ABSTRACT
We propose a crossover operator for genetic programming that is approximately semantically geometric on the level of homologous program fragments. For a pair of corresponding loci in the parent solutions, the operator finds a semantically intermediate procedure from a library prepared prior to evolutionary run, and inserts it at that location, creating so the offspring. When tested on six symbolic regression problems, this approach, termed locally geometric semantic crossover (LGX), leads to significant increase in search performance (best-of-run fitness) when compared to standard tree-swapping crossover and several other control methods. In addition, the programs produced by LGX turn out to be superior on the test set of fitness cases. This holds also when the same amount of processing time is given to compared methods, despite the fact that LGX requires some extra computational overhead. These results indicate that search operators that recombine homologous program fragments in a semantically aware way constitute an attractive alternative to other approaches. We also hypothesize that, thanks to their properties, they can be useful tools for exploiting modular tasks.

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

Keywords
genetic programming, program semantics, semantic crossover

1. INTRODUCTION
In a broad sense, program is a sequence of symbols (instructions), where each symbol has a particular, given a priori, semantics. The semantics of an instruction determines its effect: how, given the input, its output should be determined (computed). A human programmer familiar with programming language knows that semantics, and can use it to, e.g., anticipate the combined behavior of two concatenated instructions or substituting one instruction with another. However, the algorithms considered in genetic programming (GP) have no access to such information. From their perspective, program is a purely symbolic structure, where opcodes associated with particular instructions have no particular meaning.

On one hand, this is consistent with the evolutionary aspect of GP: in the end, natural evolution does not ‘know’ the phenotypic expression of genes. On the other hand, the knowledge of semantics is definitely one of factors that makes human programming so effective. Therefore, equipping GP algorithms in some semantic-aware extensions could lead to substantial progress in automated programming, and this opportunity attracted notable interest in recent GP research [5, 9, 3].

In this paper we devise a method that makes GP alert to certain semantic aspects of the programs it operates on. We achieve this by designing a crossover operator that, for a pair of procedures (subtrees) located in parent programs, finds a procedure that is approximately intermediate in semantic sense. Such procedure replaces the selected subtrees in parent programs. Importantly, we implement this process in a computationally effective way, so that its burden on the overall cost of evolutionary search can be outweighed by better search efficiency.

2. THE METHOD
The proposed approach exploits the compositional character of programs, by which we mean that not only complete programs, but also parts of programs and program conglomerates have valid interpretation in many programming languages. In particular, we assume that any aggregate (e.g., concatenation) of two or more programs is a correct program itself and can be run on any input data that would be accepted by its constituent components. This is valid for most application areas of GP (e.g., symbolic regression, logical function synthesis).

We also assume that the fitness function captures the divergence between the program output and some known desired output. This is consistent with GP standards, where individuals are usually tested on a set of fitness cases, and fitness is some form of error built upon the outcome of these tests. Formally, we assume that a metric ||| is given that
calculates such error (absolute deviation in experimental part). Importantly, the same metric can be also used to measure the semantic similarity of pairs of programs, and this possibility is a key element of the proposed approach.

The essential consequence of this assumption is that fitness landscape is a convex surface spanned over the space of vectors that hold program outputs. Convexity allows designing recombination operators that are likely to yield offspring of good quality (e.g., topological crossover [1]). This is easy to demonstrate for Euclidean metric and space: given a point \( x \) in such space, which corresponds to the desired output of a program, and a pair of points \( x_1, x_2 \) representing parent solutions, any point on the segment connecting \( x_1 \) with \( x_2 \) cannot be more distant from \( x \) than \( \max(||x_1 - x||, ||x_2 - x||) \).

The proposed method aims as exploiting this property locally, i.e., on the level of program fragments, rather than entire programs. It operates in two major phases. Prior to evolutionary run, it creates a library of short programs, calculates their semantics, and builds upon them an indexing structure for fast access. Then, during an evolutionary run, the library is used by the crossover operator to modify the fragments of parents’ programs in a semantically aware way. Below we detail these components. We explain the approach and run experiments using Koza-style type-free programs represented as trees [2], with leaves feeding in program input, and the root node returning the output of a program. However, it can be ported to other program representations.

### 2.1 Building the library of procedures

The input to the method is the set of instructions \( I \), each of them being an operator of arbitrary arity. The first step consists in creating from \( I \) a library \( L \) of short programs, called procedures in following. The purpose of the library is to provide the crossover operator described in Section 2.3 with a semantically diverse sample of code fragments. The choice of procedures to be gathered in \( L \) could be done along different criteria, but here, for simplicity, we fill \( L \) with all trees of height at most \( h \).

Next, we calculate the semantics \( s(p) \) of every procedure \( p \in L \). Throughout this paper, by semantics we mean the vector of outcomes produced by program for the sample of inputs (fitness cases).

Any two procedures \( p_1, p_2 \) that have the same semantics \( ||s(p_1), s(p_2)|| = 0 \) do the same thing, which is redundant from the viewpoint of the method. Therefore, we discard from \( L \) procedures that duplicate the semantics of other procedures. For each subset of procedures that have the same semantics, only the shortest one is stored in \( L \).

### 2.2 Indexing the library

Once we have generated all procedures of height at most \( h \), computed their semantics, and filtered out the semantic duplicates, we are ready to construct a helper indexing structure for it.

As stated before, the semantics \( s(p) \) of a procedure \( p \) is a vector of \( d \) numbers, a point in a \( d \)-dimensional space. Locations of these points reflect semantic differences between procedures. Semantically similar procedures are located close to each other, while the very different ones occupy distant positions. In general, this space is sparse, since the output values returned by procedures can have arbitrary ranges, while the number of procedures to be stored in the library is rather low.

As it will become clear in the next section, we need a way to efficiently search this space for procedures that are as close as possible to an arbitrarily selected point. In order to provide this functionality, we employ spatial index, a data structure designed for geographic and spatial databases. From multiple variants of spatial indexes, we have chosen R-tree [1], because of its flexibility in representation of shape and position, and ability to work under arbitrary chosen norm and dimensionality. An R-tree splits a set of objects into hierarchical groups. The locations of points in each group determine its minimum bounding rectangle. R-trees support many kinds of queries, like shape containment, intersection and nearest-neighbor queries, of which we employ the last ones.

### 2.3 Locally geometric semantic crossover

After the library is built and equipped with an R-tree index, a GP run is launched. It proceeds as a regular GP, except for employing a specialized homologous crossover operator, which we term locally geometric crossover (LGX). Given two parent programs \( p_1 \) and \( p_2 \), LGX first identifies the structurally common subtree, i.e., the set of tree node locations that occur in both parents. The common subtree can be considered as an intersection of the parents, however, with an important remark that the op codes are ignored – only the tree structure matters. The subtree can embrace at most all locations in both parents, but typically it is smaller.

In the next step, LGX selects a single random location in the common subtree, intended to serve as crossover point. This choice follows the same rules as in the canonic Koza-style GP [2], with probability 0.1 the location is a randomly selected leaf of the common subtree, and with probability 0.9 an internal node is selected. Such distribution is intended to reduce bloat, as otherwise most crossovers would take place in leaves.

Subsequently, LGX identifies the subtrees in \( p_1 \) and \( p_2 \) rooted in the drawn location; let us denote them as \( p_1' \) and \( p_2' \), respectively. In consistency with the assumptions formulated at the beginning of Section 2, such subtrees are independent programs that can be executed. This allows us to calculate their semantics (vectors of responses to fitness cases), \( s(p_1') \) and \( s(p_2') \), and determine the midpoint between them in the semantic space:

\[
    s_m = \frac{s(p_1') + s(p_2')}{2}
\]

This point represents the semantics of a hypothetical procedure \( p : s_m = s(p) \) which, when inserted into parents at the appointed location, would make the resulting offspring programs semantically intermediate (medial) at the point of crossover (and, as we argue later, can make the entire offspring programs semantically more similar). However, finding \( p \) would in general require solving an inverse problem \( p = s^{-1}(s_m) \), which is a separate program induction problem in itself. Also, as \( s_m \) is a combination of semantics of two, potentially big trees, it does not have represent any program available within the assumed program space.

This is where the library comes at help. Instead of looking for a procedure whose semantics is exactly \( s_m \), we find in \( L \)
the procedure that is semantically most similar to \( s_m \), i.e.:

\[
p = \arg \min_{p' \in L} ||s(p') - s_m||
\]  

(2)

The procedure \( p \) replaces then the subtrees \( p'_1 \) and \( p'_2 \) in the parent solutions, which so become the offspring. This steps concludes the crossover act.

Most steps of the above algorithm require little computational expense. The major burden is in the search for the semantically most similar procedure (Eq. 2). To reduce this cost, we employ the R-tree structure described in the previous section.

### 2.4 Properties of the approach

An important property of the proposed approach is completeness. As the library contains representatives of all semantic equivalence classes obtainable from procedures of depth up to \( h \), LGX can produce any tree. The search space is not constrained.

There is obviously a notable computational overhead when compared to the standard approach, i.e., GP. This overhead is determined by a sum of:

1. The time required to prepare the library and build an R-tree,
2. The time of querying the R-tree in LGX (Section 2.3).

The total time of the former is technically the sum of the times required to generate the program, run them on the fitness cases to calculate program semantics, eliminating the semantic duplicates, and building the R-tree. This cost depends on the number of fitness cases and the number of procedures to be stored in the library, which in our case is a function of \( h \). For large \( h \), it can be substantial, thus, to keep the computational cost at bay, we \( h = 3 \) and \( h = 4 \).

Let us also note that the library depends only on the assumed set of instructions and choice of fitness cases. Therefore, it can be reused to solve multiple problem instances (optimize different fitness functions) that share the same instruction set, which is frequent in practice (e.g., symbolic regression, Boolean function synthesis).

The extra computational overhead of the second component, when compared to GP, is technically the cost of querying the R-tree for the nearest neighbor of given semantics. According to [8], the worst-case complexity is linear with respect to the number of objects (here: library size \( |L| \)), but usually the query time is significantly lower.

### 2.5 Related and similar approaches

There are several contributions in the past GP literature that the approach presented here has something common with. They can be grouped according to the two main features: use of a library and semantically-aware modification of solutions. Concerning the former, LGX can be likened to, e.g., run transferable libraries [10], which are repositories of program fragments intended to be used across multiple GP runs applied to different problem instances.

Concerning the latter, although the research on semantic in GP intensified only recently, there are a few relevant contributions in that respect. McPhee et al. were probably one of the first to study the impact of crossover on program semantics and so-called semantic building blocks [5]. In [6], Moraglio et al. considered properties of semantic spaces for different metrics and provided guidelines for designing semantically geometric crossovers. The semantically-aware crossover by Quang et al. [9] swaps a pair of subtrees in parent solutions that have similar, yet not too similar, semantics.

In the context of these contributions, LGX remains unique in combining three elements: (i) directing the choice of program fragments according to their semantic properties, (ii) homologous character of crossover, and (iii) use of a library of previously prepared procedures.

### 3. THE EXPERIMENT

The main objective of the experiment is to verify whether the semantic properties of the proposed crossover operator (LGX) influence the efficiency of evolutionary search. Our experimental framework is univariate symbolic regression, with the set of instructions \{+, −, ×, / \} and one terminal, the independent variable \( x \). Semantics is defined as the vector of values returned by a program for 20 fitness cases distributed equidistantly in the interval \([-1, 1]\).

We consider two libraries, for the maximum procedure height \( h = 3 \) and \( h = 4 \). For the former, there are 81 procedures (trees of height up to 3 composed of the four assumed instructions), but only 38 of them are semantically distinct, so \(|L| = 38\). For \( h = 4 \), these figures amount to 2185 and 1697, respectively.

We confront our approach with two types of control setups. The first of them is the standard Koza-style GP [2], which involves conventional tree-swapping crossover that uses the same probability distribution as LGX for node selection (0.1 for leaves and 0.9 for internal nodes). Like all other operators considered here, it never replaces the root node.

The latter, called RX (random crossover), is intended to verify whether the results to be observed are due to the geometric character of performed crossover, or potentially due to some other side effects. To certain extent, RX operates as LGX (Section 2.3), i.e., it replaces the parts of parent trees at homologous loci with procedures from the library. However, its choice of procedure from \( L \) is purely random. RX is thus similar to our approach in terms of mode of operation, but it is completely blind to the structure of semantic space.

To sum up, there are 5 setups in total: standard GP, and RX and LGX for \( h = 3 \) and \( h = 4 \). They will be referred to as GP, RX3, RX4, LGX3, and LGX4. All algorithms evolve a population of 1024 individuals.

### 3.1 Test problems and fitness assessment

We solve six univariate symbolic regression problems shown in Table 1, three polynomials and three rational functions. For each configuration, 150 runs are carried out, each starting from different initial population and lasting for 250 generations. Fitness is minimized and defined as the absolute

<table>
<thead>
<tr>
<th>Problem Definition (formula)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sectic ( x^6 - 2x^3 + x^2 )</td>
<td></td>
</tr>
<tr>
<td>Septic ( x^7 - 2x^6 + x^5 - x^4 + x^3 - 2x^2 + x )</td>
<td></td>
</tr>
<tr>
<td>Nonic ( x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x )</td>
<td></td>
</tr>
<tr>
<td>R1 ( (x + 1)^3/(x^2 - x + 1) )</td>
<td></td>
</tr>
<tr>
<td>R2 ( (x^5 - 3x^3 + 1)/(x^2 + 1) )</td>
<td></td>
</tr>
<tr>
<td>R3 ( (x^6 + x^5)/(x^4 + x^3 + x^2 + x + 1) )</td>
<td></td>
</tr>
</tbody>
</table>
error of the output produced by an individual w.r.t. the desired output, summed for the 20 fitness cases. The selection method is tournament of size 7, crossover performs with likelihood 0.9, and reproduction with likelihood 0.1. There is no mutation operator involved. Other parameters are set to defaults used in the ECJ package, which served as our experimental framework [1].

### 3.2 Search progress

The charts in Figure 1 present the best-of-generation fitness graphs for evolutionary runs, averaged over 150 runs. Each chart refers to single problem and plots data series for GP, RX₃, RX₄, LGX₃, and LGX₄. Gray shading marks 0.95-confidence intervals.

The first observation evident in the plots is that LGX₃ is an unquestionable winner in terms of speed of convergence. The mean fitness of the best-of-generation individual for this setup improves at the highest rate for all considered benchmark problems.

LGX₃ makes much slower progress. This may be explained by the fact that the library it uses is almost two orders of magnitude smaller than that of LGX₄ (38 vs. 1697 procedures). As a consequence, the semantic diversity of the procedures inserted into offspring (the number of unique semantic) is here much lower, which deteriorates the algorithm’s ability to perform effective exploration.

The random crossovers, RX₃ and RX₄, fare surprisingly well. For a few problems and ℎ values, they turn out superior to GP at the end of run. We have two hypotheses to explain this phenomenon. Firstly, RX can compensate the lack of mutation in our experiments, and act as its substitute. It constantly supplies population with diverse genetic material from the library, while GP, equipped only with the treeswapping crossover, is devoid of such possibility, and can, in an extreme case, cause some terminals or code fragments to disappear from the population altogether.

However, this factor is unlikely to be critical for ℎ = 3, where the library is very small (38 procedures). Thus, another explanation, not conflicting with the former, is the homologous nature of crossover. In LGX and RX, the crossover points are selected only from the loci that are present in both parents. The average depth of such locations is by definition smaller than the average depth of nodes that can be chosen by the tree-swapping crossover in GP, i.e., all nodes in the parent trees (except root, as noted earlier). For these operators then, crossover affects nodes that are on average closer to the root. Such modifications are less likely to be neutral for program semantics, which can make the exploration of the search space more intense and effective.

Whatever is the true cause for the quite good performance of RX₃ and RX₄, they almost never substantially surpass their geometric counterparts LGX₃ and LGX₄ (the only exceptions are ℛ₁ and ℛ₂, where RX takes a lead for a part of run; however, the overlap of confidence intervals may suggest that this is a mere fluke). This is the main result of this study, as it demonstrates that introducing ‘medial’ tendency in crossover makes the search process converge faster towards good solutions. It is particularly remarkable when we recall that our crossover never affects the root node. Therefore, the effects of geometric-aware changes introduced into deeper tree nodes must propagate to its root, and, on average, improve the fitness of offspring more than for the other methods.

On the other hand, one has to admit that the fitness gap between LGX and the control algorithms at the end of runs, albeit for most problems statistically significant, is usually small. This clearly demonstrates that LGX, by operating locally, on subtrees that can be located very deep in the parent trees, cannot guarantee improving the fitness of an entire program every time it is applied. Last but not least, let us notice that the confidence intervals for LGX are much narrower than those for the other methods. The behavior of this method is thus much more predictable, and, in convenient circumstances, it should be possible to estimate the expected number of generations required to attain an assumed fitness level.

The two previous observations have yet another side, clearly visible in Fig. 1: the rate of progress for LGX seems to slow down in a very predictable, asymptotic manner. To an extent, this was expected: the geometric, approximative character of LGX can be likened to the mode of operation of traditional optimization algorithms, like the bisection method or the Newton-Raphson method, where improvements usually decrease with each iteration. This may suggest that LGX is more suitable for solving optimization problems than search problems.

### 3.3 Importance of homology

In terms of operation, LGX adds two elements to standard tree-swapping crossover: homology (only procedures at corresponding loci are affected) and the semantically geometric choice of procedures (the pasted procedure is approximately medial with respect to the parents’ subtrees). The superiority of LGX to RX shown in Section 3.2 demonstrates that taking into account the geometric aspects of semantic is essential. However, a question arises: would LGX perform equally well if it was not homologous? To settle this issue, we prepared an additional control setup that uses non-homologous, locally geometric crossover operator (NHX). NHX follows the steps presented in Sec-

### Table 2: The errors (mean absolute deviation with 0.95 confidence interval) committed by the best-of-run individuals on the training set (average of 150 runs; best results in bold).

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Sextic</th>
<th>Septic</th>
<th>Nonic</th>
<th>ℛ₁</th>
<th>ℛ₂</th>
<th>ℛ₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>0.003±0.001</td>
<td>0.162±0.042</td>
<td>0.098±0.031</td>
<td>0.292±0.053</td>
<td>0.198±0.034</td>
<td>0.142±0.024</td>
</tr>
<tr>
<td>RX₃</td>
<td>0.002±0.001</td>
<td>0.130±0.039</td>
<td>0.128±0.040</td>
<td>0.175±0.039</td>
<td>0.130±0.028</td>
<td>0.164±0.027</td>
</tr>
<tr>
<td>LGX₃</td>
<td>0.003±0.001</td>
<td>0.109±0.034</td>
<td>0.114±0.041</td>
<td>0.170±0.033</td>
<td>0.086±0.020</td>
<td>0.179±0.024</td>
</tr>
<tr>
<td>NHX₃</td>
<td>0.001±0.001</td>
<td>0.138±0.047</td>
<td>0.085±0.039</td>
<td>0.292±0.073</td>
<td>0.145±0.033</td>
<td>0.141±0.030</td>
</tr>
<tr>
<td>RX₄</td>
<td>0.005±0.002</td>
<td>0.102±0.024</td>
<td>0.130±0.035</td>
<td>0.140±0.035</td>
<td>0.101±0.019</td>
<td>0.076±0.014</td>
</tr>
<tr>
<td>LGX₄</td>
<td>0.001±0.001</td>
<td>0.044±0.011</td>
<td>0.043±0.009</td>
<td>0.061±0.014</td>
<td>0.041±0.013</td>
<td>0.028±0.007</td>
</tr>
<tr>
<td>NHX₄</td>
<td>0.002±0.001</td>
<td>0.084±0.022</td>
<td>0.063±0.014</td>
<td>0.102±0.022</td>
<td>0.060±0.014</td>
<td>0.045±0.010</td>
</tr>
</tbody>
</table>
tion except for the choice of loci to be affected. In that step, it works as the standard tree-swap crossover used by GP, i.e., selects, independently for each parent, an internal tree node with probability 0.9, or a leaf with probability 0.1. The result it produces as LGX: finds the semantically most medi- 
dial procedure in the library (Eq. 2), and inserts it into both 
parents.

Table 2 compares the final (end-of-run) fitness of best-of-run individuals evolved by NHX with the other methods. For \( h = 3 \) (small library), the duel between NHX and LGX is inconclusive, methods win or loose depending on the problem. However, for \( h = 4 \) the conclusion is clear: LGX yields lower error rate and not worse variance than NHX. Homol- 
geny is then an essential component for this operator that significantly contributes to its performance.

### 3.4 Impact on tree size

As discussed above, we expect that LGX affects not only fitness, but also tree size. Figure 2 depicts the number of nodes per individual, calculated over all individuals in populations and averaged over 150 runs.

The results are very similar across all benchmark prob- 
lems. The methods using large library (\( h = 4 \)) tend to suffer from substantial bloat that is more severe than for small library (\( h = 3 \)). This can be easily explained. The aver- 
age tree depth of procedures in the library is substantially greater for \( h = 4 \) than for \( h = 3 \). On average then, every act of crossover brings more genetic material to the population in the former case. Apparently, the selection process, based exclusively on fitness, does not counteract this tendency, and hence the greater bloat.

Another observation following from Fig. 2 is that RX suf- 
fers from bloat more than LGX. This suggests that the seman- 
tically close-to-geometric procedures inserted by LGX are on average shorter than the procedures selected from the library at random by RX. The hypothesis we bring to explain this phenomenon pertains to the distribution of se- 
mantic of procedures. Due to the nature of instructions used here, and to the range of independent variable, this distribution is highly non-uniform. Many procedures have semantics that crowd closely around the origin of the semantic space (vector \( 0 \)). Typically, short procedures will have semantics of small magnitudes, as it is unlikely to produce large values using arithmetic instructions that operate on numbers form interval \([-1,1]\) (an obvious exception from this rule is the division operator). The longer procedures, on the contrary, will be capable of producing larger values on their output. The net effect of these tendencies is that semantic crossover, which has the propensity to look for procedures that are seman- 
tically medial with respect to parents’ subtrees, ends up with shorter trees more frequently.

Bloat is undesirable not only due to the detrimental effect it can have on the computational performance. In machine learning, models (hypotheses) that are overly complex of- ten turn out to perform poorly on the test set. This rises a question: how do the observed differences in tree sizes im- pact the predictive accuracy on an external test set? The next section brings an answer to this matter.

### 3.5 Test-set performance

To assess the generalization capability of considered meth- 
ods, we prepared a testing set of cases. For consistency with the training set, 20 cases are drawn from the interval \([-1,1]\), but this time randomly, with uniform distribution. The best-of-run individual for each benchmark problem and run is executed on these cases, and its generalization cap- 
pability is expressed in the same terms as for the training process (evolution), i.e., as mean absolute deviation (MAD) from the desired values.

Table 3 presents the test-set MAD of the best-of-run indi- 
viduals collected from all runs. Comparison of these figures with the performance on the test set, i.e., with the fitness values at the end of runs in Fig. 1 leads to conclusion that all methods suffer from overfitting. In terms of the ratio of test-set MAD to training-set MAD, GP is superior. How- 
ever, in absolute terms, LGX attains the lowest error on the test set for all problems. Moreover, LGX is also the most stable – the errors of programs it produces have very low variance. This result becomes particularly interesting, when we confront it with the fact that LGX, yields substantially bigger trees than GP (Fig. 2). Despite that, its predictive ability on the test data is better. It is hard to explain this phenomenon without a more thorough investigation, which is beyond the scope of this study. Apparently, there must be something to the semantic structure of the trees built by LGX that makes them more robust in this sense.

### 3.6 Timewise analysis

As admitted in Section 2.4, the benefits of LGX come at extra computational cost of creating the library and search- ing for the semantically similar procedures. While the former turns out to be low (1 . . . 2 seconds compared to 100 . . . 150 seconds of the cost of evolutionary run), the latter cannot be ignored. The roughly 250 \( \times (1024/2) \times 0.9 = 115,200 \) R-tree queries per run make LGX substantially slower. In effect, its overall runtime is on average 2.8 times longer than GP’s. This, together with the shapes of series shown in Fig. 4, urges us to ask the question: will LGX maintain its superi- ority over GP with same amount of computational resources allocated to both methods?

To verify this possibility, we conducted an additional ex- periment, which consisted in giving GP roughly the same amount of processor time as it took the corresponding LGX

<table>
<thead>
<tr>
<th>Sextic</th>
<th>Septic</th>
<th>Nonic</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>0.005 ± 0.003</td>
<td>0.187 ± 0.048</td>
<td>0.117 ± 0.035</td>
<td>0.373 ± 0.096</td>
<td>0.217 ± 0.046</td>
</tr>
<tr>
<td>RX3</td>
<td>0.004 ± 0.002</td>
<td>0.140 ± 0.040</td>
<td>0.146 ± 0.047</td>
<td>0.191 ± 0.044</td>
<td>0.129 ± 0.028</td>
</tr>
<tr>
<td>LGX4</td>
<td>0.005 ± 0.002</td>
<td>0.122 ± 0.038</td>
<td>0.117 ± 0.045</td>
<td>0.226 ± 0.069</td>
<td>0.086 ± 0.020</td>
</tr>
<tr>
<td>RX4</td>
<td>0.029 ± 0.031</td>
<td>23.443 ± 42.628</td>
<td>0.262 ± 0.102</td>
<td>8.025 ± 14.853</td>
<td>0.196 ± 0.055</td>
</tr>
<tr>
<td>LGX4</td>
<td>0.003 ± 0.002</td>
<td>0.116 ± 0.032</td>
<td>0.099 ± 0.026</td>
<td>0.102 ± 0.027</td>
<td>0.077 ± 0.032</td>
</tr>
</tbody>
</table>
By comparing the results of these runs, presented in Table 4 with the final fitness values of LGX presented in Table 2, we can conclude that GP, despite being given more time, cannot catch up with LGX, although it manages to reach the performance level of LGX3 for some problems. Therefore, LGX can be considered attractive not only from theoretical viewpoint, but also in practical perspective.

4. CONCLUSION

The main conclusion we draw from this study is that search operators that are homologous and at the same time semantically meaningful can improve the efficiency of GP search and cause the evolved programs to generalize better. Thus, it may pay off to force evolution to work in a special mode, in which, with time of evolution, a common semantic blueprint begins to emerge in the population. The subprograms located in particular loci start to specialize in solving certain subproblems of the original problem. This hypothesis sounds very attractive, as it suggests capability to discover semantic modules in the structure of the problem, which in turn could provide possibility of problem decomposition.

In other words, LGX seems to facilitate the discovery of subproblems within the problem being solved. By making the exchange of genetic material at the same time homologous and driven by program semantic, we enable evolution to bind a subproblem to a specific locus in the genotype, and work semi-independently on the parts of programs delegated to solving particular subproblems.

LGX and the related concepts have been presented and verified here in the context of symbolic regression. It is however worth noticing that it has wider applicability. Any domain for which semantic is computable and semantic metric is available, can be subject to this approach. If the metric is also a norm, as it was here, Equation 1 can be used to locate the midpoint between parents' semantics. But it does not have to: if it is not a norm, there are other alternative ways in which a crossover can be made semantically medial.

5. REFERENCES


Figure 1: Best-of-generation fitness graphs averaged over 150 evolutionary runs.
Figure 2: Number of nodes (population mean) averaged over 150 evolutionary runs.