Internal and online simplification in genetic programming: an experimental comparison

Yaroslav Borcheninov, Yuri Okulovsky

Ural Federal University
Yekaterinburg, Lenina str. 51
Email: yuri.okulovsky@gmail.com

***Abstract*—Genetic programming is an evolutionary algorithm, which allows performing symbolic regression — the important task of obtaining the analytical form of a model by the data, produced by the model. One of the known problems of genetic programming is expressions' bloating that results in ineffictevely long expressions. To prevent bloating, symbolic simplification of expression is used. We introduce a new approach to simplification in genetic programming, making it a uniform part of the evolutionary process. To do that, we develop a genetic programming on the basis of transofmation rules, similarly to computer algebra systems. We compare our approach with existed solution, and prove its adequacy and effectiviness.**

***Index Terms*—genetic programming, symbolic computations, computer algebra systems**

I. Introduction

Symbolic regression is an approach to data mining, which accepts a data, generated by some model, and produces an analytic form of this model. Probably, the most known and earliest successful application of the symbolic regression is Johannes Kepler's astronomical laws, which mathematically describe observations made by Tycho Brahe. Symbolic regression is an important step in the scientific method that prescribes explaining observed data through the construction of their mathematical model. By the close examination of such mathematical model, scientists understand its internal structure and suggest hypotheses about their underlying nature.

We should stress the difference between the symbolic regression and numerical regression methods, like the linear, segmented linear or polynomial regression. In case of numerical regression the model is fixed, and only its quotients are to be found. For example, by applying polynomial regression to the data, we explicitly suggest that the model is a polynomial function. If the actual model is a trigonometric function, line sinus, the regression can be made arbitrarily accurate by choosing the appropriate polynom's degree. However, no matter how accurate it is in the sense of mean square error, the polynomial regression is still incorrect, because it will unavoidable miss the fact that the observed model is the trigonometric function. Symbolic regression allows finding the model itself, and therefore the sinus function will be recognized as sinus.

Until recently, the symbolic regression could be performed only manually, and no algorithm of symbolic regression was available. With the discovery of genetic programming technique by John Koza [Poli et al., 2008], it becomes possible to automate symbolic regression. Now automated symbolic regression is widely used in natural sciences [Schmidt and Lipson, 2009], robotics [Robertson and Dumont, 2002], economics [Koza, 1994], medicine [Zhang and Wong, 2008], etc.

The algorithm processes versions about the actual data's model. These versions are expressions, encoded as trees and stored in the *pool*. Initially, these expressions are random. Then, the algorithm alters expressions with the following procedures.

- *Mutation*. The randomly chosen expression is changed by a replacement of a node.
- *Crossover*. Two randomly chosen expressions exchange subtrees.
- After all the mutations and crossovers are performed, the resulting expressions' set is subjected to the *selection*, which evaluates how each expression fits the experimental data. The least valuable expressions are then removed from the population.

With the time, expressions become better until the satisfiable solution is found.

The known problem of genetic programming is expressions' bloating, which means that expressions become ineffectively long. For example, expression $(x+1)^2-(x-1)^2-3x$ is bloated, because it actually equals to x and should be replaced by x in the pool. One result of bloating is unacceptable form of the algorithm output. It can be resolved with the simplification of the algorithm's result. However, bloating also hampers the algorithm's work by increasing the expression length and therefore the time required to compute them, and also by leading the algorithm along the blind alley. It can be resolved with the online simplification [Zhang et al., 2006], [Kinzett et al., 2008], when all expressions in the pool are simplified with some frequency. There exist other approaches ([Poli et al., 2008], [Mori et al., 2009]), however online simplification is considered to be more effective.

We argue that online simplification is too rough. Simplifying the expression inevitably leads to the elimination of potential growing points. For example, while approximating the function $(x+1)y^2$, the intermediate solution $(1+1)y^{1+1}$ can be found. This solution will be simplified to $2y^2$, which requires at least two mutations to become a correct answer,

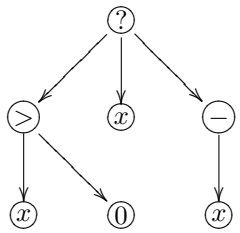

Fig. 1. The tree representation of the function $f(x) = |x|$.

e.g. $2y^2 \Rightarrow xy^2 \Rightarrow (x+1)y^2$. The initial solution $(1+1)y^{1+1}$ requires only one mutation $(1+1)y^{1+1} \Rightarrow (x+1)y^{1+1}$. Hence, the simplification hampers the evolution in this case. On other hand, the partial simplification $(1+1)y^{1+1} \Rightarrow (1+1)y^2$ does not produce such effect for the function $(x+1)y^2$, but does so for $2y^{x+1}$. Therefore, the question of where to apply the simplification depends on the problem specification, on the particular found expression, etc. In other words, the simplification can alter evolution of expressions in the same way the mutation and crossover do.

In [Borcheninov and Okulovsky, 2011], we introduce an approach of integration of simplification into genetic programming as uniform part. We call our approach internal simplification genetic programming (ISGP), as opposed to online simplification genetic programming (OSGP). The key aim of this paper is to measure the advantage of ISGP in comparison with OSGP.

Simplification is based on the rules, which describe ways of correct expressions' transformation. Since we use the simplification inside the algorithm, we must base our algorithm on the rules. In section 1, we show how to implement OSGP and ISGP an instances of more general rule-based algorithm. In section 2, we describe experiments to compare internal and online simplification.

II. Algorithm essentials

A. Expressions, trees and rules

An expression is represented as a tree of *nodes*. The example of such tree that encodes the function $f(x) = |x|$ is shown in the Fig. 1. Three types of nodes are considered: *constants*, *variables* and *operators*. In Fig. 1, node $\textcircled{x}$ is a variable node, $\textcircled{0}$ is a constant node. The remaining nodes are operators: addition $\textcircled{+}$, comparison $\textcircled{>}$ and ternary logical operator $\textcircled{?}$, defined as follows

$$?(x,y,z) = \begin{cases} y, & \text{if} \quad x \\ z, & \text{if} \quad \neg x \end{cases} .$$

Each node has a *return type*, which is an arbitrary C# type. Different return types can be used in one expression. For example, in Fig. 1, all nodes have `double` return type, except for the node $\textcircled{>}$ that has the `bool` return type.

We define numerous rules to transform these expressions. Some of these rules are universal, and can be applied to the tree regardless of data types or operations that are used in it.

(I-Re)	**select**	?A(?B)
	where	A.Type=B.Type
	mod	A→B
(I-Cr)	**select**	?A,?B
	where	A.Type=B.Type
	produce	A→B; **ret** A.Root

In I-Re rule, the **select** clause specifies the nodes that will be selected as a tuples (A, B), and then processed by the rule. The notion `?A(?B)` specifies that A is an arbitrary descendant of root (i.e., and arbitrary node in the tree), and B is an arbitrary descendant of A. Then, selected tuples are subjected to selection according to **where** clause. In I-Re, we accept only the tuples (A, B) such that they returning types coincides. To selected tuples, we can apply **mod** clause. In the case of I-Re, it replaces A with B. The tree remains correct, because of the selection in **where** clause. In I-Cr rule, the select clause `?A,?B` denotes that the rule accepts two trees, and selects an arbitrary node from each of them. Therefore, this rule is binary, while I-Re rule is unary. Then we demand the equality of their returning types, and finally replace A with B and return the root of A as an output. Using **produce** clause means that we specify directly the output of the rule. It is necessary, because binary rules accept two trees, and it is not clear which one of them should be the output.

Most of the rules, however, are not universal. With each data type T, the following rules are associated

(I-Co)	**select**	?A
	where	A.Type=T
	mod	A→new Const(v)
(I-Va)	**select**	?A
	where	A.Type=T
	mod	A→new Var(i)
(I-Tu)	**select**	?A
	where	A is Const
	mod	A→new Const(R(A.Value))

I-Co rule replaces the node with the return type T with the constant of the same type. Here v is a randomly selected value of the constant. I-Va rule replaces the node with the return type T with the variable. The argument i is a number of the variable in the argument array of the expression. Instances of I-Va rule have to be created for each variable of type T.We can also define tunning rules that adjust the constants. For Boolean and integer data types, such rules seem to be redundant, because they are just instances of I-Co. However, for floating point data type, rule I-Tu can be written. Here R is a random function $R(x)$ that returns a random number from $[x(1-c), x(1+c)]$. I-Tu rule allows changing the constant value gradually, near its initial value, and therefore differs from I-Co rule that does not take the previous value into account.

Some rules are even more specific, and are associated not with data types, but with the operations domain. The domain is a set of operations that are commonly used together and are bound by some mathematical laws. Examples are arithmetic

domain (addition, multiplication, etc.); trigonometric domain (sinus, cosinus, etc.); logical domain (conjunction, negation, etc.).

For each operation, we need an introduction rule. Two approaches to operation's introduction are possible.

	select	?A
(G-In)	**where**	A.Type = double
	mod	A→new Mult(A,new Const(1))

	select	?A
(G-In*)	**where**	A.Type = double
	mod	A→new Mult(A,new Const(v))

G-In rules selects a node with floating point return type, and replaces it with a new multiplication operation. G-In rule differs from all the rules above, because it does not change the function, encoded by the expression. It only inflates the expression and adds potential growing point in it. Of course, we could combine G-In rule with I-Co, therefore obtaining G-In*. However, it is not convenient. Suppose our task is to transform x into $2x$. With the modified G-In rule, we need the double luck to do that: we need to guess correctly both the operation and the constant. Wrong choice of constant may lead to significant decrease of the expression correctness, and therefore the expression will be removed, without a chance to adjust the constant. Original G-In rule does not affect correctness, and therefore modified rule can remain in the pool for a long time, so different mutations by I-Co rule can occur in the future and a right constant has more chances to be chosen.

For each operation, we also define simplification rules, for example transforming a multiplication of two constants into a constants with their multiplication, or transforming the multiplication of any node and zero into zero. We call such simplifying rules S-rules. They are known from computer algebra systems, so we will not study them deeply. Some rules are developed not for a single operations, but for several operations in the domain. The example is distributivity of addition and multiplication, which is G-rule for transformation $a \cdot (b + c) \to ac + cb$ and S-rule for reverse transformation.

Aside from simplification rules, we can also define a crossover rules for domain, with a very natural meaning:

	select	A,B
(I-CA)	**where**	A.Type=double and B.Type=double
	produce	new Div(new Plus(A,B),2)

The absence of quotation marks before A and B means that they are not descendants of the root, but the roots themselves. Crossover I-CA is reasonable: if two expressions fit the task, their halfsum may fit even better.

B. Implementation of genetic programming algorithms

To define a concrete algorithm in the genetic programming algorithms' family, we need to specify the operations, mentioned in the Intoduction: mulation, crossover and evaluation. We define mutation and crossover operations on basis of rules collection. The algorithm has two sets of rules: the set of unary rules for mutation, and the set of binary rules for crossover. In order to perform mutation, algorithm randomly selects expressions for mutation. Then, for each expression, we randomly select a rule, and perform it to obtain a mutated expression. Correspondingly, in order to cross two randomly selected expressions, the algorithm chooses a binary rule from the collection and performs it.

From the start of observations it becomes clear that different rules must have different probability to be applied. Each rule has multiple tags that describe the place of the rule in our classification. Then we assign to each tag its weight, and calculate the weight of the rule as the product of associated tags' weights. The greater the rule's weight is, the more the probability of rule's application is.

The most important tags are Inductive and Simplification tags. Inductive tags marks all the rules, which enlarge the expressions (G-rules from section 1.1), or changes the function the expression encodes (I-rules). Simplification rules make the expression shorter (S-rules). The ratio of Inductive and Simplification tags κ is the first important parameter of our algorithm.

The evaluation of the expression is performed by calculating several metrics and obtaining their weighted total. The fitness metric describe, how good the found expression g fits given data $(x_{1,j}, \ldots, x_{n,j}, y_j)$, and is calculated as

$$\mu_f(g) = \left(1 + \sum_{i=1}^{n} (g(x_{i,1}, \ldots, x_{i,m}) - y_i)^2\right)^{-1}.$$

Taking the reciprocal value is important, because it allows bounding the value of ρ, and provides correspondence between a higher value of ρ and a better expression. The length metric μ_l is a reciprocal to the count of operations in g. Valuation of an expression is determined as a weighted total $e(g) = w_f \mu_f(g) + w_l \mu_l(g)$. The ratio between the fitness metric and the length metric $\lambda = w_l / w_f$ is the second important parameter of our algorithm.

To perform online simplification, we modify the described algorithm. First, only I- and G-rules are allowed to be used in the algorithm. Second, the weight of length metric is set to zero, because algorithm does not have necessary means to decrease the expression's length. Finally, after each ξ iterations, we apply a simplification algorithm to each expression in the pool. Namely, we apply S-rules to expression until it is possible, and return the resulting expression in the pool. Online simplification algorithm has only one parameter ξ.

III. Experimental results

We conducted the following experiments to compare online and internal simplification in genetic programming. At first, we prepared test sets to run the algorithm on. Then, we found the optimal parameters of both algorithms to fetch best performances. Finally, we compared the performance of both algorithms.

In order to achieve a reasonable ratio between the representativeness of experiments and the time of computations, we followed the guidelines below. We limited the domain

of expressions by algebraic expressions that contain addition, subtraction, multiplication, division and power operations and integer constants. The reason is limiting the amount of parameters of algorithms. Two parameters are unavoidable: length/evaluation metrics ratio λ and inductive/calculation κ tags ratio. Introducing floating point constants demands us to use tunning rule (I-Tu). Our observations showed that intensity of this rule should be much greater than others', in order to find the appropriate values of constants. This adds one more parameter. Correspondingly, the introduction of trigonometric functions leads to various expressions like $\sin(\sin(\cos(\ldots)))$, and therefore these operations need to have their own tag with reduced value. Therefore, widening the domain requires increase of parameters. Since we needed to obtain the optimal parameters in order to compare approaches, we decided to limit the domain.

On the other hand, we made a high demand to the algorithm's outcome. The algorithm was provided with a very strict amount of data points: 10 for unary function and 100 for binary. The amount of iteration was limited by 10000, which takes about 15 minutes to compute. We also demanded the algorithm to find the exact function, used to generate the data, not its good approximation. The function may be presented as different expressions, however. It is a very strict requirement: sometimes the algorithm found the solution that was very close to data (root mean error is about 2-3%), and nevertheless, we neglected such solution and demanded the exact solution to be found. Summarizing, we can say that algorithm had to find an exact function with a limited data set in a short time. We believed that the complexity of this task compensates the domain narrowness.

To build the test set we made a rundown over different expressions, tested them with our algorithm and therefore obtained a knowledge about "complexity" of these expressions in terms of the algorithm. The considered parameters of expressions was the number of expression's arguments; the number of operations, used in the expression; the level of white noise, applied to data. At first, we builded a random tree with desired count of operations and tested, if the expression truly depends on all its arguments. Then, we formed test set as an array

$$\begin{array}{llll} x_{1,1}, & \ldots, & x_{n,1}, & y_1 \\ x_{1,2}, & \ldots, & x_{n,2}, & y_2 \\ & \ldots & & \\ x_{1,m}, & \ldots, & x_{n,m}, & y_m \end{array}$$

where

$$\{(x_{1,j}, \ldots, x_{n,j} \mid j \in 1, \ldots, m\} = K^n,$$

the set K is $\{0, 0.1, \ldots, 1\}$, $y_j = f(x_{1,j}, \ldots, x_{n,j}) \cdot (1 - p + 2p\alpha)$, p is white noise level and α is a uniform random number between 0 and 1. If f cannot be calculated for some j, we dropped the expression and searched again. On each data set, we run the algorithm several times and measure the average success rate. If the algorithm had accidentally found the form of expression containing least operation that planned, the data set was also considered invalid and was excluded from experimental result.

Variable count = 1

	p=0	p=0.01	p=0.02	p=0.05
c=2	96.67	95.38	90.87	95.62
c=3	29.47	30.63	33.93	25.68
c=4	11.67	0	0	0

Variable count = 2

	p=0	p=0.01	p=0.02	p=0.05
c=2	98.62	97.39	98.24	98.67
c=3	25.58	39.46	31.19	29.02
c=4	2.4	2.45	3.65	5.74

TABLE I
SUCCESS RATE, IN PERCENTS, FOR DIFFERENT VARIABLE COUNT, COUNT OF OPERATIONS c AND WHITE NOISE LEVEL p

Function	Success rate, %	Description
$(x^2)(x+4)$	80	Simple polinom
$\frac{y}{2}(x+2)$	80	A simple polynom with two variables
x^{x-y}	80	A simple non-rational function
$x^2 - (y+3)$	50	Intermediate polynom with two variables
$\frac{x}{2(x-3)}$	50	Intermediate rational function
$(3x)^{2y}$	50	Intermediate non-rational function
$2xy - y - 2$	20	Hard polynom
$\frac{x(x+4)}{x-5}$	20	Hard rational function
$\left(\frac{x}{4}\right)^{4x}$	20	Hard rational function

TABLE II
FUNCTIONS, SELECTED TO TEST SET

For each set of parameters, we run 50 successful data set, and each data set was processed by the algorithm 10 times. Obtained result are presented in Table III. The overall tendency is clear. The complexity is determined mostly by count of operations, then by the level of white noise. Additional variables seem to reduce the complexity, probably because of widening data set from 10 to 100 samples. We can also conclude that the algorithm is functional, even though initial parameters could be far from optimal.

We selected 9 expressions as test set for the OSGP and ISGP comparison. Selected expressions are listed in Table III. We did not selected expressions with 0% success rate, because it this case the difference between hard and impossible is not clear. For the same reason, we omited expressions with 100% success.

We ran ISGP algorithm with different length/fitness metrics ratio λ and calculation/induction tags ratio κ and obtained the resuls, presented in Table III. We see that the algorithm is in tote stable, and its success rate varies in range 60–70%. It is unlikely to find some local maxima outside the considered parameters' range. Parameters λ and κ by definition are greater that zero. When $\lambda = 0$ or $\kappa = 0$, the simplification is simply not performed, and expressions bloat rapidly, blocking the algorithm. When $\lambda > 1$ or $\kappa > 1$, the simplification is too strong: by out observation, no expressions of length more than 3 can be produced. Therefore we believe that the best success

κ	0.01	0.02	0.05	0.1	0.2	0.4
$\lambda = 0.01$	66.67	71.67	64.44	63.89	63.33	56.67
$\lambda = 0.02$	67.22	68.33	66.11	62.22	65	59.44
$\lambda = 0.05$	66.67	66.67	67.22	67.78	64.44	58.89
$\lambda = 0.1$	66.11	62.22	63.89	70	60	56.11
$\lambda = 0.2$	63.89	66.67	65.56	64.44	62.22	53.89
$\lambda = 0.4$	68.89	66.67	65.56	66.11	65.56	62.78
$\lambda = 0.8$	66.67	63.33	65.56	65	67.22	61.67

κ	0.02	0.04	0.06	0.08	0.1
$\lambda = 0.01$	68.33	73.89	68.89	67.22	66.11
$\lambda = 0.0333$	66.11	63.89	62.22	63.33	68.89
$\lambda = 0.0666$	66.67	63.89	61.67	62.22	64.44
$\lambda = 0.1$	73.33	65	63.33	70	63.33

TABLE III
SUCCESS RATES, IN PERCENTS, OF OSGP WITH VARIOUS VALUES OF THE PARAMETERS κ AND λ. THE LOWER TABLE GIVES A CLOSER LOOK TO THE AREA, WHERE LOCAL MAXIMA SEEM TO BE.

$\xi = 10$	$\xi = 20$	$\xi = 30$	$\xi = 40$	$\xi = 50$	$\xi = 60$
36.11	55	70.56	70	71.67	68.89

$\xi = 70$	$\xi = 80$	$\xi = 90$	$\xi = 100$	$\xi = 160$
68.89	65	70	68.33	65.56

TABLE IV
SUCCESS RATES, IN PERCENTS, OF ISGP WITH VARIOUS VALUES OF THE PARAMETER ξ.

rate of our algorithm is about 70% on our test set.

For OSGP, we need to determine the count of iterations between simplifications, ξ. The results of OSGP for different ξ are presented in Table III. Again, it is unlikely that the optimal value of ξ is greater than 160, because such rare simplification is hardly noticeable. On other hand, when the simplification is performed too often ($\xi < 5$), long expressions are almost never appear in the pool.

In table III we present the success rate of best algorithm's variants on test set. We can conclude, that the algorithms are very close in terms of performance. It is also obvious that accurate choise of parameters is important, and improves effectiveness significantly, at least for some functions.

IV. CONCLUSION

The research, presented in this article, proves the internal simplification genetic programming to be an operational technique that prevents bloating of expressions and provides effective symbolic regression. The only way to implement ISGP is to found genetic programming on the basis of expressions' transformation rules, as it was described in section 1.

The performance comparison states that ISGP is not worse than existed online simplification approach. ISGP also open a road for further research in the following areas. At first, we plan to explore the more presice devision of rules into groups, and finding the appropriate tags for such devision. This task can be considered even for the algebraic domain: for example, we could consider different tags for I- and G-rules. For the greater domains, this task is even more important, because additional tags emerge anyway.

Function	ISGP, $\kappa = 0.5$, $\lambda = 0.1$	ISGP, $\kappa = 0.04$, $\lambda = 0.01$	OSGP, $\xi = 50$
$(x^2)(x+4)$	80	100	100
$\frac{y}{2}(x+2)$	80	100	100
x^{x-y}	80	100	100
$(3x)^{2y}$	50	100	100
$x^2 - (y+3)$	50	95	100
$\frac{x}{2(x-3)}$	50	95	95
$\left(\frac{x}{4}\right)^{4x}$	20	25	25
$2xy - y - 2$	20	50	5
$\frac{x(x+4)}{x-5}$	20	0	20

TABLE V
SUCESS RATES, IN PERCENTS, OF ALGORITHMS ON TEST SET. THE SECOND COLUMN REPRESENTS THE INITIAL SUCCESS RATES, GENERATED WHEN BUILDING TEST SET. THE THIRS AND FOURTH COLUMNS ARE BEST RESULTS OF OSGP AND ISGP, CORRESPONDINGLY.

The more intriguing branch of research is adjusting the tag's weights during the algorithm's work. The tentativ observations show that such adjusting can sometimes drive the algorithm out of the local minimun by speeding up induction, or narrow the search around the best expression by increasing of the fitness metric weight.

We also plan to develop a distributed version of our OSGP implementation, and test it in real-world problems, mostly from robotics field.

ACKNOWLEDGMENTS.

The work is supported by the program of President of Russian Federation MK-844.2011.1.

REFERENCES

[Borcheninov and Okulovsky, 2011] Borcheninov, Y. V. and Okulovsky, Y. S. (2011). Genetic programming with embedded features of symbolic computations. In *KDIR 2011 — Proceedings of the International Conferenceon Knowledge Discovery and Information Retrieval*.

[Kinzett et al., 2008] Kinzett, D., Johnston, M., and Zhang, M. (2008). Numerical simplification for bloat control and analysis of building blocks in genetic programming. *Evolutionary Intelligence*, 4.

[Koza, 1994] Koza, J. R. (1994). Genetic programming for economic modeling. In *Intelligent Systems for Finance and Business*.

[Mori et al., 2009] Mori, N., McKay, B., Hoai, N. X., Essam, D., and Takeuchi, S. (2009). A new method for simplifying algebraic expressions in genetic programming called equivalent decision simplification. *Journal of Advanced Computational Intelligence and Intelligent Informatics*, 13(14):237–238.

[Poli et al., 2008] Poli, R., Langdon, W. B., McPhee, N. F., and Koza, J. R. (2008). *A Field Guide to Genetic Programming*.

[Robertson and Dumont, 2002] Robertson, A. P. and Dumont, C. (2002). Design of robot calibration models using genetic programming. In Mayorga, R. V. and Rios, A. S.-D. L., editors, *Proceedings of the Third International Symposium on Rob. and Autom.*, volume 3, pages 449–454.

[Schmidt and Lipson, 2009] Schmidt, M. and Lipson, H. (2009). Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85.

[Zhang and Wong, 2008] Zhang, M. and Wong, P. (2008). Genetic programming for medical classification: a program simplification approach. *Genetic Programming and Evolvable Machines*, 9(2):229–255.

[Zhang et al., 2006] Zhang, M., Wong, P., and Qian, D. (2006). Online program simplification in genetic programming. *Simulated Evolution and Learning - SEAL*, pages 592–600.