Matching Subtrees in Genetic Programming Crossover Operator

Martin Šlapák*, Roman Neruda†
*Department of Theoretical Computer Science, Faculty of Information Technology, CTU in Prague,
Thákurova 9, 160 00, Praha 6, Czech Republic;
Email: slapamar@fit.cvut.cz
†Institute of Computer Science, Academy of Sciences of the Czech Republic,
Pod Vodárenskou věží 271/2, 182 07, Praha 8, Czech Republic;
Email: roman@cs.cas.cz

***Abstract*—In this paper we study techniques that should reduce the destructive impact of crossover in genetic programming. The quality of crossover offsprings is often lower than ancestors due to the fact that a small change in individual's genotype tree structure has a great impact to its phenotype. Therefore we propose and test several methods for matching subtrees to find the best possible cutting point for crossover of trees. Our approach utilizes the adaptive probability of operators with the intent to reinforce the well-performing operators. A relation to the semantic genetic programming approach is also investigated. The experimental results show that the average arity based technique performs best from the proposed methods.**

I. Introduction

The primary goal of the crossover operator in evolutionary computing is to give individuals the ability to exchange parts of their genotype. In accord with selection pressure this should support the combination of high-quality sections of genotypes. In nature the crossover also brings resistance to errors caused by mutation of genotype. This is the reason why many researchers believe that a well-defined crossover is important, although approaches like evolutionary programming dealing with more complex genotypes usually abandon crossover in favor of sophisticated mutation operators.

The most common encoding of individuals in genetic programming (GP) represents a genotype as a tree structure. There are also other, mostly indirect, encoding approaches, such as evolving a set of commands describing creation process of the final tree, linear GP, or grammatical approaches. In this work we focus on a direct tree encoding where an individual is represented as a syntactic tree corresponding to the program it realizes.

There are several problems with applying crossover operator to tree structures describing the genotype. In many cases the best of ancestors produced from mating of two parent candidates is even worse than both of its parents. This is the reason why standard simple crossover (swapping randomly selected subtrees) [1] is often not the preferred operator in GP. The impact of operator applying is too strong in many cases, since a small change in individual's genotype tree structure can have a big and unpredictable impact to its phenotype.

The structure of this work is as follows: In the following section we briefly review relevant existing work on alternative crossovers for GP. Section III introduces various measures of subtree similarity. Next section introduces our environment for performing the tests, namely by describing details of a our genetic program. Results of experiments on standard benchmark tasks are reported in section VI and discussed in the Conclusion section.

II. Related work

There have been several works on alternative crossover operators that improve the exchange of suitable subtrees. The context preserving crossover by D'Haeseleer [2] attempts to preserve the context in which subtrees appeared in the parent trees. The author introduced a coordinate scheme for nodes in a tree and allows crossover only between nodes with matching coordinates. Uy et al. [3] introduced a semantic distance of subtrees and defined a semantic similarity-based crossover which controls the distance of subtrees chosen for crossover. Their approach is tested on several symbolic regression problems.

A modification of crossover and mutation which measures performance of subtrees as a guidance for subtree selection has been proposed in [4]. Authors consider several approaches from simple performance of a subtree, to complexity and performance, to correlation of subtrees as the guidance factor. The monograph [5] also introduces a way how to measure structural similarity of subtrees utilized by crossover.

Beadle and Johnson in [6] introduce a semantically driven crossover which prevents offsprings to be semantically equivalent to their parents. The authors use a cannonical representation of trees in order to check the semantic equivalence without accessing the fitness. Their approach is further extended by [7] where authors investigate the effect of semantic guidance to the crossover. They define two operators, one considering the semantics of the exchanged subtrees, and one comparing the semantics of the offspring trees to their parents. They show that these operators perform better on a set of polynomial symbolic regression problems. A more detailed survey od semantically based methods of crossover

978-1-5386-2165-3/17/$31.00 ©2017 IEEE

operator in genetic programming is given by Vanneschi at al. [8].

The common element of all mentioned works is to prevent a distortion of a well performing subtree, which can be seem as a problem of breaking of a good building block. The idea behind our approach is very similar — to introduce only slight and reasonable changes in phenotype.

III. Subtree similarity

We present six methods to measure compatibility of two subtrees. These methods are compared with traditional crossover performing a random approach to find matching trees as a baseline. All methods take two (sub)-trees and compute a measure of similarity of them, based on various criteria.

In the following we consider these two example trees to clarify our methods (cf. Fig. 1):

$$T_a = \text{add}(x, \text{sqrt}(y))$$

$$T_b = \text{sub}(x, 1)$$

The *average arity* method (AVA) computes average arity of functions in inner nodes. $\text{AVA}(T_a) = (2+1)/2 = 1.5, \text{AVA}(T_b) = 2$.

The *common variable set* method (CVS) counts the size of a set of the identical variables in the leaves of both subtrees. It takes the variables from leaves of T_a and from leaves of T_b. These two sets are thereafter intersected. The number of common variables then serves as a measure of semantic similarity of the subtrees. The crossover tends to choose more similar subtrees for exchange, i.e. those with more common variables. $\text{CVS}(T_a) = \text{CVS}(T_b)) = 1$ because the only one variable (x) is common for both trees.

The *common operator set* method does the same as *CVS* method, but instead of leaves' variables, it takes into account the functions (operators) from inner nodes. $COS(T_a) = COS(T_b) = \emptyset$

The *recurrent evaluation* method (REV) tries to represent a value of a subtree. It pushes a value of 0 to all variables in a tree, and retrieves a value from the root node. This value is again substituted to all variables, and so on for 20 times. The final root value is taken as a representative of given subtree. $\text{REV}(T_a) = 0 + \sqrt{0} = 0, \text{REV}(T_b) = -20$

The *vector evaluation* (VEV) method takes a randomly generated set of 20 numerical vectors $v_0, \ldots, v_{20}$. The length of these vectors corresponds to the number of unique variables in compared trees – in our case of T_a and T_b it is 2 (for x and y). The tree is evaluated for each vector v_i, thus we obtain a vector r of 20 results for each of given trees T_a, T_b. Next, the *VEV* method computes the distance δ of these vectors by:

$$\delta(r^{T_a}, r^{T_b}) = \sum_{i=0}^{i<20} (r_i^{T_a} - r_i^{T_b})^2$$

The VEV method is similar to ideas introduced in semantic genetic programming [9] or semantic similarity-based crossover proposed by [3]. It measures the performance of the tree on given set of inputs.

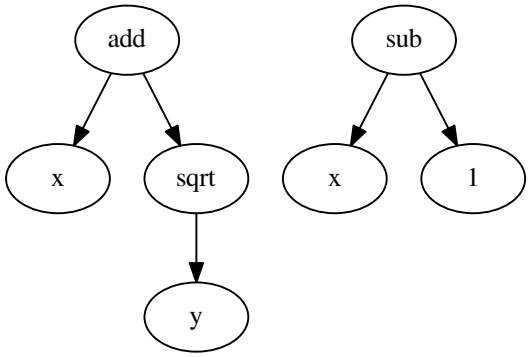

Fig. 1: Two simple example trees.

The *string* (STR) method is based on a string serialization of tree. The given trees T_a, T_b are represented after serialization as S1 = "$\text{add}(x, \text{sqrt}(y))$" and S2 = "$\text{sub}(x, 1)$". Next, the Levensthein distance[10] of these two strings is computed – $\text{STR}(T_a, T_b) = 10$. This method should mirror the semantic equivalence of the trees.

For comparison, as a baseline solution, we also take randomly selected subtrees for crossover operator.

IV. Genetic programming test environment

In the following we describe the problems to solve and the GP techniques we utilize as an environment to test our subtree similarity methods described above.

A. The problem definition

The problem at hand is the simple symbolic regression described in Koza [1]. The goal is to fit a real valued function by a tree evolved by the GP. The tree consists of inner nodes containing simple mathematical functions like addition, multiplication or $\text{abs}()$, and leaf nodes with numerical constants from $\mathbb{R}$, or variables. This variables also assume values from $\mathbb{R}$. The sample function is compared with evolved function in defined amount of points from a given interval. Some datasets use equidistant grid of inputs and some use fixed number of uniform random samples drawn from given interval, as we describe in detail further.

For practical reasons we redefined some mathematical functions so they are defined in the whole interval $[-\infty, \infty]$. Also positive or negative ∞ was replaced by *MAX_DOUBLE* constant with matching sign during evaluation of a tree. Therefore:

- $\frac{x}{0} = 1$ for $x \in \mathbb{R}$
- $\frac{1}{x} = 1$ for $x = 0$
- $\log x = \log |x|$ for $x \in \mathbb{R} \setminus 0$, -20 for $x = 0$
- $\ln x = \ln |x|$ for $x \in \mathbb{R} \setminus 0$, -20 for $x = 0$
- $\sqrt{x} = \sqrt{|x|}$ for $x \in \mathbb{R}$

The set of available simple function consists of: $-, +, \times, \%, \sqrt{x}, |x|, 1/x, -x, x^2, \sin(x), \cos(x)$. The first four are binary functions (have an arity of 2) and the rest are unary functions (arity = 1). The % symbol represents a modulo operation.

TABLE I: The configuration of GP in experiments.

population size	100
number of generations	200
selection method	tournament
tournament size	15
elitism	enabled
initialization	see Sec. IV-D
max. initial tree depth	10
operators	see Sec. IV-E

B. GP evolution configuration

Each of the experiments was performed with configuration as shown in the table I. Some specific variations are mentioned in particular experiments. The section IV-D is dedicated to process of initialization of trees in individuals.

The fitness function f is defined as in equation 1.

$$f(t) = \sum_{i=0}^{N} |x(i) - t(i)| \tag{1}$$

Where N is a number of sampling points of the fitted function x, $x(i)$ is value of the function in given point i and $t(i)$ is value returned by the evaluation of the tree which represents evaluated function. In all experiments the fitness function is maximized.

C. Simple benchmark – Sample function sets

This subsection describes how we prepared a simple benchmark set called SimpleF in our experiments. The benchmark is designed for two purposes. At first it is used to tune parameters of evolution, its initialization and selecting the best variant of adaptive operators approach. Secondary, it is used as one of several benchmarks to measure performance of proposed methods of subtree matching. The idea is to generate several random trees representing target functions, and to choose the medium complex ones as a set of fitness functions.

In a more detail, the target function which we try to fit, is a combination of simple functions mentioned above represented as tree. We generated 45 samples of full trees of depth 10 at random. These full trees are relaxed after initialization. The relaxation means that all subtrees which contain no variables are replaced by a constant leaf node. The deeper insight to initialization will be given in section IV-D.

For each from these trees we try to evolve a function which fits them by a standard GP procedure. The trees are afterward sorted by fitness function of the best evolved individual for each of them. Finally, the best and the worst thirds are removed, and the remaining 15 trees became a testing set. It can be presumed that functions represented by these trees are not so hard to fit for the GP. This dataset is named *SimpleF*.

D. Tree initialization

There are many initialization methods to construct a tree in GP [11]. Our sample function trees are full trees of a given depth. That means each leaf is in given depth and each inner node has number of children which matches to arity of function in that node. The function is selected from available simple functions at random. In leafs, there are randomly chosen constants from range $[0, 1]$ with uniform distribution. To ensure that the tree contains full set of variables we need to replace randomly selected leaf constants with variables. In the SimpleF dataset, there is only one variable x. In experiments on this dataset we need to use internally more labels for the same variable – eg. x_0, x_1, for the method *common variable set* (see section III). Another benchmarks (see section VI) use more dimensional function samples which naturally imply labeling $x_0, \ldots, x_n$ for its variables and need no artificial labeling.

To achieve sufficient diversity in initial population in GP we combine several methods of tree initialization. The first method creates a minimal tree which contains all used variables. In the case of SimpleF dataset it means that tree consists of one inner node with arity one or two and matching number of child nodes. A such tree is a hopeful building block for construction of complex trees. The second and third methods are implementation of the grow, respectively the full approach, as described in [11]. Koza calls the combination of *grow* and *full* methods as *ramped half-and-half*. The last initialization method creates random tree with constants in leaves at given depth. In fact when this tree is relaxed (evaluated) it can be replaced by only one constant node – this may be a small disadvantage during the evolution when we want to avoid bloating of trees by relaxing them.

E. Evolutionary operators

There are two approaches how to apply GP operators to individuals. The operator can be applied to the selected individual only once per generation, and the newly created offsprings are preserved from other operators till the new generation is created. On the other hand, more operators can be applied in sequence to one individual. Usually the application of operators is given by the design of particular GP. The other possibility is to adaptively react to the performance of operators and favor the more successfull operators. This approach is known as self-adaptive evolution [12], [13].

In our approach, each operator O has its own probability $0 \leq P_O \leq 0.9$ to be applied. This probability is adaptively changed during evolution. When the ancestor has better fitness than the parent, the P_O of the applied operator is increased by 1×10^{-5}, if not, it is decreased by 1×10^{-6}. The sum of all P_O can exceed 1.0 but probabilities of all operators are then scaled to the range $[0, 1]$.

The following several mutation operators, and a crossover, guided by different similarity measured, were used in our experiments:

- *mutateLeafHillClimb*
  For a randomly selected leaf value the local space is searched, and the best found value replaces the old one.
- *mutateLeafByVal*
  A randomly selected leaf with constant value is changed by its momentum. This momentum is updated by -5 % to +5 %

TABLE II: Probabilities of operators before and after evolution. The values are averaged over 1000 runs with given setup.

operator	pre P_O	post P_O
mutateLeafHillClimb	0.143	0.228
mutateLeafByVal	0.143	0.174
mutateLeafAddSubtree	0.143	0.144
mutateLeafSwitchVariableConst	0.143	0.269
mutateInnerNodeFunction	0.143	0.144
mutateInnerNodeToConstLeaf	0.143	0.145
cross	0.143	0.143

- *mutateLeafAddSubtree*
  A randomly selected leaf is replaced by a newly generated tree of depth 3. The grow method for generating is used.
- *mutateLeafSwitchVariableConst*
  If selected leaf contains a constant, it is replaced by a randomly selected variable, and vice versa.
- *mutateInnerNodeFunction*
  In given inner node the function is changed to another with the same arity. When no function of current arity is available then a new function is chosen randomly and a proper count of child nodes are either constructed or destroyed as needed.
- *mutateInnerNodeToConstLeaf*
  The selected inner node is replaced by a constant and becomes a leaf.
- *cross*
  The cross operator take the second individual by tournament selection and tries to find a best matching subtrees of itself and the other tree. The measure of similarity will be defined in next section. The best matching subtrees are finally swapped.

V. Experimental results of adaptive operators

This section describes the experiments perfomed with different initialization of P_O's. There are four experiments as follows: *Uniformly distributed probabilities*, *preferred crossover*, *supressed crossover* and *only crossover*. All these experiments are performed on the *SimpleF* dataset.

A. Uniformly distributed probabilities

The first experiment put the same probability to all operators as you can see in table II. The impact of adaptive changes in probabilities is shown as the second column. We can state that *leafHillclimb* and *leafSwitchVariableConst* operators brought better descendants then others. The probability of crossover operator left unchanged.

B. Preferred crossover

The second experiment examined the situation when the crossover has significantly higher P_O. The results are shown in the table III. All mutation operators have increased their P_O. The crossover operator decreased its P_O by 13.6 %.This behavior brings idea of next experiment – the defended crossover.

TABLE III: The crossover is strongly preferred operator but P_Os of mutation operators were increased in contrast to decreased P_O of crossover operator. The values are averaged over 1000 runs with given setup.

operator	pre P_O	post P_O
mutateLeafHillClimb	0.05	0.086
mutateLeafByVal	0.05	0.063
mutateLeafAddSubtree	0.05	0.051
mutateLeafSwitchVariableConst	0.05	0.104
mutateInnerNodeFunction	0.05	0.051
mutateInnerNodeToConstLeaf	0.05	0.052
cross	0.7	0.605

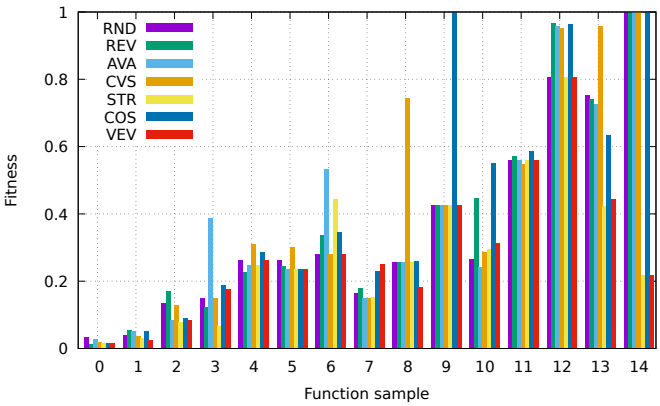

Fig. 2: Averaged performance from 30 runs of each method on the SimpleF dataset.

C. Suppressed crossover

In this experiment there is blocked a punishment for crossover operator when it creates a worse ancestor then itself. The final P_Os of all mutation operators are nearly the same as in case of previous experiment – *preferred crossover*.

So we can state that using an evolution with more generations leads only to the primary reinforcement of two mutation operators. An only artificial defended crossover preserves its initial higher chance to be applied.

D. Crossover only

In the last experiment of adaptive operators we have disabled all mutation operators. Therefore only the crossover operator is considered. The final crossover's P_O was 0.621 from initial 0.7. It can be stated that nevertheless the worse ancestors are still created by the crossover, the evolution is capable to find optimal solutions on SimpleF dataset. The figure 2 shows averaged performance from 30 runs of each method on the SimpleF dataset. Horizontal axis represents given samples which were fitted. On the vertical axis there is an average fitness function value of the best individuals.

For illustration, the best evolved trees from experiments on SimpleF dataset are presented on figures 3 and 4. Blue nodes represent variables, constants are gray, and yellow nodes represent atomic function.

For further experiments with subtree similarity methods we used *suppressed crossover* setup. The crossover had much higher probability ($P_O = 0.7$) then all other operators ($P_O =$

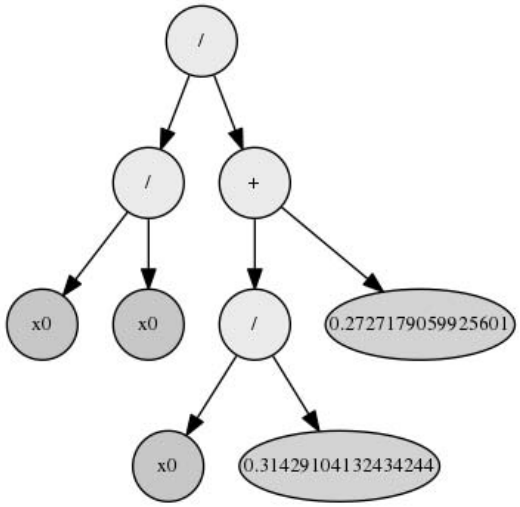

Fig. 3: An example of the best evolved tree from experiments on SimpleF dataset.

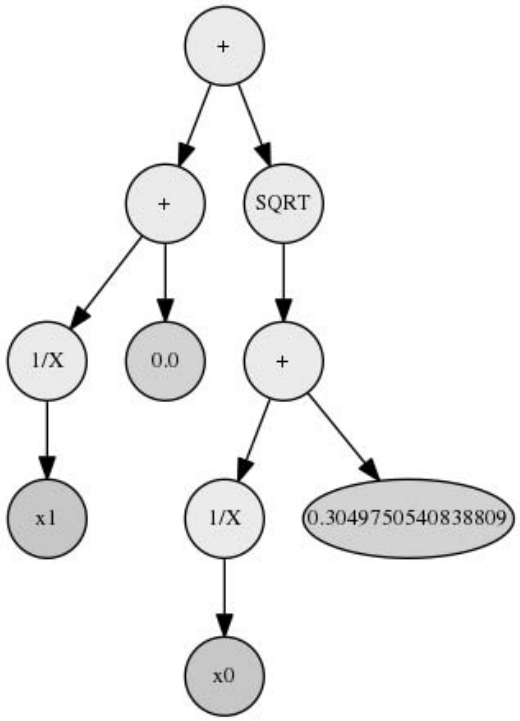

Fig. 4: An example of the best evolved tree from experiments on SimpleF dataset. The tree is not relaxed during the evolution due to preservation of diversity in population – see additon of 0.0 in the left side of depicted tree.

0.05 each). All other parameters of GP were the same as shown in the table I.

VI. Experimental results of subtree matching

The overall performance of the subtree matching methods are described in this section.

The survey [14] presented several GP benchmarks and discuss its suitability for GP experiments. Beside our own artificial dataset SimpleF for benchmarking (see Sec. IV-C) we adapted the following benchmark functions from the survey: kaijzer, nguyen7, pagie1, vladislavleva4. The last two show itself as too difficult – no one of proposed methods was capable to perform significantly better than random baseline.

The table IV shows benchmarks configuration. Each combination of matching method and dataset is computed 30 times to avoid an influence of a random essence of GP. Over all datasets totaly 61 unique function samples were fitted by each method.

TABLE IV: The used benchmarks configuration. U[a,b,c] is c uniform random samples drawn from a to b, inclusive, for the variable. E[a,b,c] is a grid of points evenly spaced (for this variable) with an interval of c, from a to b inclusive.

benchmark	# vars.	data points	simple functions
keijzer	1	E(1, 50, 1)	$+, \times, 1/n, -n, \sqrt{n}$
nguyen7	1	U(0, 2, 20)	$+, -, \times, /, \hat{}, \ln(n)$, $\sin(n), \cos n$
pagie1	2	E(-5, 5, 0.4)	$+, \times, 1/n, \hat{}$
vlad4	5	U(0.05, 6.05, 1024)	$+, -, \times, \%, n^2$
SimpleF	1	E(-5, 5, 20)	$+, \times, 1/n, -n, \sqrt{n}$

The results are summarized in the table V. The bold text represents the best method in current experiment.

Each of methods was as good as the *random* baseline (RND) at least in one of benchmarks. We tested a null hypothesis as folows H_0: The means of samples of RND and another method on given dataset are the same. The results of T-test of H_0 are shown in table VI – the table contains probabilities that the means of RND and given method are the same. In the table there is missing row for pagie1 dataset. It is caused by the fact that this dataset has only one function for fitting and therefore there are insufficient amount of data fot T-test. For keijzer dataset the *average arity* (AVA) method is significantly (at level 0.01) better than RND. On the same level of significance there is no another method better than RND baseline. On nguyen7 dataset there is *vector evaluation* (VEV) which is better than RND starting from significance level 0.1. The remaining two datasets vladislavleva4 and simpleF seems to be too difficult for all proposed methods and used configuration of GP.

VII. Conclusion

The experiments with function fitting show that only *average arity* and *vector evaluation* crossovers outperformed the random matching of subtrees on specific dataset. The AVA method is about 15 % better on keijzer dataset than the RND baseline. The VEV outperformed RND on nguyen7. The remaining methods *recurrent evaluation*, *string*, *common operator ser* and *common variable set* are not significantly

TABLE V: The subtree similarity methods comparison on symbolic regression problems. The columns represent each of methods and their performance on given datasets. The results for each tuple method-dataset are averaged for all function samples in given dataset and 30 times repetition on each sample. The values are scaled by value or *RND* method.

	method						
dataset	**AVA**	**RND**	**CVS**	**STR**	**REV**	**COS**	**VEV**
keijzer	**1.150**	1.000	**1.040**	0.946	0.979	0.957	1.009
nguyen7	1.024	1.000	0.981	0.970	**1.003**	0.985	**1.041**
pagie1	0.997	1.000	0.976	1.013	0.980	0.991	0.998
vlad4	1.000	1.000	1.000	1.000	1.000	1.000	1.000
simpleF	1.059	1.000	0.971	**1.034**	0.995	**1.007**	0.867
avg	1.046	1.000	0.994	0.993	0.992	0.988	0.983
rank	1.	2.	3.	4.	5.	6.	7.

TABLE VI: The probabilities obtained from T-test for null hypothesis that means of RND and the another method are the same. The pagie1 dataset is missing because it has too small amount of data for statistical testing.

method	**AVA**	**CVS**	**STR**	**REV**	**COS**	**VEV**
keijzer	3,28E-06	0,18	0,06	0,01	0,06	0,38
nguyen7	0,24	0,47	0,26	0,18	0,31	0,09
vlad4	0,49	0,49	0,50	0,49	0,50	0,49
simpleF	0,46	0,50	0,48	0,48	0,50	0,41

better than *random* approach to the crossover of trees on any dataset. It may be caused by the problem domain or the current setup – e.g. the initial constant supplied to REV method, or the selected text representation in STR method.

A REV method can be further modified – one can change a fixed number of iterations, make number of iterations variable, or stop when value converges to some stable value. An interesting idea is also to use another initial value for the first substitution – it may be made dependent on interval of desired function sample.

For the future work we will focus on the comparison of presented methods on another problem like the Lawnmower problem or Artificial Ant (both described in Koza [1]).

ACKNOWLEDGMENT

Martin Šlapák has been partially supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS17/210/OHK3/3T/18. Roman Neruda has been partially supported by the Czech National Foundation project no. GA15-19877S.

REFERENCES

[1] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[2] P. D'haeseleer, "Context preserving crossover in genetic programming," in *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, Jun 1994, pp. 256–261 vol.1.

[3] N. Q. Uy, N. X. Hoai, M. ONeill, R. I. McKay, and E. Galván-López, "Semantically-based crossover in genetic programming: application to real-valued symbolic regression," *Genetic Programming and Evolvable Machines*, vol. 12, no. 2, pp. 91–119, 2011.

[4] H. de Garis, "Extending genetic programming with recombinative guidance," in *Advances in Genetic Programming, vol. 2*, 1996.

[5] W. Banzhaf, F. D. Francone, R. E. Keller, and P. Nordin, *Genetic Programming: An Introduction: on the Automatic Evolution of Computer Programs and Its Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[6] L. Beadle and C. G. Johnson, "Semantically driven crossover in genetic programming," in *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*. IEEE, 2008, pp. 111–116.

[7] Q. U. Nguyen, X. H. Nguyen, and M. ONeill, "Semantic aware crossover for genetic programming: the case for real-valued function regression," in *European Conference on Genetic Programming*. Springer, 2009, pp. 292–302.

[8] L. Vanneschi, M. Castelli, and S. Silva, "A survey of semantic methods in genetic programming," *Genetic Programming and Evolvable Machines*, vol. 15, no. 2, pp. 195–214, 2014.

[9] A. Moraglio, K. Krawiec, and C. G. Johnson, "Geometric semantic genetic programming," in *International Conference on Parallel Problem Solving from Nature*. Springer, 2012, pp. 21–31.

[10] L. Yujian and L. Bo, "A normalized levenshtein distance metric," *IEEE transactions on pattern analysis and machine intelligence*, vol. 29, no. 6, pp. 1091–1095, 2007.

[11] R. Poli, W. B. Langdon, and N. F. McPhee, *A field guide to genetic programming*. Published via `http://lulu.com`, 2008, (With contributions by J. R. Koza). [Online]. Available: http://www.gp-field-guide.org.uk

[12] N. Saravanan, D. B. Fogel, and K. M. Nelson, "A comparison of methods for self-adaptation in evolutionary algorithms," *BioSystems*, vol. 36, no. 2, pp. 157–166, 1995.

[13] K. Deb and H.-G. Beyer, "Self-adaptation in real-parameter genetic algorithms with simulated binary crossover," in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1*. Morgan Kaufmann Publishers Inc., 1999, pp. 172–179.

[14] D. R. White, J. McDermott, M. Castelli, L. Manzoni, B. W. Goldman, G. Kronberger, W. Jaśkowski, U.-M. O'Reilly, and S. Luke, "Better gp benchmarks: community survey results and proposals," *Genetic Programming and Evolvable Machines*, vol. 14, no. 1, pp. 3–29, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10710-012-9177-2