but for **union_lgp**), eight registers (but for **union_lgp**, which used 4), rate of micro mutations 25% and of macro mutations 75%, being 66% of these the rate of insertions and 33% of deletions. The constants used were integers from one to 9, and π. The function set used was: $+$, $-$, $*$, $/$, x^y, e^x, $\sqrt{x}$, $log(x)$, $sin(x)$, and $cos(x)$.

Two variations among the ones tested include the usage of non-parametric parsimony pressure [3] in order to control the bloat and verify if, when the population is composed mainly by smaller individuals, the evolution process can be made easier and lead to better quality solutions. One of the techniques is double tournament, which selects individuals based on fitness and then, in the second stage of the tournament, chooses the individuals based on size. The other variation uses proportional tournament, which has a probability P of doing the tournament based on fitness or size. In this paper it is used $P = 0.5$. We called these variations **double_lgp** and **prop_lgp**.

Two other variations consist of using these same parsimony pressure mechanisms modified to privilege individuals with a higher percentage of effective code. In LGP, individuals tend to have a lot of non-effective code, that is, instructions that are stored in registers that are never used as arguments in the individual. By using these two mechanisms, the aim is to investigate if a higher percentage of structurally effective code in the population improves the evolution process and increases the probability of generating better quality solutions. We called these variations **eff_double_lgp** and **eff_prop_lgp**.

The last variation is inspired by an issue observed more easily when working with simpler functions. Suppose one wants to find the function $f(x) = x^3 + x^2$, and one has the individuals $f(x) = x^2$ and $f(x) = x^3$. Given the constant and function sets, the chance of $f(x) = x^3$ becoming $f(x) = x^3 + x^2$ after a sequence of mutations is very low, which usually causes these individuals to be stuck in local optima. Considering that this problem can also occur in

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.

http://dx.doi.org/10.1145/2598394.2598472.

Table 1: Median R^2 over 50 executions for each algorithm in the sets of Nguyen and Keijzer. Each value is accompanied by a signal <, = or >, which corresponds to the method performing worse, equal or better, respectively, than effmut for the same function, according to the Wilcoxon test using $\alpha = 1\%$.

	effmut	double_lgp	prop_lgp	eff_double_lgp	eff_prop_lgp	union_lgp
Function	R^2	R^2	R^2	R^2	R^2	R^2
Nguyen1	0.995	0.987 =	0.999 >	0.999 >	0.998 >	1.000 >
Nguyen3	0.998	0.984 <	0.998 =	0.998 =	0.998 =	0.999 =
Nguyen4	0.991	0.987 <	0.991 =	0.992 =	0.994 =	0.997 >
Nguyen5	0.889	0.825 =	0.912 =	0.991 >	0.905 =	0.954 >
Nguyen6	0.994	0.991 =	0.995 =	0.999 >	0.994 =	1.000 >
Nguyen7	0.998	0.997 =	0.998 =	0.999 >	0.998 =	0.999 >
Nguyen8	1	1 =	1 =	1 =	1 =	1 =
Nguyen9	0.990	0.981 <	0.990 =	0.990 =	0.990 =	1.000 >
Nguyen10	0.996	0.971 <	0.996 =	0.996 =	0.996 =	0.999 >
Keijzer3	0.2561	-0.0172 <	0.1721 =	-0.0169 <	0.2659 =	0.2385 =
Keijzer4	0.46072	0.00508 <	0.39933 =	0.42721 =	0.43721 =	0.45310 =
Keijzer5	0.9485	0.7597 <	0.9471 =	0.9318 =	0.9502 =	0.9486 =
Keijzer7	1	1 =	1 =	1 =	1 =	1 =
Keijzer8	1	1 =	1 =	1 =	1 =	1 =
Keijzer10	1	1 =	1 =	1 =	1 =	1 =
Keijzer11	0.945	0.945 =	0.945 =	0.945 =	0.945 =	0.924 <
Keijzer12	0.784	0.784 =	0.890 =	0.925 =	0.784 =	0.978 >
Keijzer13	-0.18077	-0.06270 =	0.13328 =	-0.11850 =	0.00248 =	0.91490 >
Keijzer14	0.196	0.257 =	0.569 =	0.623 =	0.478 =	0.774 >
Keijzer15	0.875	0.739 <	0.866 =	0.783 <	0.864 =	0.899 =
Count <		8	0	2	0	1
Count =		12	19	14	19	9
Count >		0	1	4	1	10

more complex functions (though not necessarily so close to the optimum solution), it was devised a simple crossover operator that combines the effective code of two individuals by adding their results. We called this last variation **union_lgp** and used this union operator in the place of macro-mutations, while keeping micro mutations.

We have tested the variations described above in the sets of symbolic regression benchmark functions provided by Nguyen and Keijzer in [4], which contains details about the training and testing sets proposed for evaluation. Nguyen functions are mostly single-variable polynomial problems, whereas Keijzer functions include more variables and constants, and a different set for testing, thus being more difficult to achieve better results.

We have performed 50 executions for each pair of algorithm and function on the training set. Table 1 presents the median coefficient of determination (R^2) - that is, the median of the best individuals of each run. The values shown in Table 1 for the Keijzer set correspond to the validation set, while for the Nguyen set they correspond to the training set, as no testing set was specified in [4].

In Table 1, it can be observed that lgp_prop, lgp_prop_eff, and lgp_double_eff achieved some better results when compared to effmut. However, the improvements in R^2 was not significant in the majority of functions (see the Wilcoxon's result) to indicate that a higher percentage of both smaller individuals and effective code in the population leads to a better evolution and to the generation of better quality solutions.

For some of the Keijzer problems, even when the median R^2 was better in the proposed methods, the statistical test resulted in no significant differences. It was observed that very low quality models (*outliers*) influenced the comparison, but they can not be ignored.

As can be noticed, lgp_union was the variation that had the best performance in these experiments (count of > was 10). The strategy of merging two good solutions, although simple, was able to improve results, as desired. That certifies that this technique works also in more complex problems than the one given in the example, since some results with union_lgp for the Keijzer set were better than the ones with the other proposed methods.

Acknowledgments

This work was supported by Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP – Proc. 2013/20606-0).

2. REFERENCES

[1] M. Brameier and W. Banzhaf. Effective linear genetic programming. Technical report, Department of Computer Science, University of Dortmund, 44221 Dortmund, Germany, 2001.

[2] M. Brameier and W. Banzhaf. *Linear Genetic Programming*. Springer, 2007.

[3] S. Luke and L. Panait. Fighting bloat with nonparametric parsimony pressure. In *Proceedings of the 7th International Conference on Parallel Problem Solving from Nature*, PPSN VII, pages 411–421, London, UK, UK, 2002. Springer-Verlag.

[4] J. McDermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong, and U.-M. O'Reilly. Genetic programming needs better benchmarks. In *Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference*, GECCO '12, pages 791–798, New York, NY, USA, 2012. ACM.

[5] M. Oltean and C. Grosan. A comparison of several linear genetic programming techniques. *Complex Systems*, 14(4):285–314, 2003.