On Improving Grammatical Evolution Performance in Symbolic Regression with Attribute Grammar

Muhammad Rezaul Karim
Department of Computer Science
University of Calgary, Canada
mrkarim@ucalgary.ca

Conor Ryan
BDS Group, Department of CSIS
University of Limerick, Ireland
conor.ryan@ul.ie

ABSTRACT

This paper shows how attribute grammar (AG) can be used with Grammatical Evolution (GE) to avoid invalidators in the symbolic regression solutions generated by GE. In this paper, we also show how interval arithmetic can be implemented with AG to avoid selection of certain arithmetic operators or transcendental functions, whenever necessary to avoid infinite output bounds in the solutions. Results and analysis demonstrate that with the proposed extensions, GE shows significantly less overfitting than standard GE and Koza's GP, on the tested symbolic regression problems.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms

Algorithms

Keywords

Attribute Grammar, Symbolic Regression

1. INTRODUCTION

GE (a grammar-based GP system) can generate numerous ineffective subexpressions for the symbolic regression problem. Those ineffective subexpressions can be generated due to the presence of some special structures called invalidators [3], which nullify the effect of ineffective code. These ineffective subexpressions not only increase the size of the derived solutions, but may lead to poor generalization ability of the evolved models. Because of the limited expressive power of CFGs used in standard GE, it is not possible to avoid invalidators in GE evolved symbolic regression models. In this paper, we show how AG can be effectively used with GE to generate symbolic regression expressions without generating certain types of invalidators as well as how interval arithmetic can be implemented using AG. Unlike the previous application of interval arithmetic in GP, where interval method was used to perform static analysis to discard certain trees [2], here interval method is used in a different way. Interval arithmetic is used in this paper to avoid selection of certain arithmetic operators or transcendental functions that leads to infinite bound for the resulting expression, if selected and applied at a particular context.

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.
ACM 978-1-4503-2881-4/14/07.
http://dx.doi.org/10.1145/2598394.2598488 .

2. AG FOR SYMBOLIC REGRESSION

Table 1: AG for the symbolic regression problem

Rule	Semantic Function
<Prog> := <Expr>	
<Expr>$_1$:= (<Expr>$_2$ <Expr>$_3$ <Op>)	Expr$_1$.last = 'DC' Op.allowed = genAllowedOp(Expr$_2$,Expr$_3$,Op) setOpIntervalBounds(Expr$_1$,Expr$_2$,Expr$_3$,Op)
<Expr>$_1$:= (<Expr>$_2$) <Pre-op>	Pre-op.allowed=genAllowedPreOp(Expr$_2$) Expr$_1$.last = Pre-op.last setPreOpIntervalBounds(Expr$_1$,Expr$_2$,Pre-op)
<Expr> := <Var>	Expr.last = Var.last Expr.low = Var.low Expr.up= Var.up
<Expr> := <Const>	Expr.last = Const.last Expr.low = Const.low Expr.up = Const.up
<Op> := +	Op.last = '+'
<Op> := *	Op.last = '*'
<Op> := /	Op.last = '/'
<Op> := −	Op.last = '-'
<Pre-op> := sin	Pre-op.last = 'sin'
<Pre-op> := cos	Pre-op.last = 'cos'
<Pre-op> := exp	Pre-op.last = 'exp'
<Pre-op> := log	Pre-op.last = 'log'
<Const> := 1.0	Const.last = '1.0' setConstantIntervalBounds($Const$,1.0)
<Const> := 2.0	Const.last = '2.0' setConstantIntervalBounds($Const$,2.0)
...	...
<Const> := 9.0	Const.last = '9.0' setConstantBounds($Const$,9.0)
<Var> := x_0	Var.last = 'x_0' setVariableIntervalBounds(Var,x_0)
...	...
<Var> := x_n	Var.last = 'x_n' setVariableIntervalBounds(Var, x_n)

We use a postfix L-attributed [1] grammar for the symbolic regression problem. The advantage of using postfix grammar is that it ensures that the values of all the necessary operands for an operator or a function are available, before those values can be used in the interval arithmetic to rule out some operators from <Op> or <*Pre-op*>. Every non-terminal symbol in this grammar has a synthesized attribute *last* to store the last selected variable, last operator, last selected function or last constant value, depending on the non-terminal. The non-terminal symbol <*Op*> and <*Pre-op*> in this grammar has an inherited attribute *allowed*. This

Table 2: Best of run individuals produced by the three methods. NRMS [2] refers to the training performance in terms of percentage point and is averaged over 50 runs. Linearly scaled MSE (mean squared error) is used while calculating NRMS [2]. '*' indicates that the difference between GE with AG and the method as indicated by a column is statistically significant (*Mann-Whitney U* test with $p < 0.05$). When the difference is statistically significant, the value of *Vargha-Delaney A* measure is shown beside '*' in parenthesis. Overfitting refers to the destructing overfitting on test data and it indicates the percentage of times the MSE exceeds 10000. For problem B, the training set is [0 : 1 : 100] [2], while the test set is rnd(0,100) [2] with 50 cases. For the other problems, the training set is [-3 : 0.1 : 3], while the test set is rnd(-3,3)(50 cases).

Problem	GE with AG		GE with CFG			Koza-GP		
	NRMS	Overfitting	NRMS	Overfitting	S-test	NRMS	Overfitting	S-test
A. $f(x) = 0.3xsin(2\pi x)$	62.670± 1.587	0%	38.640 ± 8.860	94%	*(0.040)	53.700 ± 7.630	0%	*(0.110)
B. $f(x) = arcsinh(x)$	0.996 ± 0.002	0%	0.994 ± 0.000	0%	*(0.000)	0.995 ± 0.000	82%	*(0.000)
C. $f(x) = xy + sin((x-1)(y-1))$	14.773 ± 0.260	0%	17.790 ± 0.030	0%	*(1.000)	13.390 ± 0.700	4%	*(0.001)
D. $f(x) = x^4 - x^3 + y^2/2 - y$	3.755 ± 2.389	4%	6.660 ± 2.116	94%	*(0.914)	4.700 ± 1.369	18%	*(0.805)
E. $f(x) = x^3/5 + y^3/2 - y - x$	8.384 ± 7.026	2%	12.497 ± 0.961	86%	*(0.902)	14.538 ±1.531	16%	*(0.920)

attribute holds a list of valid binary operators or transcendental functions, depending on the non-terminal. When <*Pre-op*> is expanded, it can select a function from this list of valid functions, represented by the attribute *allowed*. For this non-terminal, *genAllowedPreOp*, an auxiliary function of the corresponding semantic function is used to compute the value of *allowed*. This auxiliary function does two things. First, it checks whether the value of the last selected function is *exp* or not. If it is, it does not include *exp* in the list so that any structure like $exp(exp(exp(exp(x))))$ can be avoided. This structure is a kind of invalidator leading to ∞. Second, using interval arithmetic, the lower (the value of the attribute *low*) and upper bound (the value of the attribute *up*) of the symbol $<Expr>_2$ of the relevant production (See Table 1), it rules out some of the possible functions, which if selected and applied on the subexpression represented by the symbol $<Expr>_2$, leads to infinite bound.

For the non-terminal <*Op*>, the value of the attribute *allowed* is evaluated using *genAllowedOp*, an auxiliary function of the corresponding semantic function. Like the other auxiliary function *genAllowedPreOp*, it also performs two checks: one to avoid invalidators (e.g. 0) that is caused by subexpressions like *x*-*x* or *c*-*c* and the other is to avoid expressions with infinite output bound. These two checks are based on the value of the attribute *last* and attributes representing interval bounds (*low* and *up*), respectively, of the two non-terminals $<Expr>_2$ and $<Expr>_3$ representing two input subexpressions in the relevant production. Using interval arithmetic, this function rules out some of the possible binary operators, which if selected at the current context and applied on these two input subexpressions, leads to infinite output bound. All semantic functions in the grammar with name in the format $setXXXIntervalBounds$ (XXX can be 'Op' for binary operators, 'Variable' for variables etc.), set the lower and upper interval bounds of the symbols on the head of the relevant productions.

3. EXPERIMENTATION

For GE, in all experiments, we use a steady state GA, a population size of 500 individuals, roulette wheel selection, one-point crossover with a probability 0.9, and bit mutation with probability 0.01. We use Sensible Initialization (grow probability of 0.5, the maximum depth of 10 and the minimum depth of 5) for initializing the population. Like standard GE, 8-bit codons are used with wrapping size of 5.

First, we analyze the performance of the three algorithms based on the training error. From Table 2, we see that GE with AG performs better than GE with CFG for three problems (*Vargha-Delaney A* measure value>0.5), while the latter was better than the former for two problems. It seems that all three algorithms have good performances on the training set. Even though GE with CFG shows good training performance, it shows extreme overfitting on three problems (problem A, D and E), with problem A showing 94% overfitting, problem D showing 94% overfitting and the problem E showing 86% overfitting. GE with AG, on the other hand, shows no overfitting for 3 problems and very small overfitting for the rest of the two problems. Koza's GP with linear scaling also suffers from overfitting in 3 of the 5 benchmark problems with problem B (82%), problem C (4%), problem D (18%) and problem E (16%) overfitting. From the results, it is evident that when the proposed AG is used with GE, it reduces overfitting.

4. CONCLUSION

In this paper, we have shown how AG can be used to avoid certain types of invalidators. We have also shown how GE can be benefitted by avoiding selections of certain functions and binary operators that can be the cause of infinite output bound for the evolved expressions. The results and analysis show that when invalidators are avoided and interval arithmetic is implemented, GE shows significantly less overfitting than standard GE and Koza's GP on the tested problems.

Acknowledgments

This research was supported by the Science Foundation of Ireland and conducted at the BDS group, University of Limerick, Ireland.

5. REFERENCES

[1] M. R. Karim and C. Ryan. Sensitive ants are sensible ants. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, GECCO '12, pages 775–782, New York, NY, USA, 2012. ACM.

[2] M. Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In *Proceedings of the 6th European conference on Genetic programming*, EuroGP'03, pages 70–82, Berlin, Heidelberg, 2003. Springer-Verlag.

[3] S. Luke. Code growth is not caused by introns. In *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, pages 228–235. Morgan Kaufmann, 2000.