Hybrid single node genetic programming for symbolic regression

Kubalìk, Jiřì; Alibekov, Eduard; Žegklitz, Jan; Babuska, R.

DOI
10.1007/978-3-662-53525-7_4

Publication date
2016

Document Version
Accepted author manuscript

Published in
Transactions on Computational Collective Intelligence XXIV

The final publication is available at link.springer.com

DOI: 10.1007/978-3-662-53525-7_4

Hybrid Single Node Genetic Programming for Symbolic Regression

Jiří Kubalík[1], Eduard Alibekov[1,2], Jan Žegklitz[1,2], and Robert Babuška[1,3]

[1] Czech Institute of Informatics, Robotics, and Cybernetics,
CTU in Prague, Prague, Czech Republic
{kubalik,babuska}@ciirc.cvut.cz

[2] Department of Cybernetics, Faculty of Electrical Engineering,
CTU in Prague, Prague, Czech Republic

[3] Delft Center for Systems and Control,
Delft University of Technology, Delft, the Netherlands

Abstract. This paper presents a first step of our research on designing an effective and efficient GP-based method for symbolic regression. First, we propose three extensions of the standard Single Node GP, namely (1) a selection strategy for choosing nodes to be mutated based on depth and performance of the nodes, (2) operators for placing a compact version of the best-performing graph to the beginning and to the end of the population, respectively, and (3) a local search strategy with multiple mutations applied in each iteration. All the proposed modifications have been experimentally evaluated on five symbolic regression benchmarks and compared with standard GP and SNGP. The achieved results are promising showing the potential of the proposed modifications to improve the performance of the SNGP algorithm. We then propose two variants of hybrid SNGP utilizing a linear regression technique, LASSO, to improve its performance. The proposed algorithms have been compared to the state-of-the-art symbolic regression methods that also make use of the linear regression techniques on four real-world benchmarks. The results show the hybrid SNGP algorithms are at least competitive with or better than the compared methods.

Keywords: Genetic Programming, Single Node Genetic Programming, Symbolic Regression

1 Introduction

This paper presents a first step of our research on genetic programming (GP) for the symbolic regression problem. The ultimate goal of our project is to design an effective and efficient GP-based method for solving dynamic symbolic regression problems where the target function evolves in time. Symbolic regression (SR) is a type of regression analysis that searches the space of mathematical expressions

to find the model that best fits a given dataset, both in terms of accuracy and simplicity[4].

Genetic programming belongs to effective and efficient methods for solving the SR problem. Besides the standard Koza's tree-based GP [12], many other variants have been proposed. They include, for instance, Grammatical Evolution (GE) [20] which evolves programs whose syntax is defined by a user-specified grammar (usually a grammar in Backus-Naur form). Gene Expression Programming (GEP) [4] is another GP variant successful in solving the SR problems. Similarly to GE it evolves linear chromosomes that are expressed as tree structures through a genotype-phenotype mapping. A graph-based Cartesian GP (CGP) [18], is a GP technique that uses a very simple integer based genetic representation of a program in the form of a directed graph. In its classic form, CGP uses a variant of a simple algorithm called $(1 + \lambda)$-Evolution Strategy with a point mutation variation operator. When searching the space of candidate solutions, CGP makes use of so called *neutral mutations*, meaning that a move to the new state is accepted if it does not worsen the quality of the current solution. This allows an introduction of new pieces of genetic code that can be plugged into the functional code later on and allows for traversing plateaus of the fitness landscape.

A Single Node GP (SNGP) [9], [10] is a rather new graph-based GP system that evolves a population of individuals, each consisting of a single program node. Similarly to CGP, the evolution is carried out via a hill-climbing mechanism using a single reversible mutation operator. The first experiments with SNGP were very promising as they showed that SNGP significantly outperforms the standard GP on various problems including the SR problem. In this work we take the standard SNGP as the baseline approach and propose several modifications to further improve its performance.

The goals of this work are twofold. The first goal is to verify performance of the vanilla SNGP compared to the standard GP on various SR benchmarks and to investigate the impact of the following three design aspects of the SNGP algorithm:

- a strategy to select the nodes to be mutated,
- a strategy according to which the nodes of the best-performing expression are treated in the population,
- and a type of the search strategy used to guide the optimization process.

The second goal is to propose a hybrid variant of SNGP which incorporates the LASSO regression technique for creating linear-in-parameters nonlinear models. We compare its performance with other state-of-the-art symbolic regression methods which also make use of linear regression techniques.

The paper is organized as follows. Section 2 describes the SNGP algorithm. In Section 3, three modifications of the SNGP algorithm are proposed. Experimental evaluation of the modified SNGP and its comparison to the standard SNGP and standard Koza's GP is presented in Section 4. Section 5 describes two

[4] https://en.wikipedia.org/wiki/Symbolic_regression

variants of the SNGP utilizing the linear regression technique, LASSO, to improve its performance. The two versions of SNGP with LASSO are compared to other symbolic regression methods making use of the linear regression techniques in Section 6. Finally, Section 7 concludes the paper and proposes directions for the further research on this topic.

2 Single Node Genetic Programming

2.1 Representation

The Single Node Genetic Programming is a GP system that evolves a population of individuals, each consisting of a single program node. The node can be either terminal, i.e. a constant or a variable node, or a function from a set of functions defined for the problem at hand. Importantly, individuals are not isolated in the population, they are interlinked in a graph structure similar to that of CGP, with population members acting as operands of other members [9].

Formally, a SNGP population is a set of N individuals $M = \{m_0, m_1, \ldots, m_{N-1}\}$, with each individual m_i being a single node represented by the tuple $m_i = \langle u_i, f_i, Succ_i, Pred_i, O_i \rangle$, where

- $u_i \in T \cup F$ is either an element chosen from a function set F or a terminal set T defined for the problem,
- f_i is the fitness of the individual,
- $Succ_i$ is a set of successors of this node, i.e. the nodes whose output serves as the input to the node,
- $Pred_i$ is a set of predecessors of this node, i.e. the nodes that use this individual as an operand,
- O_i is a vector of outputs produced by this node.

Typically, the population is partitioned so that the first N_{term} nodes, at positions 0 to $N_{term} - 1$, are terminals (variables and constants in case of the SR problem), followed by function nodes. Importantly, a function node at position i can use as its successor (i.e. the operand) any node that is positioned lower down in the population relative to the node i. This means that for each $s \in Succ_i$ we have $0 \leq s < i$ [9]. Similarly, predecessors of individual i must occupy higher positions in the population, i.e. for each $p \in Pred_i$ we have $i < p < N$. Note that each function node is in fact a root of a *direct acyclic graph* that can be constructed by recursively traversing through successors until the leaf terminal nodes.

2.2 Evolutionary model

In [9], a single evolutionary operator called *successor mutate* (*smut*) has been proposed. It picks one individual of the population at random and then one of its successors is replaced by a reference to another individual of the population making sure that the constraint imposed on the successors is satisfied. Predecessor

lists of all affected individuals are updated accordingly. Moreover, all individuals affected by this action must be reevaluated as well. For more details refer to [9].

The evolution is carried out via a hill-climbing mechanism using a *smut* operator and an acceptance rule, which can have various forms. In [9], it was based on fitness measurements across the whole population, rather than on single individuals. This means that once the population has been changed by a single application of the *smut* operator and all affected individuals have been re-evaluated, the new population is accepted if and only if the sum of the fitness values of all individuals in the population is no worse than the sum of fitness values before the mutation. Otherwise, the modifications made by the mutation are reversed. In [10] the acceptance rule is based only on the best fitness in the population. The latter acceptance rule will be used in this work as well. The reason for this choice is explained in Section 3.4.

3 Proposed Modifications

In this section, the following three modifications of the SNGP algorithm will be proposed:

1. A selection strategy for choosing nodes to be mutated based on depth and performance of nodes.
2. Operators for placing a compact version of the tree rooted in the best performing node to the beginning and to the end of the population, respectively.
3. A local search strategy with multiple mutations applied in each iteration.

In the following text, the term "best tree" is used to denote the tree rooted in the best performing node.

3.1 Depthwise Selection Strategy

The first modification focuses on the strategy for selecting the nodes to be mutated. In the standard SNGP, the node to be mutated is chosen at random. This means that all function nodes have the same probability of selection irrespectively of (1) how well they are performing and (2) how well the trees of which they are a part are performing. This is not in line with the evolutionary paradigm where the well fit individuals should have higher chance to take part in the process of an evolution of the population.

One way to narrow this situation is to select nodes according to their fitness. However, this would prefer just the root nodes of trees with high fitness while neglecting the nodes at the deeper levels of such well-performing trees which themselves have rather poor fitness. In fact, imposing high selection pressure on the root nodes might be counter-productive in the end as the mutations applied on the root nodes are less likely to bring an improvement than mutations applied on the deeper structures of the trees.

We propose a selection strategy that takes into account the quality of the mutated trees, so that better performing trees are preferred, as well as the depth

of the mutated nodes so that deeper nodes of the trees are preferred to the shallow ones. The selection procedure has four steps:

1. A function node n is chosen at random.
2. A tree t with the best fitness out of all trees that use the node n is chosen.
3. All nodes of the tree t are collected in a set S. Each node is assigned a score equal to its depth in the tree t.
4. One node is chosen from the set S using a binary tournament selection considering the score values in the higher the better manner.

3.2 Organization of the Population

The second modification aims at improving the exploration capabilities of the SNGP algorithm. Two operators for placing a compact version of the best performing graph to the beginning and to the end of the population, respectively, are proposed.

Move left operator. Let us first describe the motivation for and the realization of the operator that places the compact version of the best graph to the beginning of the population. The motivation for this operator, denoted as $moveLeft$ operator, is that well-performing nodes (and the whole graph structure rooted in this node) can represent a suitable building block for constructing even better trees when used as a successor of other nodes in the population. Since the chance of any node of being selected as a successor is higher if the node is more to the left in the population, it might be beneficial to store the well-performing graphs at lower positions in the population. Thus, the operator takes the best graph, G_{best}, and places it in a *compact form* to the very beginning of the population. By the compact form of a graph G we mean a sequence of nodes representing the whole G such that it contains only nodes involved in G. The $moveLeft$ operator works as follows:

1. Extract nodes of the graph G_{best} rooted in the best-performing node and put the nodes into a compact ordered list L.
2. Set all successor and predecessor links of nodes within L so that L represents the same graph as the original graph G_{best}.
3. Place L to the beginning of the population, i.e. the first node of L being at the first function node position in the population.
4. Update the successor links of nodes of the original graph G_{best} so that it retains the same functionality as it had before the action.
   It must be made sure that all nodes of the original G_{best} have properly set their successors. If for example some successor of a node of the original G_{best} gets modified (i.e. the successor falls into the portion of the population newly occupied by the compact form of the G_{best}), then the successor reference is updated accordingly. In Fig. 1, this is for example the case of the second successor of the node at position 8, which originally pointed to the node number 4 and after the $moveLeft$ operation has been redirected to the node number 2.

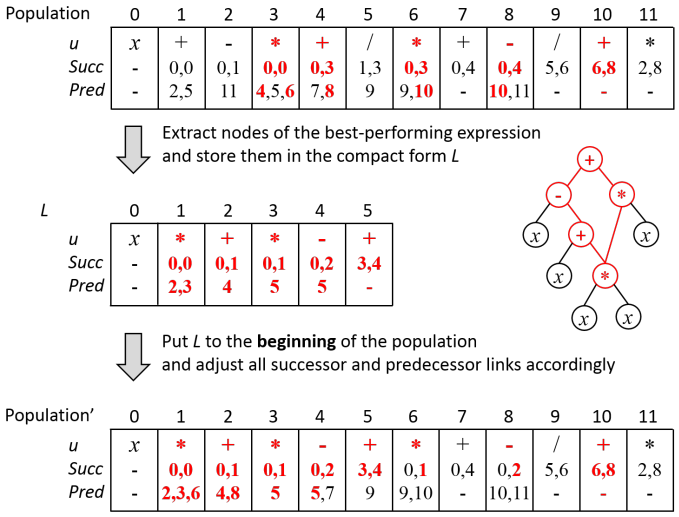

Fig. 1. Illustration of the *moveLeft* operator. Function nodes involved in the original and compact form of the best graph are shown in red. After the application of the *moveLeft* operator the population contains two occurrences of the best graph, the one represented by a sequence of nodes [0, 1, 2, 6, 8, 10] and the one represented by nodes [0, 1, 2, 3, 4, 5].

5. Update the predecessor lists of nodes in the compact form of G_{best}) in order to reestablish links to other nodes in the population that use the nodes as successors.
   In the example in Fig. 1 this is the case of the predecessor number 7 of node number 4 and the predecessor number 9 of node number 5, respectively.

Note that after the application of the $moveLeft$ operator the population contains two versions of the G_{best}, the original one and the compact one, see the example in Fig. 1.

Move right operator. Similarly, an operator that places the compact version of the best graph G_{best} to the end of the population is proposed. The motivation for this operator, denoted as *moveRight* operator, is that a performance of some well-performing graphs can more likely be improved by mutations applied to the nodes on deeper levels of the graph than by mutations applied to the root node or shallow nodes of the graph. In order to increase the number of possible structural changes to the deeper nodes of the best graph the compact version of the graph is placed to the end of the population. The working scenario for the operator is similar to the one of the *moveLeft* operator, see Fig. 2. Note that the application of the *moveRight* operator might result in the final population that contains just a single occurrence of the G_{best}. This might happen when the nodes of the original G_{best} fall into the area of its compact form.

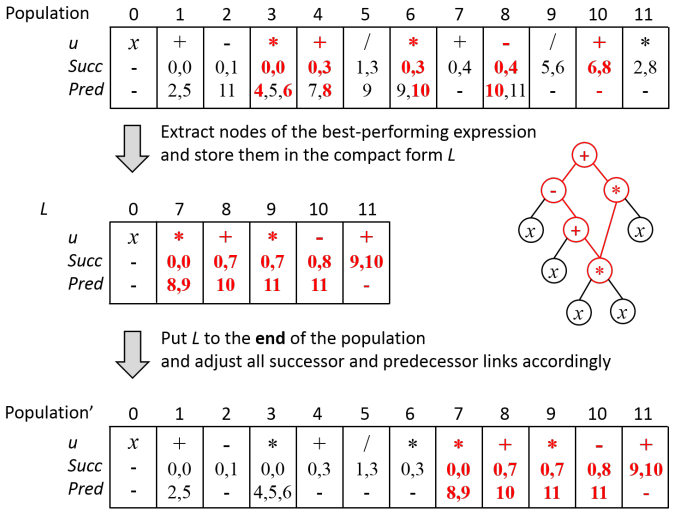

Fig. 2. Illustration of the *moveRight* operator. Function nodes involved in the original and compact version of the best graph are shown in red. In this example, the final population contains just one occurrence of the best graph represented by a sequence of nodes [0, 7, 8, 9, 10, 11].

3.3 Local Search Strategy

The last modification of the standard SNGP algorithm consists in allowing multiple mutation in a single iteration of the local search procedure. The idea behind this modification is rather straightforward. During the course of the optimization process the population might converge to the local optimum state where it is hard to find further improvement by just one application of the smut operator. With multiple mutations applied in each iteration, the probability of getting stuck in such local optimal should be reduced. In this work, a parameter $upToN$ specifying the maximum number of mutation applications is used. Thus, if the parameter is set for example to 5, a randomly chosen number from interval $\langle 1, 5 \rangle$ of mutations are applied to the population in each iteration.

3.4 Outline of Modified SNGP Algorithm

This section presents an outline of the generic SNGP algorithm with possible utilization of the proposed modifications, see Fig. 3. In each generation, k mutations are applied to nodes of the population, see steps 8-10. In case of the standard SNGP just a single mutation is applied in each generation. After all k mutations have been applied, the nodes affected by this action gets reevaluated. If the best fitness of the modified population is not worse than the current best-so-far fitness than the modified population becomes the current population

```
1  Initialize population of nodes, P
2  evaluate P
3  bestSoFar ← best of P
4  i ← 0                    // number of generations
5  do
6      P' ← P               // work with a copy of P
7      Choose the no. of mutations, k ∈ (1, upToN),
       to be applied to nodes from P'
8      for(n = 1 ... k)
9          Choose the node to be mutated, N
10         P' ← Apply mutation to node N of P'
11     evaluate P'
12     currBest ← best of P'
13     if(currBest is not worse than bestSoFar)
14         bestSoFar ← currBest
15         P ← P'           // update the population
16         If applicable, apply either moveLeft
           or moveRight operator to P
17     i ← i + 1
18 while (i ≤ maxGeneration)
19 return bestSoFar
```

Fig. 3. Outline of the modified SNGP algorithm.

for the next generation, see step 15. Here, the fitness of each individual is calculated as the **sum of absolute errors** (SAE) generated by the individual over all training samples. In step 16, the operators moving the best tree to the beginning or to the end of the population are applied to the population, if applicable. Then the fitness evaluation counter is incremented and if there are still some fitness evaluations left the next generation is carried out. Once the maximal number of fitness evaluations is used the best node (and its tree) of the population is returned.

In this work, we use the acceptance criterion, step 13, working with the best fitness in the population, not the average fitness of the population. The reason is that when the *moveLeft* and *moveRight* operators are used, they might significantly change the average fitness of the population while the best fitness stays intact.

4 Experiments with Modified SNGP

This section presents experiments carried out with standard GP, standard SNGP and SNGP with the proposed modifications.

4.1 Artificial Benchmarks

The algorithms have been tested on five symbolic regression benchmarks

- $f_1(x) = 4x^4 - 3x^3 + 2x^2 - x$,
  32 training samples equidistantly sampled from $\langle 0, 1.0)$,
- $f_2(x) = x^6 - 2x^4 + x^2$,
  100 training samples equidistantly sampled from $\langle -1.0, 1.0)$
- $f_3(x) = x^6 - 2.6x^4 + 1.7x^2$,
  100 training samples equidistantly sampled from $\langle -1.0, 1.0)$
- $f_4(x) = x^6 - 2.6x^4 + 1.7x^2 - x$,
  100 training samples equidistantly sampled from $\langle -1.4, 1.4)$
- $f_5(x_1, x_2) = \frac{(x_1-3)^4+(x_2-3)^3-(x_2-3)}{(x_2-2)^4+10}$,
  100 training samples equidistantly sampled from $\langle 0.05, 6.05)$

The first two functions are rather simple polynomials with small integer constants. We chose the function f_1 since it was used in the original SNGP paper [9]. Function f_2 is the Koza-3 function taken from [17]. Functions f_3 and f_4 are modifications of the Koza-3 function so that they involve non-trivial decimal constants. Thus, these functions should represent harder instances than f_1 and f_2. The function f_4 is made even harder than f_3 while breaking the symmetry by adding the term "$-x$". The last function f_5 is a representative of a rational function of two variables. This function, known as Vladislavleva-8 function [17], represents the hardest SR problem used in this work.

4.2 Experimental Setup

All the tested variants of the SNGP use a population of size 400. The population starts with terminal nodes representing the variable x_1 and x_2 and a constant 1.0 followed by function nodes of types {+, -, *, /}. SNGP was run for 25,000 iterations, in each iteration just a single population reevaluation is computed (note, just the nodes that were affected by the mutation are reevaluated). The number of iterations was chosen so as to make the comparisons of the GP and SNGP as fair as possible. This way a balance between processed nodes and fitness evaluations is found, see [19].

The proposed modifications of the SNGP algorithm are configured with the following parameters:

- $upToN \in \{1, 5\}$,
- *selection* is either random (denoted as 'r') or depthwise (denoted as 'd')
- *moveType* is either $moveLeft$ (denoted as 'l'), $moveRight$ (denoted as 'r') or no move (denoted as 'n').

Names of the tested configurations of the SNGP are constructed as follows "SNGP_*upToN*_*selection*_*moveType*". The standard SNGP is denoted as SNGP_1_r_n, i.e. SNGP with a random selection and no move operator applying a single mutation per generation.

Standard GP with generational replacement strategy was used with the following parameters:

- Function set: {+, -, *, /}
- Terminal set: {x_1, x_2, 1.0}
- Population size: 500
- Initialization method: Ramped half-and-half
- Tournament selection: 5 candidates
- Number of generations: 55, i.e. 54 generations plus initialization of the whole population
- Crossover probability: 90%
- Reproduction probability: 10%
- Probability of choosing internal node as crossover point: 90%

For the experiments with the GP we used the Java-based Evolutionary Computation Research System ECJ 22[5].

One hundred independent runs were carried out with each tested algorithm on each benchmark and the observed performance characteristics are

- *fitness* – the mean best fitness (i.e. the sum of absolute errors) over 100 runs;
- *sample rate* – the mean number of successfully solved samples by the best-fitted individual calculated over 100 runs, where the sample is considered to be successfully solved by the individual iff the absolute error achieved by the individual on this sample is less then 0.01;
- *solution rate* – the percentage of complete solutions found within 100 runs, where the runs completely solves the problem iff the best individual generates on all training samples the absolute error less than 0.01;
- *size* – the mean number of nodes of the best solution found calculated over 100 runs.

Table 1: Results of the modified SNGP variants and standard GP on artificial benchmarks f_1 - f_5. The best mean fitness value for each benchmark is highlighted.

function	algorithm	fitness	sample rate (%)	solution rate (%)	nodes
f_1	GP	**0.14**	82.8	49	151
	SNGP_1_r_n	0.65	37.2	5	26.8
	SNGP_1_d_n	0.29	63.4	18	33.7
	SNGP_1_r_l	0.62	38.1	3	22.5
	SNGP_1_d_l	<u>0.25</u>	68.4	24	33.9
	SNGP_1_r_r	0.66	35.9	4	34.5
	SNGP_1_d_r	0.28	66.9	17	56.6
	SNGP_5_d_n	0.16	78.8	49	32.3
	SNGP_5_d_l	0.17	77.5	49	28.5
	SNGP_5_d_r	**0.14**	84.7	53	52.4

Continued on next page

[5] https://cs.gmu.edu/ eclab/projects/ecj/

Table 1 – continued from previous page

function	algorithm	fitness	sample rate (%)	solution rate (%)	nodes
f_2	GP	0.78	88.7	69	175.1
	SNGP_1_r_n	0.85	73.4	33	27.7
	SNGP_1_d_n	0.17	94.4	78	27.6
	SNGP_1_r_l	0.75	78	33	30.8
	SNGP_1_d_l	0.25	92.8	65	27.7
	SNGP_1_r_r	0.84	72.7	17	47
	SNGP_1_d_r	0.15	94.3	82	40.9
	SNGP_5_d_n	**1e-6***	100	100	22.6
	SNGP_5_d_l	0.08	97.8	87	21.2
	SNGP_5_d_r	0.05	98.2	91	37.2
f_3	GP	**1.19**	68.4	0	155
	SNGP_1_r_n	2.7	40	0	28.9
	SNGP_1_d_n	1.5	54.0	0	33.6
	SNGP_1_r_l	2.39	43.7	0	30.1
	SNGP_1_d_l	1.4	59.2	0	35
	SNGP_1_r_r	2.75	37.2	0	43.4
	SNGP_1_d_r	1.6	51.6	0	56.2
	SNGP_5_d_n	1.37	59.2	0	32.6
	SNGP_5_d_l	1.23	65.6	0	30.3
	SNGP_5_d_r	1.35	60.8	0	57.2
f_4	GP	11.0	19.4	0	146.8
	SNGP_1_r_n	10.5	12.9	0	26.1
	SNGP_1_d_n	7.8	21.7	0	34.2
	SNGP_1_r_l	11.2	10.8	0	26.5
	SNGP_1_d_l	8.3	19.0	0	32.9
	SNGP_1_r_r	8.9	19.0	0	45.8
	SNGP_1_d_r	7.4	22.4	0	53.9
	SNGP_5_d_n	7.15	25.8	0	31.5
	SNGP_5_d_l	7.4	24.0	0	28
	SNGP_5_d_r	**6.7**	27.2	0	50.8
f_5	GP	71.2	4.4	0	194.3
	SNGP_1_r_n	64.1	4.0	0	26.0
	SNGP_1_d_n	61.6	3.8	0	35.6
	SNGP_1_r_l	64.9	3.8	0	27.3
	SNGP_1_d_l	**60.0**	4.1	0	34.9
	SNGP_1_r_r	63.6	4.4	0	44.3
	SNGP_1_d_r	61.1	4.1	0	51.6
	SNGP_5_d_n	60.7	3.7	0	32.8
	SNGP_5_d_l	60.9	3.6	0	31.9
	SNGP_5_d_r	60.4	4.0	0	51.1

4.3 Results

Results obtained with the compared algorithms are presented in Table 1. The first observation is that the results obtained on the benchmark f_1 are quite different than the results presented in [9], as the performance of the SNGP is not as good as the SNGP performance presented there whilst the standard GP performs much better than presented in [9]. This might be caused by different configurations of the SNGP and GP used in our work and in [9]. We used different acceptance criterion in SNGP and the generational instead of the steady-state replacement strategy in GP. This observation might indicate that both approaches are quite sensitive to the proper setting of their individual components.

The second observation is that the modified versions of SNGP systematically outperform the standard SNGP with respect to the *fitness*, *sample rate* and *solution rate* performance measures. On the other hand, the modified SNGP is not a clear winner over the standard GP. The SNGP outperforms GP on f_2, f_4 and f_5. On f_1 it performs equally well as the GP. On f_3, all versions of SNGP get outperformed by the GP with respect to the $fitness$. It turns out functions f_3, f_4 and f_5 represent a real challenge for all tested algorithms since no one was able to find a single correct solution within the 100 runs. We hypothesize the hardness of f_3 and f_4 stems from the fact these benchmarks involve non-trivial constants that might be hard to evolve. Function f_5 is hard since it is a rational function.

The third observation is that there is a clear trend showing that the depthwise node selection works significantly better than the random one. Whenever the SNGP configurations differ just in the selection type the one using the depthwise selection outperforms the one with the random selection.

The fourth observation is that the reorganization of the population using either the $moveLeft$ or $moveRight$ operator does not have any systematic impact on the overall performance of the algorithm. It happens only rarely that the SNGP using $moveLeft$ or $moveRight$ outperforms its counterpart configuration with no move operator used. In particular, the $moveLeft$ operator was significantly better[6] than $noMove$ in four cases, the $moveRight$ operator was significantly better than $noMove$ in two cases, all the cases indicated by underlined values. On the other hand, the $noMove$ configuration happened to outperform both the $moveLeft$ and $moveRight$ configuration on function f_2 as indicated by an asterisk.

The fifth observation is that the local search strategy allowing multiple mutations in one iteration outperforms the standard local search procedure with just a single application of the mutation operator per iteration. This is with agreement with our expectations.

Last but not least, the SNGP consistently finds much smaller trees than the GP. This is very important since very often solutions of small size that are interpretable by human are sought in practice.

[6] Checked using the t-test calculated with the significance level $\alpha = 0.05$

5 Hybrid SNGP with Linear Regression

It has widely been reported in the literature that the evolutionary algorithms work much better when hybridized with local search techniques, the concept known as the memetic algorithms [7]. EA serves as a global search strategy, while the local search technique provides an efficient means for fine-tuning the solutions. A similar approach can be used to develop efficient methods for symbolic regression.

Recently, several methods emerged [1], [2], [15], [21], [22] that explicitly restrict the class of models to generalized linear models, i.e. to a linear combination of possibly non-linear basis functions. With the help of linear regression techniques applied to the basis functions, such models can be learned much faster.

GPTIPS [21], [22] is an open-source SR toolbox for MATLAB. It is an implementation of Multi-Gene Genetic Programming (MGGP) [8] and thus has its roots in classical GP. Each solution is composed of multiple independent trees, called genes, and their outputs are linearly combined. The coefficients of this linear combination are computed optimally with respect to the MSE of the final output to the true target values by classical least-squares linear regression. GPTIPS (MGGP) is based on classical Genetic Program- ming. This means that it works with a population of fixed size, subtree mutation, subtree crossover, tournament selection, standard initialization procedures, and is able to handle the internal constants of the model (to certain extent) using ephemeral random constants. The output of GPTIPS is a population of models; it is up to the user to choose the final one. By default, GPTIPS uses Lexicographic Parsimony Pressure [13] using (by default) Expressional Complexity [24] of the models. MGGP was shown to be faster and more accurate than conventional GP [8] and also a comparable or better alternative to classical methods like Support Vector Regression and Artificial Neural Networks [6].

FFX, or Fast Function Extraction [2], is a deterministic algorithm for symbolic regression. It first exhaustively generates a massive set of basis functions, which are then linearly combined using Pathwise Regularized Learning [5], [25] to produce sparse models. The algorithm produces a Pareto-front of models with respect to their accuracy and complexity. Again, it is up to the user to choose the final model. There are two kinds of bases that are generated: univariate bases and bivariate bases. Univariate bases are: a variable raised to a power (chosen from a fixed set of options) and (non-linear) functions applied to another univariate base. Bivariate bases are products of all pairs of univariate bases excluding the pairs where both the bases are of function-type; the author argues that such products are "deemed to be complex". FFX also includes a trick that allows it to produce rational functions of the bases using the same learning procedure. The original paper reports FFX to be more accurate than many classical methods including conventional GP, neural networks and SVM.

EFS, or Evolutionary Feature Synthesis [2] is a recent evolutionary-based algorithm. In EFS, the population does not consist of complete models but rather of features which, collectively, form a single model. The initial population is formed by the original features of the dataset. Then, in each generation, a

model is composed of the features in the current population by Pathwise Regularized Learning and is stored if it is the best. The next step in a generation is the composition of new features by applying unary and binary functions to the features already present in the current population. This way, more complex features are created from simpler ones. Also, the features are selected during this composition step according to the Pearson correlation coefficient with the feature's parents. EFS does not build the symbolic model explicitly – it works with the data of the features in a vectorial fashion and only stores the structure for logging purposes. This results in a very fast algorithm. The original paper reports EFS being comparable to neural networks and similar or better than Multiple Regression Genetic Programming [1] which itself was reported to outperform conventional GP, multiple regression and Scaled Symbolic Regression [11].

In this section we propose two variants of hybrid SNGP that make use of the linear regression technique to improve its performance. Both use the Least Absolute Shrinkage and Selection (LASSO) regression technique, the one used in EFS, to build generalized linear regression models. The first one, denoted as *Single-Run SNGP with LASSO* (s-SNGPL), evolves a population of candidate features for the LASSO regression in a single run of the SNGP. The second variant, denoted as *Iterated SNGP with LASSO* (i-SNGPL), builds the LASSO model in an iterative manner where in each iteration a new feature for the LASSO model is evolved in a separate SNGP run.

5.1 Single-Run SNGP with LASSO

In this method, all features of the generalized linear regression model are evolved in a single run of the SNGP. The outline of the algorithm, see Fig. 4, is very much like the one of the modified SNGP, see Fig. 3. The only difference is in the evaluation of individual nodes in the population and in assessment of the overall population's quality after the content of the population has been altered in each generation. First, a quality of each node is calculated as the Pearson product-moment correlation coefficient between the node's output and the desired output values (line 11). Then, a generalized linear regression model of a subset of features present in the population is calculated using the LASSO technique (line 12). Finally, the fitness of the whole population is calculated as the **sum of absolute errors** between the LASSO regression model output and the desired output values. Thus, the hybrid SNGP uses the same fitness as the modified SNGP, see 3.4.

The complexity of the LASSO model is controlled by (1) the maximal depth of features evolved in the population and (2) the maximum number of features the LASSO model can be composed of. Note, the features can be non-linear functions.

```
1   initialize population of nodes P
2   calculate fitness of nodes in P
    based on their correlation with expected outputs
3   build LASSO regression model LM(P)
4   i ← 0              // number of generations
5   do
6       P' ← P         // work with a copy of P
7       Choose the no. of mutations, k ∈ (1, upToN),
        to be applied to nodes from P'
8       for(n = 1 ... k)
9           choose the node to be mutated, N
10          P' ← Apply mutation to node N of P'
11      calculate fitness of nodes in P'
        based on their correlation with expected outputs
12      build LASSO regression model LM(P')
13      if(LM(P') is not worse than LM(P))
14          P ← P'     // update the population
15      i ← i + 1
16  while (i ≤ maxGenerations)
17  return LM(P)
```

Fig. 4. Outline of the Single-Run SNGP with LASSO.

5.2 Iterated SNGP with LASSO

Unlike the s-SNGPL, here the set of candidate features **F** for the LASSO regression model is not evolved within a single population of SNGP. Instead, an external set **F** is build incrementally, starting from an empty set and adding one feature in each iteration, see Fig. 5.

Each feature f_i is evolved in a separate run of the SNGP (line 6) such that it correlates the most with the residua **R** (i.e. the vector of error values over all training samples) produced by the current LASSO regression model composed of $i-1$ features. The residua are initialized to desired output values of the training samples. The idea is that in each iteration a new feature is evolved such that it possibly helps to reduce the error of the resulting LASSO model. The algorithm stops when either the set of candidate features reached the preset maximum or the error of the LASSO model becomes zero.

6 Experiments with Hybrid SNGP

First experiments with hybrid SNGP variants s-SNGPL and i-SNGPL are carried out on the artificial benchmarks listed in Section 4.1. Another series of

```
1   i ← 0        // number of candidate features
2   initialize feature set F = ∅
3   initialize residua R
4   do
5       i ← i + 1
6       evolve a new feature f_i using separate SNGP
        based on its correlation with R
7       add f_i to F
8       build Lasso model LM(F) using all features in F
9       update residua R
10  while (i ≤ maxFeatures and R ≠ 0)
11  return LM
```

Fig. 5. Outline of the Iterated SNGP with LASSO.

experiments are carried out on real-world benchmarks described in the following section.

6.1 Real-World Benchmarks

Following four real-world benchmarks, acquired from the UCI repository [14], were used in this work

- **Energy Efficiency of Cooling (ENC) and Heating (ENH)** are datasets regarding the energy efficiency of cooling and heating of buildings. Dimension is 8, number of datapoints is 768.
- **Concrete Compressive Strength (CCS)** is a dataset representing a highly non-linear function of concrete age and ingredients. Dimension of the dataset is 8, the number of datapoints is 1030.
- **Airfoil Self-Noise (ASN)** is a dataset regarding the sound pressure levels of airfoils based on measurements from a wind tunnel. Dimension of the dataset is 5, the number of datapoints is 1503.

These benchmarks were used in the work on EFS [2] and other relevant literature.

Each dataset was split 100 times (using the 0.7/0.3 ratio for training/testing). Each algorithm was run once on each of the dataset instances producing a single model. The accuracy and complexity of the resulting models are then aggregated and statistically compared.

6.2 Experimental Setup

We compare the proposed hybrid SNGP algorithms to the GPTIPS, EFS and FFX. We used GPTIPS version 2 retrieved from [23], FFX in version 1.3.4 retrieved from [16], EFS was retrieved from [3]. The goal is to perform a comparison

of the chosen methods as ready-to-use tools. Therefore we didn't modify to the code of the algorithms[7], and we left all of the settings at their default values. We set a timeout to 10 minutes for both EFS and GPTIPS. FFX has no support for timeout. However, the algorithms's performances have not been analyzed from the computation time point of view. No parameter tuning method was used to find an optimal configuration of the compared algorithms for particular benchmarks.

The most important for our evaluation purposes is how the algorithms control the resulting model complexity. GPTIPS has (user-defined) limits on the maximum number of nodes and/or maximum depth, and on the maximum number of bases. By default there is a depth limit of 4, and maximum number of bases (not counting the intercept) is also 4. EFS computes the maximum number of bases from the number of input features, p; the number of bases was set to $3p$ and maximum number of nodes in a base is hard-coded to 5. The FFX procedure results in a maximum model depth of 5.

The hybrid SNGP algorithms with LASSO regression were run on artificial benchmarks with the same population size and the sets of terminals and functions as were used in Section 4.2. The maximum number of generations of s-SNGPL was set to 1000. The maximum number of generations of each individual SNGP run of the i-SNGPL was set to 1000 as well. The maximum number of features the LASSO model can be composed of was set to 15 and the maximum depth of the evolved features was set to 4.

The modified SNGP and hybrid SNGP algorithms with LASSO regression were run on real-world benchmarks with the following changes in the configuration. The set of terminals was extended with constants 2.0, 3.0 and 4.0 and the set of functions was extended with functions square, cube, sqrt and sin. Similarly to EFS, the maximum number of features was set to $3p$, unless stated otherwise.

On the real-world benchmarks, we compare the resulting models with respect to the root mean square error (RMSE) and the number of nodes used in the model. We define the number of nodes as a sum of the numbers of nodes in the model's bases, i.e. we count neither the coefficients (including the intercept) of the linear combination, nor the multiplications between these coefficients and the bases. FFX's hinge functions, having a form $max(0; x - threshold)$ or similar, count as 5 nodes.

6.3 Results on Artificial Benchmarks

Table 2 shows results of the modified SNGP algorithm and the two hybrid SNGP algorithms using LASSO regression on the artificial benchmarks. Only the best performing configuration of the modified SNGP is selected for each benchmark based on the results presented in Table 1.

[7] The only exception is EFS: we changed the round variable to false (which was originally hard-coded to true) according to the issue on the algorithm's GitHub repository, see https://github.com/exgp/efs/issues/1.

There is no single winner algorithm consistently outperforming the others on all five benchmarks. However, there is a clear trend showing that the SNGP without LASSO is doing well on rather simple benchmarks f_1 and f_2 (it is even better than both hybrid algorithms on f_2), i.e. the polynomials that involve only trivial integer constants. As the difficulty of the target model increases (from f_3 to f_5) the hybrid SNGP algorithms start to dominate. Of the two variants the i-SNGPL is better with respect to the SAE performance measure. Note, the superiority of the i-SNGPL is achieved at the cost of rather highly complex models, approximately 150 nodes and more compared to 65 to 118 nodes in case of s-SNGPL and 30 to 50 nodes in case of simple SNGP. These observations are in accordance with our expectations.

Table 2: Comparisons of the modified SNGP with two variants of hybrid SNGP using LASSO regression on artificial benchmarks f_1 - f_5. The best mean SAE value in each row is highlighted. For f_3, f_4 and f_5 the highlighted mean value was significantly better than the other two values as supported by the t-test calculated with the significance level $\alpha = 0.05$.

function	algorithm	SAE	sample rate (%)	solution rate (%)	nodes
f_1	SNGP_5_d_r	0.14	84.7	53	52.4
	s-SNGPL	0.58	44.1	0	42.0
	i-SNGPL	**0.11**	97.5	54	74.8
f_2	SNGP_5_d_n	**1e-6**	100	100	22.6
	s-SNGPL	0.07	99.9	97	85.5
	i-SNGPL	0.34	93.6	44	124.5
f_3	SNGP_5_d_l	1.23	65.6	0	30.3
	s-SNGPL	**0.13**	99.5	84	83.6
	i-SNGPL	0.4	90.8	40	146.6
f_4	SNGP_5_d_r	6.7	27.2	0	50.8
	s-SNGPL	6.3	18.0	0	64.8
	i-SNGPL	**2.53**	34.5	0	147
f_5	SNGP_1_d_l	60.0	4.1	0	34.9
	s-SNGPL	27.8	3.3	0	117.9
	i-SNGPL	**15.9**	4.0	0	170.1

6.4 Results on Real-World Benchmarks

This section presents comparisons of the proposed modified and hybrid SNGP algorithms with GPTIPS, EFS, and FFX on the real-world benchmarks. The first observation based on results in Table 3 is that the simple SNGP without LASSO regression gets defeated by the other algorithms on all benchmarks.

Table 3. Comparisons of SNGP_5_d_n, s-SNGPL and i-SNGPL with GPTIPS, EFS and FFX on the real-world benchmarks with respect to the median RMSE observed on testing data. The best value in each row is highlighted. In all cases the highlighted value was significantly better than the other values as supported by the Mann-Whitney U-test calculated with the significance level $\alpha = 0.01$.

	GPTIPS	EFS	FFX	SNGP_5_d_n	s-SNGPL	i-SNGPL
ENC	2.9073	1.6398	1.7906	3.5657	1.7076	**1.3978**
ENH	2.5375	0.5455	1.0455	3.4295	0.6583	**0.4754**
CCS	8.7618	6.4293	**5.9860**	10.55	6.4052	6.2144
ASN	4.1384	3.6232	3.5804	6.6852	6.1353	**2.9561**

Table 4. Median number of nodes for each algorithm and dataset

	GPTIPS	EFS	FFX	SNGP_5_d_n	s-SNGPL	i-SNGPL
ENC	48	108	136	22	115.5	201
ENH	47.5	105	146	20.5	107	196.5
CCS	43	108	474.5	23	127	201
ASN	58	67	52.5	22.5	88	131

The i-SNGPL outperforms the other algorithms on all benchmarks but the CCS, where the FFX exhibits the best median RMSE value. However, this is at the cost of very large models produced, see Table 4. Also the superiority of i-SNGPL on the three benchmarks is thank to large models produced by the algorithm. The s-SNGPL is competitive to the three compared algorithms with respect to the median RMSE as well as the model size.

Tables 5 and 6 show the performance of s-SNGPL and i-SNGPL achieved with smaller LASSO models. Values $k_1 \dots k_4$ specify the maximum number of features the algorithms are allowed to use in the LASSO regression models. An interesting observation is that both algorithms, and especially the i-SNGPL one, stay competitive with the compared algorithms even when producing smaller models.

Figure 6 presents progress plots observed for the s-SNGPL algorithm on real-world benchmarks. In each generation, the mean of the best-so-far fitness (i.e. SAE) calculated over 100 independent runs is shown. It illustrates the effect of the evolutionary component of the algorithm as there is a clear continuous improvement in the best-so-far fitness along the whole run.

7 Conclusions

This paper deals with the Single Node Genetic Programming method, proposes its modifications and ways of hybridization to improve its performance.

Table 5. Median RMSE and median number of nodes observed for **s-SNGPL** on testing data. Performance of the algorithms is tested for different values of the maximum number of features allowed for the LASSO model. Values $k_1 = 12$, $k_2 = 16$, $k_3 = 20$ and $k_4 = 24$ are tested on benchmarks ENC, ENH and CCS. Values $k_1 = 8$, $k_2 = 10$, $k_3 = 12$ and $k_4 = 15$ are tested on benchmark ASN.

	k_1		k_2		k_3		k_4	
	RMSE	#nodes	RMSE	#nodes	RMSE	#nodes	RMSE	nodes
ENC	1.8582	58.5	1.7694	82	1.6897	94	1.7076	115.5
ENH	1.0842	60	0.8461	75	0.8123	95	0.6583	107
CCS	6.8929	63	6.6912	82	6.5595	102	6.4053	127
ASN	4.0013	48	3.8395	61	3.7463	70	3.4817	88

Table 6. Median RMSE and median number of nodes observed for **i-SNGPL** on testing data. Performance of the algorithms is tested for different values of the maximum number of features allowed for the LASSO model. Values $k_1 = 12$, $k_2 = 16$, $k_3 = 20$ and $k_4 = 24$ are tested on benchmarks ENC, ENH and CCS. Values $k_1 = 8$, $k_2 = 10$, $k_3 = 12$ and $k_4 = 15$ are tested on benchmark ASN.

	k_1		k_2		k_3		k_4	
	RMSE	#nodes	RMSE	#nodes	RMSE	#nodes	RMSE	nodes
ENC	1.5490	101.5	1.4502	133	1.4085	170	1.3978	201.5
ENH	0.5648	97	0.5179	130.5	0.4978	166	0.4754	196.5
CCS	6.5912	103	6.4412	135	6.2973	167	6.2144	201
ASN	3.3894	71	3.2492	89.5	3.0989	106	2.9561	131

First, three extensions of the standard SNGP, namely (1) a selection strategy for choosing nodes to be mutated based on the depth and performance of nodes, (2) operators for placing a compact version of the best-performing graph to the beginning and to the end of the population, respectively, and (3) a local search strategy with multiple mutations applied in each iteration were proposed.

These modifications have been experimentally evaluated on five artificial symbolic regression benchmarks and compared with standard GP and SNGP. The achieved results are promising showing the potential of the proposed modifications to improve the performance of the SNGP algorithm.

Further, two variants of hybrid SNGP utilizing the linear regression technique, LASSO, were proposed. The proposed hybrid algorithms have been compared to the state-of-the-art symbolic regression methods making use of the linear regression techniques on four real-world benchmarks. The results show the proposed algorithms are at least competitive with or better than the compared methods.

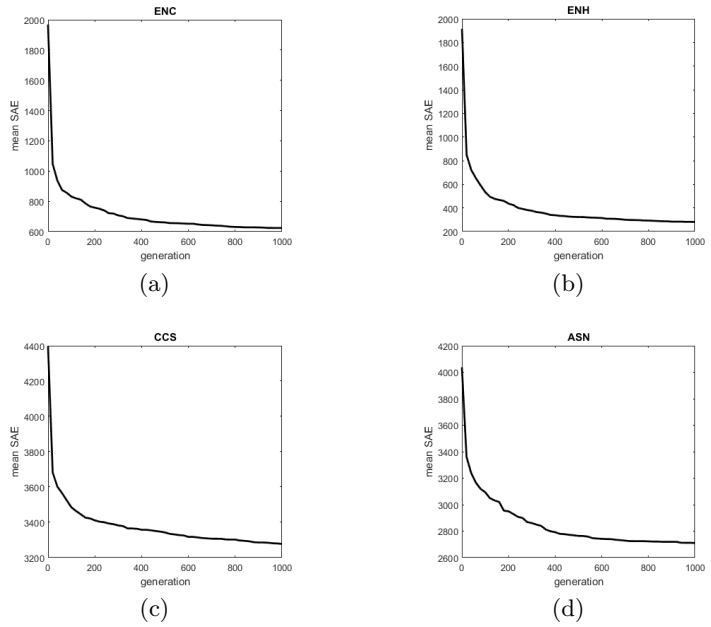

Fig. 6. Plots showing an average progress of the best SAE value for s-SNGPL on real-world benchmarks.

The next step of our research will be to carry out a thorough experimental evaluation of the modified SNGP algorithms with the primary objectives being the speed of convergence and the ability to react fast to the changes of the environment in order to be able to deploy the algorithm within the dynamic symbolic regression scenario. Further investigations will include utilization of new mutation operators, identification of suitable "high-level" basic functions to the SNGP's function set, design of mechanisms to evolve inner constants of the models and mechanisms for escaping from local optima.

8 Acknowledgment

This research was supported by the Grant Agency of the Czech Republic (GAČR) with the grant no. 15-22731S entitled "Symbolic Regression for Reinforcement Learning in Continuous Spaces".

References

1. Arnaldo, I., Krawiec, K., and O'Reilly, U.-M. (2014). Multiple regression genetic programming. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, pages 879–886, New York, NY, USA. ACM.

2. Arnaldo, I., O'Reilly, U.-M., and Veeramachaneni, K. (2015). Building predictive models via feature synthesis. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 983–990, New York, NY, USA. ACM.
3. EFS commit 6d991fa. http://github.com/exgp/efs/tree/6d991fa.
4. Ferreira, C. (2001). Gene expression programming: A new adaptive algorithm for solving problems. *Complex Systems*, 13(2):87–129.
5. Friedman, J., H. T. and Tibshirani, R. (2010). Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1):1–22.
6. Garg, A., G. A. and Tai., K. (2013). A multi-gene genetic programming model for estimating stress-dependent soil water retention curves. *Computational Geosciences*, 18(1):45–56.
7. Hart, E., Smith, J. E. and Krasnogor, N. (2005). Recent Advances in Memetic Algorithms. Studies in Fuzziness and Soft Computing Volume 166 2005.
8. Hinchliffe, M., Hiden, H., McKay, B., Willis, M., Tham, M., and Barton, G. (1996). Modelling chemical process systems using a multi-gene genetic programming algorithm. In Koza, J. R., editor, *Late Breaking Papers at the Genetic Programming 1996 Conference*, pages 56–65.
9. Jackson, D. (2012a). A new, node-focused model for genetic programming. In *EuroGP 2012*, pages 49–60.
10. Jackson, D. (2012b). Single node genetic programming on problems with side effects. In *PPSN XII*, pages 327–336.
11. Keijzer, M. (2004). Scaled symbolic regression. *Genetic Programming and Evolvable Machines*, 5(3):259–269.
12. Koza, J. (1992). *Genetic programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, 2nd edition.
13. Luke, S. and Panait, L. (2002). Lexicographic parsimony pressure. In *Proceedings of GECCO-2002*, pages 829–836. Morgan Kaufmann Publishers.
14. Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.
15. McConaghy, T. (2011). Ffx: Fast, scalable, deterministic symbolic regression technology. In R. Riolo, E. V. and Moore, J. H., editors, *Genetic Programming Theory and Practice IX, Genetic and Evolutionary Computation*, pages 235–260.
16. FFX 1.3.4. http://pypi.python.org/pypi/ffx/1.3.4.
17. McDermott, J. e. a. (2012). Genetic programming needs better benchmarks. In *Proceedings of the GECCO '12*, pages 791–798, New York, NY, USA. ACM.
18. Miller, J. and Thomson, P. (2000). Cartesian genetic programming. In *Poli, R. et al. (eds.) EuroGP 2000, LNCS, vol. 1802, pp. 121–132*. Springer.
19. Ryan, C. and Azad, R. M. A. (2014). A simple approach to lifetime learning in genetic programming-based symbolic regression. *Evolutionary Computation*, 22(2):287–317.
20. Ryan, C., Collins, J., Collins, J., and O'Neill, M. (1998). Grammatical evolution: Evolving programs for an arbitrary language. In *LNCS 1391, Proceedings of the First European Workshop on Genetic Programming*, pages 83–95. Springer-Verlag.
21. Searson, D. P., L. D. E. and Willis, M. J. (2010). Gptips: an open source genetic programming toolbox for multigene symbolic regression. In *International MultiConference of Engineers and Computer Scientists*, volume 1, pages 77–80.
22. Searson, D. P. (2015). Gptips2: an open-source software platform for symbolic datamining. In A. H. Gandomi, A. H. A. and Ryan, C., editors, *Springer Handbook of Genetic Programming Applications*.

23. GPTIPS 2. http://sites.google.com/site/gptips4matlab.
24. Vladislavleva, E. J., Smits, G. F., and Den Hertog, D. (2009). Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. *Trans. Evol. Comp*, 13(2):333–349.
25. Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320.