Approximating Geometric Crossover by Semantic Backpropagation

Krzysztof Krawiec
krawiec@cs.put.poznan.pl

Tomasz Pawlak
tpawlak@cs.put.poznan.pl

Institute of Computing Science, Poznan University of Technology
Piotrowo 2, 60965 Poznań, Poland

ABSTRACT

We propose a novel crossover operator for tree-based genetic programming, that produces approximately geometric offspring. We empirically analyze certain aspects of geometry of crossover operators and verify performance of the new operator on both, training and test fitness cases coming from set of symbolic regression benchmarks. The operator shows superior performance and higher probability of producing geometric offspring than tree-swapping crossover and other semantic-aware control methods.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

genetic programming, program semantics, geometric crossover

1. INTRODUCTION AND MOTIVATIONS

The conventional search operators used in genetic programming (GP) are designed to be generic. Whether the domain is symbolic regression, Boolean function synthesis, or artificial ant, a standard operator like tree-swapping crossover and subtree-replacing mutation manipulates the code in the same way. Their actual impact on the *effect* of program execution (i.e., on program output) can be very complex and depend strongly not only on the code of the program that undergoes modification, but also on the choice of location at which the operator is applied. As a result, a mutated GP program yields output that is often very different from that of its parent. Similarly, an offspring solution resulting from GP crossover frequently cannot be said to be a 'mixture' of its parents in terms of the output it produces. This is unfortunate as, in the end, it is the output of program execution that undergoes fitness assessment and drives the search process.

This issue was known since the early days of GP, and characterized from different perspectives. The convenient framing of this phenomenon involves the notion of *locality* [10], meant as the degree of distortion introduced by the genotype-phenotype mapping. Put in these terms, in GP genotypes (program code) map into phenotypes (program behavior) at low locality, so that even a small change of code can translate into huge difference in behavior.

This problem has been recently addressed in a systematic way using program *semantics*. The common feature of semantic-aware approaches in GP is consideration of the operational effects of code, which is usually realized by investigating the output values returned by programs or program fragments. Such semantic information can be exploited in different ways. It can be used as a means to design search operators with properties that are believed to be advantageous, e.g., crossover operators that swap only code fragments that have sufficiently similar effect [11]. But even more can be gained by noticing that semantic information induces certain geometric structure on the space of solutions and the fitness landscape [7, 4]. In this paper, following the recent studies on such operators [9], we propose a specific approximate realization of semantically geometric semantic crossover that exploits reversibility of instructions. This operator and its experimental validation are the main contributions of this study.

2. BACKGROUND

In GP, the objective is to evolve a program that produces desired output for a predefined set of program arguments, i.e. *inputs* (*examples*, *fitness cases*). The vector of desired outputs associated with particular examples forms the *target* for the search process. This process is driven by fitness function that usually aggregates the errors committed by a program on particular examples into single scalar value. Usually, fitness function is expressed as a distance between the vector of outputs produced by a program and the target, where the former one is known as *program semantics* (or sampling semantics) [1, 11]. The particular form of distance metrics depends on output data type and problem formulation: Euclidean or city-block distance are commonly used for real-valued programs (e.g., symbolic regression), and Hamming distance for Boolean domain. However, such a metric can also be defined for non-scalar data types, i.e., the Levenshtein distance for strings.

Formally, semantics $s(p)$ of program p is a vector of outcomes produced by p in a response to the given set of ex-

GECCO'13, July 6–10, 2013, Amsterdam, The Netherlands.

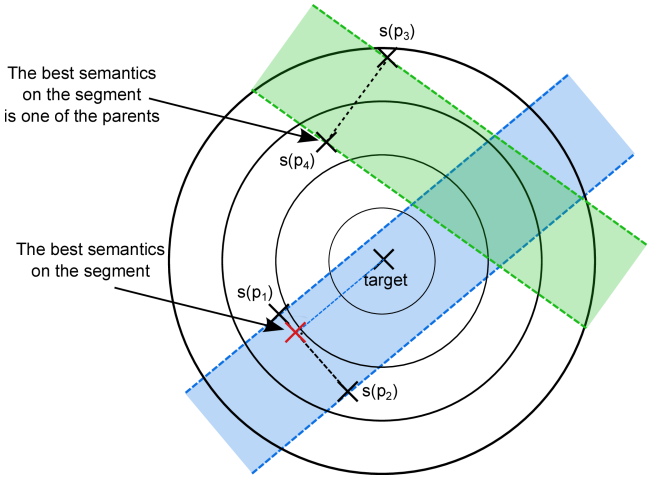

Figure 1: The conic fitness landscape defined by fitness function expressed as Euclidean distance from the given target semantics. The segments between parents show the range of semantics of geometric offspring. For p_1 and p_2 there exists a geometric offspring that is better than the best of the parents. For p_3 and p_4 there is no better offspring. It is the presence or absence of the target in the shaded area that distinguishes these cases.

amples. For instance, if p is a symbolic regression program tested on n fitness cases, its semantic $s(p)$ is the vector of n numbers it returns when applied to these cases. Denoting the above mentioned metric by d, the fitness f of a program p can be expressed as $f(p) = d(s(p), t)$, where t is the target. Importantly, the same metric can be used to measure the semantic proximity $d(s(p_1), s(p_2))$ of any pair of programs p_1 and p_2, and thus it endows the space of semantics (and indirectly the space of programs) with certain geometric structure.

An important consequence of defining fitness as a distance from the target t is that the fitness landscape spanning the space of all semantics is a cone centered at t. By the properties of metric, this landscape is unimodal, with the only one global minimum in t. If a program p is modified so that its semantics $s(p)$ becomes closer to the target t, the fitness $f(p)$ decreases (improves) too, because $f(p) = d(s(p), t)$. There is perfect correlation between f and the phenotypic distance d, because they are equivalent. If one could manipulate the semantics directly, the search on such landscapes would become trivial, with even the simplest local search algorithms guaranteed to find the target. However, typical search operators employed in GP operate only on program code, disregarding semantics, and such purely syntactic manipulations translate into changes of program semantics in a complex way.

The possibility of exploiting the properties of semantic fitness landscape propelled research on corresponding search operators. One of the most important concepts on this avenue of research is geometric crossover. According to Moraglio, a recombination operator is a geometric crossover under the metric d if all offspring are in the d-metric segment between its parents [8]. The offspring resulting from geometric crossover have certain attractive properties that we explain in the following for the special case of Euclidean metric.

Consider the space of programs represented by their semantics and the conical fitness landscape spanned over that space by the Euclidean fitness function (see Fig. 1). Choose any two programs from the space and cross them over. For the standard tree-swapping crossover, the semantics of the resulting offspring may be located almost anywhere on the cone, and that location is only weakly related to parents' semantics. However, the semantics of an offspring resulting from geometric crossover must lay somewhere on the segment between the parents, so its fitness must be not worse than the fitness of the worst of parents. Moreover, if it happens to be closer to the target than the semantics of the better parent, the offspring is more fit that both its parents.

This example illustrates that geometric offspring can outperform its parents. The main challenge for technical realization of such operators is that, in GP, there is no direct control on program semantics: the search operators change program code, and these changes translate into complex trajectories in the semantic space. Nevertheless, certain degree of 'geometricity' can be attained using simple means, e.g., by applying standard crossover multiple times and picking the offspring that is as geometric as possible [4]. However, the major recent advancement in studies on geometric crossover is [9], where Moraglio *et al.* have proposed a systematic way of designing *exact* mutation and crossover operators that exploit the conic shape of the semantic fitness landscape (geometric semantic genetic programming, GSGP). The key idea there is to use the instructions available to the GP search process to build an expression atop the parent(s) that modifies its semantics in a way that *guarantees* preserving of the geometric properties. In this way, a mutated program is guaranteed to be semantically similar to the parent, and crossover's offspring has to be located on the segment between the semantics of the parents. For instance, the semantically geometric crossover applied to a pair of symbolic regression parent programs (p_1, p_2) produces an offspring program of the form $\alpha p_1 + (1 - \alpha)p_2$, where α is a constant. It is immediately obvious that such an offspring, being a linear combination of p_1 and p_2, has semantics located on the Euclidean segment connecting their semantics.

However, as Moraglio *et al.* report [9], bloat remains the main challenge for GSGP. Because the offspring solutions are syntactic aggregates of entire parent programs (offspring include parents as subprograms), every generation of offspring involves more code, and additional simplification is necessary to make the search process technically feasible. And simplification procedures incur also additional computational overhead. In this paper, we make an attempt to circumvent this problem by proposing an approximate geometric crossover operator. This operator, described in the subsequent section, does not guarantee to produce perfectly geometric offspring, but it is also less affected by bloat.

3. APPROXIMATE GEOMETRIC CROSSOVER

The main idea behind Approximate Geometric Crossover (AGX) is to replace subtrees in parents with such code fragments (subtrees) that the semantics of offspring lays in the middle of the segment connecting parents' semantics. The operator first calculates the semantics $s(p_1)$ and $s(p_2)$ of, respectively, parents p_1 and p_2[1]. Next, the midpoint m on the segment connecting $s(p_1)$ and $s(p_2)$ is determined. For

[1]If the programs have been previously evaluated, their semantics are already known.

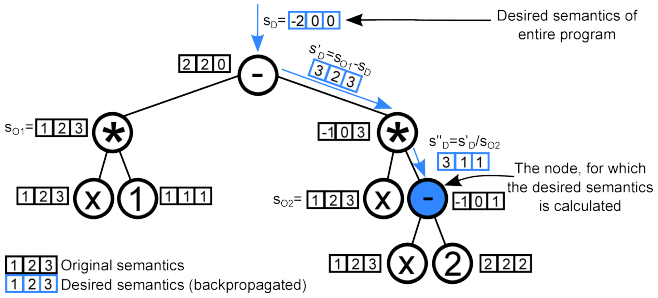

Figure 2: An example of semantic backpropagation. For a given desired program semantics, we calculate the desired semantics for each node on path from the root node to the crossover point (the blue one).

numerical semantics and Euclidean distance, this is the average of semantics of both parents. The calculated vector m becomes the desired semantics of offspring, as it corresponds to a 'perfect mixing' of parents' behaviors.

Next, AGX chooses two crossover points in p_1 and p_2. The subsequent steps apply to each parent independently, so for simplicity from now we focus on p_1 only. Then the operator executes all instructions on the path from program root to the chosen crossover point in a 'reverse manner'. The input data for this process is the semantics m. Compared to regular program execution, the order of execution is reversed (execution proceeds top-down in the tree), and at each step of this process an 'inverse' instruction is executed.

The details of reverse execution can be found in Section 3.1, and here we explain its objective. Let us consider the first stage of this process, i.e., the root node r and its direct child c on the path defined above. Reverse execution determines what should be the output of c in order for r to produce m. In other words, we calculate the *desired semantics* of the subprogram to replace c (desired in the sense that it would make the resulting offspring a perfect mixture of the parents). This process, continued along the path, allows us to propagate this information to the instruction located at the crossover point, and thus determine the desired semantics of the subtree to be replaced.

Once the operator determined the desired semantics for crossover point, it searches the previously prepared library of short programs (called 'procedures') for a procedure, that has the closest semantics to the desired one (see Section 3.2). Finally, the selected procedure replaces entire subtree under the crossover point, producing the offspring. The same steps are applied to the other parent.

3.1 Semantic backpropagation

We call the algorithm that calculates the desired semantics of an intermediate node *semantic backpropagation*. By analogy to error backpropagation in neural networks, we apply the desired value of semantics to program's root node and then propagate this value reversely, down the program tree (Fig. 2). The algorithm needs to know the semantics of all nodes in the program (subtrees) before performing the backpropagation, but this requirement is already fulfilled if the program has been previously evaluated (which is the case in the standard GP loop).

To explain the details, let us focus on a single tree node. Given the desired semantics of that node (desired output) and the actual semantics of all children of that node, we want to calculate the desired value of semantics of a specific child of this node. To do that, we need to execute the inversion of actual instruction in the node in context of the other children and the desired semantics of current node.

For the instructions characteristic to symbolic regression (considered in the experimental part), inversion can be implemented by using a *complementary operation* that reverts the actions performed by the actual instruction. For instance, consider subtraction instruction $output := child_1 - child_2$. Since subtraction is an asymmetric operation, the form of complementary operation depends on which child we propagate the desired semantics to. For the first child the complementary operation is $child_1 := output + child_2$, and for the second child it is $child_2 := child_1 - output$. The values calculated in this way for all elements of the vector that defines the desired semantics ($output$) form the semantics (vector shown in blue in Fig. 2) to be propagated further down the tree.

As long as an instruction implements a bijective function (transforms distinct arguments to distinct values in the codomain and vice versa) it can be safely inverted in the above way. However if an instruction is surjective (and not bijective), its inversion is ambiguous, since there are many arguments that cause it to return the same output value. In general, two types of ambiguity are possible, with a finite and infinite number of inversions. In the former case one can calculate, provided enough computational resources, all possible inversions and continue the semantic backpropagation in child node for each of these semantics. However for the latter case, the set of possible inversions can be only sampled.

To cope with the above problems while keeping computational cost at bay, in case of ambiguity we calculate at most two desired semantics for the child node. The subsequent part of the semantic backpropagation is then carried out for both desired semantics independently. Each of them can possibly lead to further ambiguities. Thus, even with this constraint, the number of backpropagated semantics may increase exponentially with node depth.

Another special case needs to be handled when inverting non-surjective instructions. For instance, $\forall x : \exp(x) > 0$, so for negative values of desired semantics no argument x exists that could be propagated down the tree. In other words, there is no semantics for the appointed crossover point (and thus no procedure in the library) that can move the semantics of the entire program to the midpoint of the segment. If this happens even for a single element of desired semantics, we discard it. However, this does not necessarily stop the entire semantic backpropagation: if the ancestor nodes had ambiguous inversions and thus led to alternative versions of desired semantics, the processing continues for them. Only when inversion fails for all versions of desired semantics, the algorithm gives up.

Note that, an instruction can be both ambiguously invertible and non-invertible. This is the case for, e.g. $y := x^2$, which maps exactly two x values to every *positive* value of y and no x value to every *negative* value of y. Therefore, the feasibility of instruction inversion is highly contextual: it may be possible to unambiguously invert an instruction at one location in a program but not at another. The ruling factor is the desired semantics of the instruction being inverted. In this sense, we may say that the instruction's desired semantics specifies (one unique inversion), underspecifies (multiple inversions), or overspecifies (no inversions) the desired semantics of instruction's argument (the selected child node).

Table 1: The benchmarks used in the experiment. $E[a, b, n]$ means n *equidistant* points from the range $[a, b]$. $U[a, b, n]$ means n random points chosen with *uniform* distribution from the range $[a, b]$.

Problem	Definition (formula)	Training set Test set
Nonic	$\sum_{i=1}^{9} x^i$	$E[-1, 1, 20]$ $U[-1, 1, 20]$
R1	$(x+1)^3/(x^2-x+1)$	$E[-1, 1, 20]$ $U[-1, 1, 20]$
R2	$(x^5-3x^3+1)/(x^2+1)$	$E[-1, 1, 20]$ $U[-1, 1, 20]$
Nguyen-7	$log(x+1)+log(x^2+1)$	$E[0, 2, 20]$ $U[0, 2, 20]$
Keijzer-1	$0.3xsin(2\pi x)$	$E[-1, 1, 20]$ $U[-1, 1, 20]$
Keijzer-4	$x^3e^{-x}cos(x)sin(x)(sin^2(x)cos(x)-1)$	$E[0, 10, 20]$ $U[0, 10, 20]$

In conclusion, due to possible ambiguities and non-invertibility of instructions, the outcome of the semantic backpropagation is a *set* of desired semantics for the given tree node. That set may contain zero, one, or many semantics.

3.2 Library of procedures

AGX makes use of the library of short programs, called *procedures* in following. Its purpose is to supply AGX with code fragments that match as close as possible the desired semantics resulting from semantic backpropagation. To provide high diversity of semantic, we populate the library with all possible program trees, up to an assumed height, composed of the elements of the assumed instruction set. Since it is only the semantic distance between the desired semantics and the semantic of a procedure that determines which procedure is fetched from the library, storing semantically identical procedures is pointless. Therefore, for any set of procedures that have the same semantics, the library stores only the shortest one. There are obviously many other ways in which the library could be populated, but here we limit our considerations to this straightforward method.

The size of the library is exponential in function of the height limit, so providing fast search for the closest semantics is essential. This process can be sped up by means of spatial (multidimensional) indexes, data structures originally designed for geographic databases. We tested wide range of spatial indexes and found out that for this particular task, characterized by high dimensionality of the search space, the KD-tree [2] works best. The nearest neighbor query to the index consisting of m elements takes on average $O(\log m)$ time, and $O(m^{1-1/k})$ in the worst case.

4. GEOMETRY OF OPERATORS

In general, AGX cannot guarantee the semantics of offspring to reach the midpoint of the segment connecting parents' semantics. There are two reasons for this. Firstly, the library is a finite set of procedures, which by definition do not implement all possible semantics. Therefore, in most crossover acts, the substituted procedure has a non-zero distance from the desired semantics, and that discrepancy propagates to the root node. Secondly, limited arithmetic precision makes the reversal process inherently inexact.

For these reasons we carry out a computational experiment aimed at quantitative assessment of the probability at which different operators produce geometric offspring. Technically, we ran evolution on six symbolic regression problems from Table 1 for 100 generations each and recorded each crossover act by AGX, standard subtree crossover (GPX) [3], and a comparable approach, locally geometric semantic crossover (LGX, [5]), that approximates geometric changes on the level of homologous program tree loci. Next, for each resulting offspring we checked whether it is located on the segment connecting parent's semantics according to the Manhattan distance. We chose this metric because it is characterized by the biggest size of segment between parents of all Minkowski distances $L_p, p \geq 1$. In other words, Manhattan distance defines geometric offspring in the most 'liberal' way.

Table 2: Fraction of geometric offspring (according to Manhattan distance) bred by AGX, LGX, GPX, and p-values of Z-test for the equality of two proportions. Best values in rows and the significant p-values ($\alpha = 0.05$, one-sided test) are in bold.

Depth of crossover	AGX	LGX	GPX	p-values		
				AGX>GPX	LGX>GPX	AGX>LGX
1	.0155	**.1676**	.0035	**.0000**	**.0000**	1.000
2	**.0151**	.0100	.0031	**.0000**	**.0000**	**.0000**
3	**.0136**	.0031	.0018	**.0000**	**.0001**	**.0000**
4	**.0105**	.0016	.0020	**.0000**	.9570	**.0000**
5	**.0055**	.0014	.0011	**.0000**	.1219	**.0000**
6	**.0028**	.0009	.0007	**.0000**	.1521	**.0000**
7	**.0017**	.0006	.0005	**.0000**	.2899	**.0000**
8	**.0012**	.0004	.0003	**.0000**	.1944	**.0000**
9	**.0010**	.0007	.0003	**.0001**	**.0121**	.0558
10	**.0006**	.0005	.0003	**.0296**	.0766	.3660
11	**.0005**	.0002	.0003	.1214	.8050	.0529
12	**.0004**	.0001	.0003	.2305	.9037	**.0499**
13	**.0003**	.0002	.0002	.3577	.6258	.3114
14	.0002	.0000	**.0005**	.9526	.8704	.2298
15	.0000	.0000	**.0002**	.6843	.6582	.5000
16	.0000	.0000	**.0005**	.5979	.6968	.5000
17	.0000	.0000	.0000	.5000	.5000	.5000
Overall	**.0057**	.0035	.0008	**.0000**	**.0000**	**.0000**

Table 2 presents the probability of producing geometric offspring by all considered operators and the corresponding p-values of Z-test for the equality of two proportions. We present the overall outcome in the last row of the table, and for particular depths of crossover points. Each offspring is counted separately, thus two offspring resulting from the same crossover act may contribute to different rows in the table.

From the table, we see that in 12 of 17 cases AGX has the highest probability of producing geometric offspring. LGX dominates both other operators only in one case. GPX is most geometric in three cases, however the probabilities are very small and insignificant. AGX is significantly more geometric than GPX in 10 of 17 cases, while LGX is significantly better in only 4 cases. Taking into account overall result, both AGX and LGX are significantly more geometric than GPX, however the test also shows that AGX is significantly more likely to produce geometric offspring than LGX. No crossover act at depth 17 was geometric in this experiment.

The important observation is that, while the probability of producing geometric offspring drops with depth for all operators, AGX maintains the relatively highest probability for the widest range of depths. For AGX, the probability

Table 3: Experimental setup.

Parameter	AGX	LGX	GPX
Instruction set	$\{+, -, \times, /, sin, cos, exp, log, x\}$[a]		
Population size	1024		
Initial max tree depth	6		
Max tree height	17		
Selection	tournament selection		
Tournament size	7		
Trials per experiment	100 independent runs		
Termination condition	250 generations		
Crossover probability	0.9		
Mutation probability	0.1		
Max procedure height	$\{3, 4\}$	$\{3, 4\}$	—
Crossover point selection	linear	homologous Koza-I [3]	

[a]The / and log are protected. Division by 0 returns 0 and logarithm calculates absolute value of its argument.

decreases due to the growing number of instructions in the path from crossover point to the program root, causing multiple transformations of geometry of semantic space. Since the semantics of the procedure chosen from the library in general only approximates the desired semantics, the error of this approximation is propagated through the program structure, where it may be increased or decreased depending on instructions on the path. This complex process may affect each fitness case to a different extent, and apparently causes the changes on the level of program root to be 'less geometric'.

In conclusion, we observe that AGX is more likely to produce geometric offspring than LGX and GPX, however the probabilities are rather low. In the next section, we verify whether these properties have any significant impact on the effectiveness of evolutionary search.

5. PERFORMANCE OF OPERATORS

We use six univariate symbolic regression problems to assess the performance of AGX, comparing to the LGX [5] and subtree crossover (GPX) [3]. The problems, enumerated in Table 1, come from [3, 6] and represent four classes. There are problems with the target defined by a polynomial, rational, logarithmic and trigonometric functions. For each problem, we define a training set of fitness cases used by fitness function during evolution, and a disjoint test set of fitness cases for post-evolution assessment of generalization capability.

Evolution is allowed to use eight non-terminal instructions (see Table 3). The only terminal instruction is the independent variable x (there are no constants). Crossover operators are engaged with 0.9 probability and are supplemented by subtree-replacement mutation. Mutation is added to prevent premature convergence and to make competition with GPX more fair for the latter (without mutation, evolution run by GPX can irreversibly lose instructions from population). There is no reproduction.

AGX and LGX employ the same libraries of procedures. We consider two libraries: a small one containing all unique program trees of height up to 3, and a big one with height limit 4. The cardinalities of libraries are, respectively, 212 and 108520 (recall that semantic duplicates are discarded).

Table 4: Symmetry test. P-values represent probability of *erroneously* judging method in row as better than method in column. Values in bold are significant ($\alpha = 0.05$).

	AGX_3	AGX_4	GPX	LGX_3	LGX_4
AGX_3					
AGX_4	**0.002**		**0.048**	0.123	0.996
GPX	0.892				
LGX_3	0.705		0.996		
LGX_4	**0.009**		0.123	0.262	

The suffix following method name indicates the type of library (e.g. AGX_4).

Note that the operators use different methods for selection of crossover point. AGX uses linear selection, which assigns equal probability to each depth of tree. First the set of nodes on particular depth is chosen, then a crossover point is selected from this set. On the other hand LGX involves homologous selection and GPX uses Koza-style selection (with 0.9 probability an intermediate node is chosen, otherwise a leaf).

5.1 Search progress

Figure 3 shows the average of best-of-generation fitness achieved by AGX_3, AGX_4, LGX_3, LGX_4 and GPX for problems from Table 1, averaged over 100 runs, with confidence intervals marked by shading. AGX_4 is the unquestionable winner in terms of speed of convergence. On average, it leads from the beginning, and in less than 50 generations achieves fitness that is unbeatable by the other methods for long time. Only for two problems, Nonic and Nguyen-7, AGX_4 is overtaken by LGX_4 at the end of trial. However, these differences are statistically insignificant. The second best method is LGX_4, which achieves fitness at the level of AGX_4 or a bit worse, however it requires significantly more time to do so.

The situation is quite different for the methods equipped with the small library. AGX_3 is better than LGX_3 only on the R2 problem, and in the other cases the latter is noticeably better. We hypothesize that it is an effect of imperfect match of procedures to desired semantics. The big library is three orders of magnitude larger than the small one, therefore it can be expected to provide procedures that match the desired semantics more closely.

The relation of GPX to other methods is highly problem-dependent. It is significantly worse than the best of the other methods. However, for each problem there is at least one other method that does not outperform it. Usually it is the AGX_3, but for R2 it is LGX_3.

We performed Friedman's test for multiple achievements of series of subjects on the average of best-of-run fitness to assess significance of the differences between methods. The resulting p-value is 0.0024, so assuming $\alpha = 0.05$ there exists at least one statistically significant difference. Therefore we conducted post-hoc analysis using symmetry test to analyze significance between particular pairs of methods. The results, shown in Table 4, indicate that AGX_4 is significantly better than GPX and AGX_3.

5.2 Test-set performance

To assess the performance of generalization, we ran best-of-run individual on independent test-set data (Tab. 1) and averaged the results over 100 runs of GP. The values shown

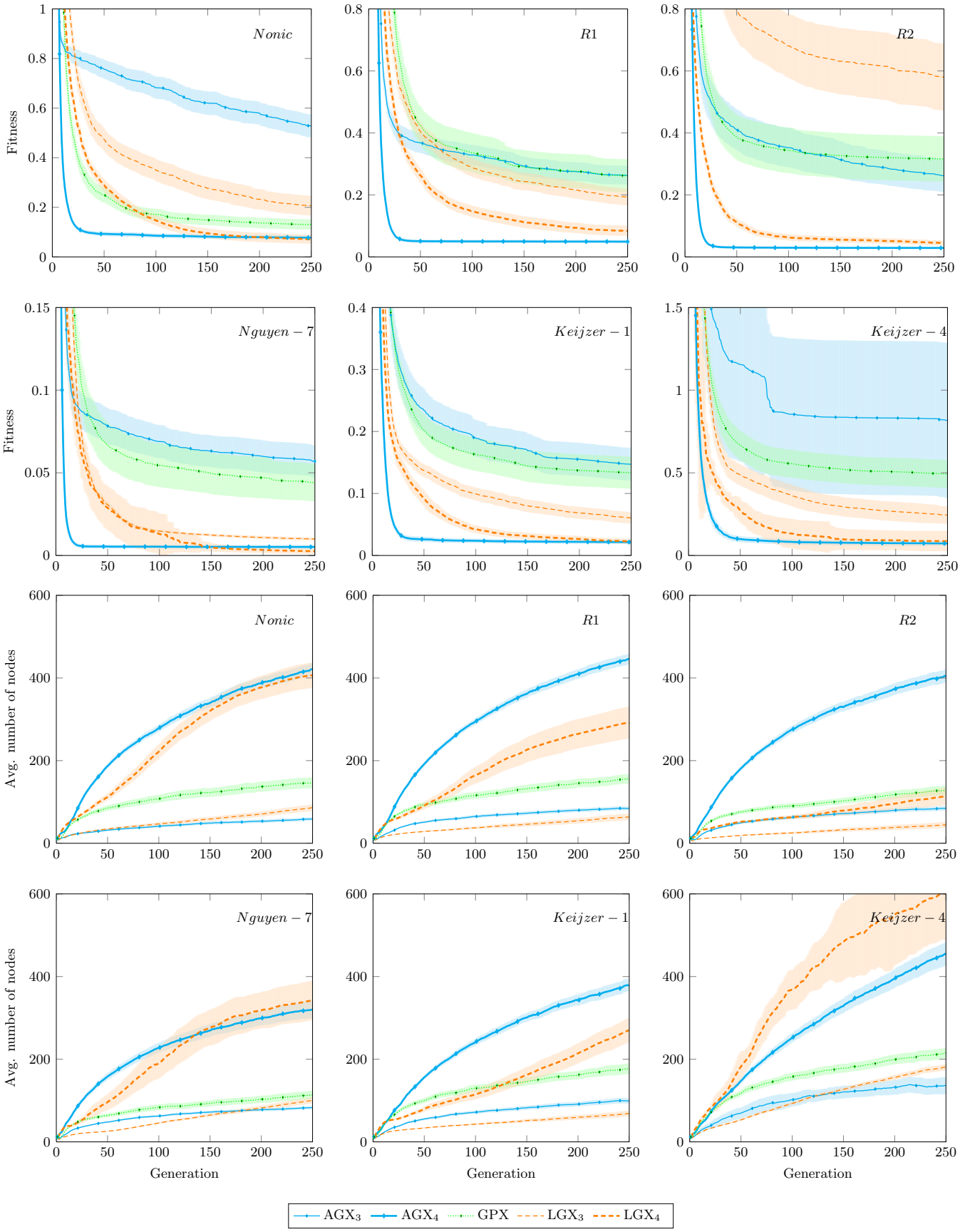

Figure 3: Performance of AGX, LGX and GPX. Rows 1 & 2: Best-of-run fitness. Rows 3 & 4: Number of nodes in programs. Values averaged over 100 runs.

Table 5: Error committed by best-of-run individual on test set (avg. of 100 runs). For values higher than 100 we left only an order of magnitude.

Problem	AGX_3	AGX_4	GPX	LGX_3	LGX_4
Nonic	0.359 ±0.036	**0.093** ±0.012	0.130 ±0.021	0.201 ±0.037	0.191 ±0.074
R1	0.224 ±0.025	**0.050** ±0.006	0.261 ±0.053	0.167 ±0.025	0.103 ±0.018
R2	$10^7 \pm 10^7$	**0.028** ±0.004	0.316 ±0.073	0.621 ±0.126	0.042 ±0.012
Nguyen-7	0.051 ±0.008	0.005 ±0.001	0.044 ±0.011	0.018 ±0.015	**0.004** ±0.003
Keijzer-1	0.190 ±0.032	**0.039** ±0.007	0.134 ±0.026	0.091 ±0.013	0.041 ±0.010
Keijzer-4	3.113 ±2.792	$10^{13} \pm 10^{13}$	**0.492** ±0.084	2.008 ±1.141	2.854 ±3.278

in the Table 5 demonstrate that AGX_4 generalizes the best for 4 out of 6 problems. In contrast, LGX_4 provides the best generalization for only one problem, however the difference compared to AGX_4 is insignificant in this case. GPX proves superior to other methods only on one problem.

Comparison of the errors obtained for the test set to the corresponding fitness values measured on the training set (Fig. 3) show that AGX_4 noticeably overfits only on the Keijzer-4 problem. On the other hand, there are no doubts that overfitting occurs also twice for AGX, once for LGX_3 and LGX_4. GPX turns out to be the most resistant to overfitting, which is however not surprising given its poor performance on the training set.

5.3 Impact on tree size

Figure 3 presents the average number of nodes of programs in evolutionary runs for AGX, LGX and GPX. The very first observation is that the size of solutions created by both AGX and LGX depends on the size of used library. Both methods, when equipped with the small library, produce solutions that are noticeably smaller than the solutions created by standard tree-swappinng crossover. With the big library, both produce significantly bigger trees.

Though the trees produced by AGX_4 at the end of runs are rather large, it is worth noticing that the method achieves its top fitness much earlier, typically in less than 50 generations (Fig. 3), and the remaining ~ 200 generations do not noticeably improve fitness. And when we consider the programs generated by AGX_4 in the 50th generation, their sizes can be claimed comparable to those evolved by, e.g., GPX.

At this point it is also desirable to compare AGX, an *approximate* geometric crossover, to the *exact* geometric crossover from [9]. The exact geometric crossover combines two parent programs into one offspring, by adding an additional structure as a new program root. Such a structure must consist of at least one node. Thus, the exact crossover on average doubles the size of programs in every generation. The rough estimate of the expected number of nodes in programs produced by this operator after g generations is $2^g \times avg_{p \in P_0}(size(p))$, where P_0 is the initial population. Even if P_0 contained single-node programs, for $g = 50$ this amounts to 1.1×10^{15}. Though the authors of [9] do not report tree sizes and claim that bloat can be kept at bay by employing simplification procedures, this analysis shows that the sizes of programs produced by AGX_4 when fitness saturates (~ 170 nodes on average) can be still attractive. In other words, compared to exact geometric crossover, AGX trades exactness for program size.

6. DISCUSSION

An observant reader has probably noticed the following. If the objective of semantic backpropagation is to 'shift' parent's semantic from its current location to specific other location (in our case: the midpoint of the segment spanning parents' semantics), then why not use the same procedure to shift the semantics *directly* to the target of the search process? In such a case, the semantic being backpropagated through the path of instructions would be simply the ultimate goal of the entire search process. Assuming that the semantic backpropagation is at least partially effective at changing the semantic in the right direction (and the above experiments suggest that it is), one may expect such search converge to the target faster than AGX.

Indeed, such *direct search* is a natural analog to AGX, and we investigate it in a separate study [12]. Note however that applicability of direct search is limited. Firstly, it requires the target to be explicitly given. This requirement is met in all typical symbolic regression tasks. However, there are conceivable scenarios in which this is not the case. Imagine for instance an application in which the target cannot be revealed due to privacy concerns or other issues (e.g., the target is a confidential historical time series). In such a case, even if the fitness function definition is fully conformant with the formalisms presented in Section 2, and fitness landscape is a unimodal cone, the target cannot be directly used. Contrary to direct search, AGX, for which the knowledge of target is irrelevant, can be still applied to such tasks.

There is also a much wider class of problems that can be approached with AGX, but not with the direct search. In this study, we assumed that fitness function is a *distance* from a *uniquely defined* target, which causes the fitness landscape to be conic and guarantees the offspring of geometric crossover to be not worse than the worse of its parents (Fig. 1). Many problems involve fitness function that cannot be expressed in that way, e.g., where fitness results from some simulation process (control problems like pole balancing, design problems, etc.). Consequently, unimodality is lost (or uncertain), and the above guarantee becomes void. Nevertheless, it is reasonable to expect that fitness landscapes for such problems are at least locally convex, and within such local basins AGX can still be effective at improve solutions (while, again, direct search is useless as the target is unknown). More specifically, as long as both parents dwell in the same basin of local optimum, AGX's operation is consistent with the reasoning presented in Section 2. Its effectiveness on such problems will be subject of our future studies.

Probably the strongest limitation of the approach proposed here is the requirement of instruction inversibility. Definitely, many instructions are in general irreversible, particularly those used in conventional programming languages. Nevertheless, we hypothesize that AGX can bring some benefits even if the inversion operations are imperfect in some sense (e.g., return multiple possible outcomes or do not return any outcome). Note that the library typically does not contain the procedure with exactly the desired semantics,

so even if all instructions on the path would be perfectly inversible, the semantics of the modified individual is not guaranteed to shift to the prescribed location. Also, in the end, given the stochasticity of the other elements of evolutionary search (selection, mutation), the approximate nature of AGX can be of secondary importance.

In conclusion, the performance of AGX is clearly related to the contents of the provided library. If we could hypothetically equip AGX with an ideal library, that provides a perfectly matching procedure to the desired semantics, the AGX would operate almost identically on semantics level to the exact geometric crossover [9]. However we expect that the sizes of trees produced by AGX would remain smaller than exact crossover's. Note that the ideal library is virtually not possible to be built. Additionally one may consider such library as a form of an oracle, which returns a program for a given semantics. It is clear that having this form of an oracle, we need not any optimization algorithm anymore.

7. CONCLUSIONS

We presented AGX, a new GP crossover operator that is semantically approximately geometric. AGX exhibits superior performance compared to the locally geometric semantic crossover [5], and the canonic tree-swapping crossover [3]. It maintains this advantage on both training set and test set. Though it suffers from substantial bloat, particularly when allowed to run too long, the sizes of trees produced by AGX are significantly smaller than the theoretical expected tree size of non-simplified programs produced by exact geometric crossover [9].

Acknowledgment

Work supported by grant no. DEC-2011/01/B/ST6/07318.

8. REFERENCES

[1] L. Beadle and C. Johnson. Semantically driven crossover in genetic programming. In J. Wang, editor, *Proceedings of the IEEE World Congress on Computational Intelligence*, pages 111–116, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.

[2] Bentley. Multidimensional binary search trees used for associative searching. *CACM: Communications of the ACM*, 18, 1975.

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

[4] K. Krawiec and P. Lichocki. Approximating geometric crossover in semantic space. In G. Raidl, et al., editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 987–994, Montreal, 8-12 July 2009. ACM.

[5] K. Krawiec and T. Pawlak. Locally geometric semantic crossover: a study on the roles of semantics and homology in recombination operators. *Genetic Programming and Evolvable Machines*, 14(1):31–63, 2013.

[6] J. McDermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong, and U.-M. O'Reilly. Genetic programming needs better benchmarks. In T. Soule, et al., editors, *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 791–798, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[7] A. Moraglio. *Towards a Geometric Unification of Evolutionary Algorithms*. PhD thesis, Department of Computer Science, University of Essex, UK, Nov. 2007.

[8] A. Moraglio. Abstract convex evolutionary search. In H.-G. Beyer and W. B. Langdon, editors, *Foundations of Genetic Algorithms*, pages 151–162, Schwarzenberg, Austria, 5-9 Jan. 2011. ACM.

[9] A. Moraglio, K. Krawiec, and C. G. Johnson. Geometric semantic genetic programming. In C. A. Coello Coello, et al., editors, *Parallel Problem Solving from Nature, PPSN XII (part 1)*, volume 7491 of *Lecture Notes in Computer Science*, pages 21–31, Taormina, Italy, Sept. 1-5 2012. Springer.

[10] F. Rothlauf. *Representations for genetic and evolutionary algorithms*. Springer, pub-SV:adr, second edition, 2006. First published 2002, 2nd edition available electronically.

[11] N. Q. Uy, N. X. Hoai, M. O'Neill, R. I. McKay, and E. Galvan-Lopez. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines*, 12(2):91–119, June 2011.

[12] B. Wieloch and K. Krawiec. Running programs backwards: Instruction Inversion for Effective Search in Semantic Spaces. In Christian Blum, et al., editor, *GECCO '13: Proceedings of the 15th annual conference on Genetic and evolutionary computation*, Amsterdam, The Netherlands, 2013. ACM.