Genet Program Evolvable Mach (2013) 14:31–63
DOI 10.1007/s10710-012-9172-7

Locally geometric semantic crossover: a study on the roles of semantics and homology in recombination operators

Krzysztof Krawiec · Tomasz Pawlak

Received: 26 June 2012 / Revised: 23 August 2012 / Published online: 30 October 2012

Abstract This study presents an extensive account of Locally Geometric Semantic Crossover (LGX), a semantically-aware recombination operator for genetic programming (GP). LGX is designed to exploit the semantic properties of programs and subprograms, in particular the geometry of semantic space that results from distance-based fitness functions used predominantly in GP. When applied to a pair of parents, LGX picks in them at random a structurally common (homologous) locus, calculates the semantics of subprograms located at that locus, finds a procedure that is semantically medial with respect to these subprograms, and replaces them with that procedure. The library of procedures is prepared prior to the evolutionary run and indexed by a multidimensional structure (*kd*-tree) allowing for efficient search. The paper presents the rationale for LGX design and an extensive computational experiment concerning performance, computational cost, impact on program size, and capability of generalization. LGX is compared with six other operators, including conventional tree-swapping crossover, semantic-aware operators proposed in previous studies, and control methods designed to verify the importance of homology and geometry of the semantic space. The overall conclusion is that LGX, thanks to combination of the semantically medial operation with homology, improves the efficiency of evolutionary search, lowers the variance of performance, and tends to be more resistant to overfitting.

Keywords Geometric crossover · Semantics · Library · Spatial index · *Kd*-tree · Homology

K. Krawiec · T. Pawlak (✉)
Institute of Computing Science, Poznan University of Technology, Poznan, Poland
e-mail: tpawlak@cs.put.poznan.pl

K. Krawiec
e-mail: krawiec@cs.put.poznan.pl

1 Introduction

Computer programs are symbolic structures that are designed to attain a desired operational effect, i.e., produce certain output given input data. That output is a combined effect of the instructions a program is composed of, or, more precisely, of their *semantics*. Semantics is the knowledge that provides 'grounding' for the symbols used at the syntax level (instruction opcodes). Formally, it can be expressed in a lower-level notation (operational semantics) or by defining the constraints that bind instruction's output to input (denotational semantics).

Though it is semantics that determines what a program actually does, most of genetic programming (GP) algorithms ignore that knowledge and manipulate programs only with syntax in mind. Focusing on syntax alone allows them to rely on simple, generic search operators. But this comes at a cost: very little, if anything, can be said about the anticipated impact of program modification on the output it produces as that is inherently semantic property. Even a minimal change on the syntactic level (genotype) can have profound and hard to predict impact on the overall semantic (phenotype) of program, and thus on its outcome and fitness. In other words, the genotype-phenotype mapping in GP is usually highly contextual (epistatic), with effects of particular instructions strongly influencing each other. This causes GP fitness landscapes to be complex [16], which in turn limits scalability of search algorithms.

Rugged and epistatic fitness landscapes are obviously not unique to GP, and occur also in more traditional learning and optimization tasks. In such problems however, solutions (individuals) are represented as fixed-length vectors of variables, and epistasis boils down to interdependencies between them. Also, the role a variable plays in a solution is determined by its position in the vector, and is common for all solutions. This eases designing effective recombination operators. Even the simple one-point crossover is tailored to exploit the 'linkage' between variables by swapping *groups* of them between parent solutions. When such generic operators do not perform well enough, problem-specific ones can be designed based on domain knowledge that often indicates which variables are likely to be interdependent. Also, methods exist that autonomously detect and exploit interdependencies, like competent GAs [9] or estimation of distribution algorithms [36].

In broader terms, recombination operators are intended to exploit *modularity* of problems [43], where modules are some entities comprising solution *components*. Variables are specific form of such components. In GP however, there is no clear counterpart of the concept of variable, and it is typically assumed that program fragments (procedures, subprograms) play the role of modules and their components. Following this assumption, typical recombination operators swap code pieces between parent programs. In doing so however, they do not care about the semantics of transferred fragments, their presumed roles within the programs they are part of, whether such roles correspond to each other in any meaningful way, and the possible impact of such a change on program output.

The main motivation for this work is our conviction that such an approach is overly simplistic, and that search operators can be designed that exploit the semantic properties of programs and program fragments. We provide an extensive account of

locally geometric semantic crossover (LGX), a recombination operator we introduced in [20]. LGX has controlled impact on program semantics and aims at making the offspring programs semantically more similar to each other than the parent programs. To attains this, LGX inserts into parents a procedure (program) that is semantically medial with respect to the code fragments being replaced. This modification takes place at the same locus (position in program code) in both parents, and is intended to impose and exploit semantic correspondence of homologous program fragments. This design, motivated in Sect. 3, causes LGX to perform better than the standard GP tree-swapping crossover and other control operators, including semantic-aware operators proposed in other studies (Sect. 5).

The paper is organized as follows. Section 2 defines the notion of program semantics we adopt for this study. Section 3 briefly describes the motivations for LGX's design, the algorithm itself, and the library of procedures it relies on. Section 4 reviews the related work and the state of the art on semantics in GP. Section 5 reports the outcomes of an extensive experimental analysis. Sections 6 and 7 discuss the results and summarize the conclusions.

2 Program semantics for GP

Introduction of a semantic-aware search operator requires adopting a formal concept of program semantics in the first place. In theories of formal languages, this concept is typically defined in operational or denotational way, each of which describes the effects of program execution for all possible input data. However, as a machine learning technique [29], GP confronts programs with a finite training set of *fitness cases*, each being a pair consisting of an input and the corresponding desired program output. Assuming that this set is the only available data that specifies the desired outcome of the sought program, a natural definition of semantics is the vector of outputs a program produces for the input parts of these cases [28]. Semantics may be then viewed as a point in n-dimensional *semantic space*, where n is the number of fitness cases. Because fitness cases usually do not enumerate all possible program inputs, this concept is also known as *sampling semantics* [44].

This framing of semantics has a few advantages. A technical advantage is that determining semantics of a program within an evolutionary process comes essentially for free, because each program has to be run on all fitness cases to calculate its fitness. Calculating a program's semantics is then a side-effect of fitness calculation, available at no extra computational cost. Moreover, this applies also to the outcomes produced by a program at any point of its execution (i.e., any subprogram of an evaluated program), which has certain implications for the approach proposed in this paper.

More importantly however, such understanding of semantics binds it closely to the fitness function, which, for most application domains, captures the divergence between program output and the desired output. Typically, fitness is defined as a distance (or a function thereof) between program semantics and the *target*, a point in

semantic space determined by the n desired program outputs. This makes the semantic space metric, which has fundamental consequences that we will exploit in Sect. 3.1 (cf. also [32]).

This definition of semantics, together with the concept of target, is very natural for function approximation (symbolic regression) and Boolean problems. However, it can be employed in other scenarios and in many, if not all, application domains of GP. For instance, a controller that balances an inverted pendulum can be evaluated on n fitness cases, each defining a different set of initial conditions of a simulation process. The time for which the controller prevents the pole from tipping over defines its performance on a given fitness case. Assumed the maximal simulation time is t, the target is the point t^n in n-dimensional semantic space, and an individual's fitness can be defined as, e.g., the city-block distance of that individual's semantics from this target point.

It is worth noting that semantics as framed above obviously does not capture program outcome for *all* possible inputs. Two programs that appear semantically equivalent, i.e., return the same output for all fitness cases, can behave differently for an input datum that was not present in the training set; GP shares this characteristic with all algorithms that inductively learn from examples. An important question, then, is how well the evolved programs generalize beyond the training data (see Sect. 5).

3 Locally geometric semantic crossover

In this section we motivate and describe the locally geometric semantic crossover (LGX) [20]. In Sect. 3.1 we present step-by-step the reasoning that underpins LGX's design. In Sect. 3.2, we provide concise, self-contained algorithmic description of LGX, and in Sect. 3.3 we describe preparation steps needed before a GP run.

3.1 Design rationale

3.1.1 Semantically geometric offspring

As argued in Sect. 2, fitness of an individual in GP is usually based on the error it commits when applied to the fitness cases. In accordance with this, we assume that fitness $f(p)$ of a program p is based on a given*metric* that captures the divergence between the output of p and the known desired output[1]:

$$f(p) = ||s(p) - t||, \tag{1}$$

where $s(p)$, the *semantics* of p, is the vector of outputs produced by p for the fitness cases, and t (*target*) is the vector of desired outputs. The metric $||\cdot||$ operates thus in an n-dimensional *semantic space*, where n is the number of fitness cases.

[1] Because we compare vectors, this metric is also a *norm*.

Importantly, it can be also used to measure the semantic similarity of *pairs* of programs, which is the key element of the proposed approach. Note also that this definition of semantics does not depend on what is the nature of data processed by the programs, nor, e.g., on what is the number of input/independent variables when the task is symbolic regression. The length of semantics depends directly only on the *number* of fitness cases.

Under this definition, the objective is to minimize f, and the fitness landscape can be visualized as a unimodal surface hovering over the space of solutions, with the lowest point touching t (as only for this argument the fitness reaches zero), and raising monotonously with the growing distance from t. For instance, if the metric used in Eq. (1) is the Euclidean distance, this surface will assume the shape of an upside-down cone with its apex located at t. Most importantly however, a fitness landscape defined by Eq. (1) is unimodal for *any* metric, and convex for L^p norms (e.g., Euclidean distance, city-block distance) [32]. This opens the possibility for designing operators that exploit these geometric properties to yield offspring of good quality (e.g., topological crossover [33]). To do so, a crossover operator applied to a pair of parent programs p_1 and p_2 should produce programs whose semantics are 'in between' the semantics of p_1 and p_2. We formalize this in the concept of *semantically geometric offspring* (*geometric offspring* for short, [32]), which is an offspring program o that fulfills

$$||s(o) - s(p_1)|| + ||s(o) - s(p_2)|| = ||s(p_1) - s(p_2)||. \quad (2)$$

Geometric offspring have several attractive properties. By minimizing the total distance from the parents, the semantics of a geometric offspring is as intermediate with respect to the parents as possible. In other words, it is a 'perfect mixture' of its parents, which is what crossover operator should provide. For certain norms, geometric offspring offers even more specific and appealing features. For instance, for the Euclidean metric, it is guaranteed to be at least as fit as the less fit of the parents, because no point on the segment $\overline{s(p_1)s(p_2)}$ can be further from t than the more distant of $s(p_1)$ and $s(p_2)$ (see [31] for proof). If the population is diversified enough and the parents occupy the opposite 'slopes' of the conic fitness landscape, geometric offspring is likely to be better than *both* parents. Similarly attractive properties can be proven for other norms. For the city-block metric (Manhattan metric, L_1), the expected fitness of a geometric offspring amounts to $(f(p_1) + f(p_2))/2$, assuming that the operator picks the offspring uniformly from the segment connecting the parents in semantic space (see [18] for proof).

As a side note, let us comment that the above definition does not control how equidistant is $s(o)$ from $s(p_1)$ and $s(p_2)$. For instance, an offspring that is semantically equivalent to one of the parents, e.g., $s(o) = s(p_1)$, is still geometric in the sense of the above formula, although it does not have more in common with p_2 than p_1. However, to keep our approach simple, in this study we ignore this aspect (which we discussed in more detail in [18]).

Unfortunately, designing a crossover operator capable of producing semantically geometric offspring is hard in general, as such offspring is defined by its semantic properties, while evolutionary search manipulates program syntax. Finding o such that $s(o)$ fulfills Eq. (2) is another program induction task in itself (and such o may

not exist in the first place).[2] To circumvent this problem, we first note that, given the stochastic nature of GP search, it may not pay off to put a lot of computational effort into searching for a *perfectly* geometric offspring. An offspring that is only approximately geometric can be still quite effective at advancing the search process towards the global optimum. This may be seen as restating the problem and searching for an offspring that minimizes divergence from 'geometricity' (cf. [18]), which can formalized by relaxing Eq. (2):

$$o = \arg\min_{p}(||s(p) - s(p_1)|| + ||s(p) - s(p_2)|| - ||s(p_1) - s(p_2)||) \tag{3}$$

However, even if posed as an optimization problem, Formula (3) still remains a program synthesis task, and as such is challenging. A brute-force way of solving it could consists in generating offspring candidates at random. However, such trial-and-error approach, which we investigated in [19], is inevitably computationally expensive. Also, the odds of randomly generating an almost-geometric offspring program diminish with the growing complexity of the parent programs.

3.1.2 Problem decomposition

Given that the prospects for effectively finding an approximately geometric offspring via search are limited, we propose an alternative way of exploiting the geometric properties of the semantic space. To this aim, we rely on the inherently compositional nature of programming tasks and use it to shift the problem of finding geometric offspring from the level of complete programs to the level of subprograms.

Let program p_t be a solution to a given programming task t, i.e., $s(p_t) = t$. Assuming p_t comprises at least two instructions, it can be decomposed into *subprograms* p' and p'' such that $p' \circ p'' \equiv p_t$, where $\circ$ is program composition operator (e.g., concatenation of trees in the case of LGX). This implies that $t' \equiv s(p')$ defines a target (desired semantics) of another problem, which is a *subproblem* of the original problem t.

Now, let p be a candidate solution to such a subproblem. It can be expected that the more similar is the semantic $s(p)$ to t', the better the fitness of the compound program $p \circ p''$ is. In broader terms, the better a part of solution becomes at realizing certain desired functionality, the fitter we expect the entire solution to be. To illustrate this in a more general setting, consider the task of designing an electric torch, composed of a bulb and a battery, aimed at maximizing the light output [21]. There certainly exists (at least one) characteristic of an ideal battery, i.e., a battery p_{bat} for which there is a 'compatible' bulb p_{bul} such that the torch $p_{bat} \circ p_{bul}$ yields maximum light. It may be also expected that, under fixed bulb p_{bul}, the more a battery p is *functionally* (semantically) similar to p_{bat}, the stronger the emitted light.

This property, termed *monotonicity* in our previous studies [21, 22], cannot obviously be expected to hold for all problems and all decompositions of a problem into subproblems. The interactions between components (here: subprograms) are often complex, so that an improvement of a component does not always monotonously

[2] Albeit in [32] we have shown that for certain problem domains a semantically geometric offspring can be constructed via purely syntactic manipulation.

ameliorate the overall fitness. Problems with deceptive decompositions can be concocted for which, by analogy to trap functions (see, e.g., [36]), making the subprogram p' closer to target t' actually *deteriorates* the fitness of the compound program $p' \circ p''$. Consider for instance $p' = x$ and $p'' = \sin()$: changing monotonously the output of p' will cause the output of p'' to rise and fall, and thus move alternately closer and further from a target. Nevertheless, it is reasonable to assume that some decompositions exhibit this property, at least in an approximate sense.

3.1.3 Semantically intermediate subprograms

Without additional knowledge of the task being solved, it is hard to automatically decompose it in a manner that guarantees that the validity of the above reasoning. More precisely, there is no way of effectively choosing, from the possibly many subproblems, the subproblem t' that provides monotonicity. Therefore, LGX avoids explicit decomposition and, by allowing arbitrary crossover loci, operates on *any* subprograms of the parent programs. To this aim, it chooses a homologous locus in the parent programs (see Fig. 1) and replaces the code at that locus with a *procedure*, a piece of code that provides low divergence from geometricity in a *local* sense, i.e., considering the semantics of the affected subprograms. The procedure is selected from a previously prepared *library* by minimization of

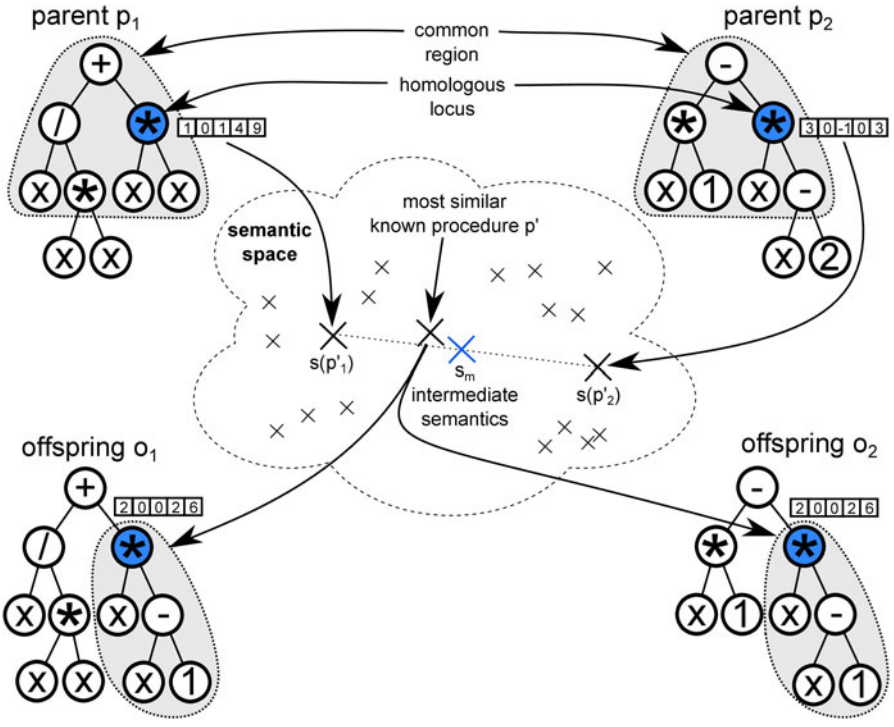

Fig. 1 The working principle of LGX. Vectors of numbers illustrate semantics of subprograms

Formula (3), with p_1 and p_2 being this time subprograms of the parent programs, located at the selected homologous locus (see Sect. 3.2 for details).

Why would pasting such a procedure into the parent programs help the offspring? Due to the geometric properties of semantic space discussed at the beginning of this section, a procedure that minimizes the divergence from parents' subprograms is likely to be semantically close to an (unknown) subgoal t'. That in turn, based on the hypothesized monotonicity discussed above, should improve the fitness of the resulting offspring programs. In other words, the 'centripetal' effects of geometric crossover, when applied locally to subprograms, may result in an analogous effect in entire offspring programs.

The motivation for making LGX homologous is as follows. Applying the above operation to subprograms p'_1, p'_2 in the parent programs p_1 and p_2 is more likely to improve their fitness if they share the remaining program part (often called *context*, following [28]), so that $p_1 = p'_1 \circ p''$ and $p_2 = p'_2 \circ p''$. However, the parents are paired at random and are in general arbitrarily different, so they typically do not share p''. Thus, to provide at least some commonality of the way the parents are affected, we make LGX homologous. As we show in the experiment, this aspect is indeed significant.

As follows from the above arguments, geometric crossover performed on the level of subprograms *can* improve the fitness of offspring programs. However, we previously argued that, when applied to entire programs, it is *guaranteed* to produce offspring with attractive properties. Why do we insist then to choose the former? The key point is that, while it is often difficult to find a close-to-geometric program for complete, possibly large and complex parent programs, achieving an analogous goal on the level of subprograms may be easier, as they are by definition simpler than programs. This can be done at a reasonable computational cost, which we cover in the next sections.

3.2 The algorithm

In this section, we provide a technical description of LGX, abstracting from the motivations presented in the previous section. The presentation assumes tree-based GP, but LGX is easily portable to other program representations.

Given two parent programs p_1 and p_2, LGX first determines the *common region* for them (see Fig. 1). The idea of the common region, introduced by Poli and Langdon [38], is a set C of loci that occur in both p_1 and p_2. Formally, C can be defined via recursion: (1) The locus of the root node belongs to C. (2) A locus belongs to C if its parent locus belongs to C and the corresponding nodes in p_1 and p_2 have parent nodes of the same arity. Note that this concept is based solely on program structure (tree shape); the labels of nodes are ignored.

Given the common region C, LGX draws from it a random locus that will be used as the crossover point. As C is a *tree* of loci, to choose this point we employ the same rules as the canonical crossover operator [17]. An internal node in C is selected with probability 0.9, and a leaf with probability of only 0.1, to reduce bloat. The root is selected only if the common region contains only one node. It is worth

emphasizing already at this point, that choosing the locus from the common region causes LGX to affect on average shallower tree fragments than many other operators, including the standard tree-swapping crossover.

Subsequently, LGX identifies the subprograms (subtrees) p'_1 and p'_2 rooted in the drawn locus in p_1 and p_2, respectively. Their semantics, $s(p'_1)$ and $s(p'_2)$ are already calculated as we cache the semantics for each tree node when executing programs during the evaluation of individuals. Given $s(p'_1)$ and $s(p'_2)$, we determine the midpoint between them in the semantic space:

$$s_m = \frac{s(p'_1) + s(p'_2)}{2} \tag{4}$$

s_m represents the semantics of a hypothetical program $p{:}s_m = s(p)$ that would be perfectly geometric with respect to subprograms p'_1 and p'_2 (see Eq. 2). To find a program that approximates s_m, we search a *library* L of previously prepared short *procedures* (see Sect. 3.3) for a procedure that minimizes the semantic distance from s_m:

$$p' = \arg\min_{p \in L} ||s(p) - s_m|| \tag{5}$$

It is worth noting that such choice of procedures is more specific than the one presented in the previous section. A procedure that minimizes the above expression not only has low divergence from geometricity (Eq. (5)), but is also likely to be equidistant to parents, a combination of features we term *medial* in the following (cf. [18]). In this way, we lower the risk of choosing a procedure that is semantically very similar (or identical) to p'_1 or p'_2 (for the setups used in the experimental part of this paper, semantically identical subtrees are selected in about 8 % acts of crossovers).

In the simplest variant of the approach, which we proposed in [20], the procedure p' chosen in this way replaces the subtrees p'_1, p'_2 at the selected locus in both parents, producing two offspring and completing the crossover act. However, choosing p' in this deterministic manner can lead to premature convergence of the search process. This is particularly likely when no other search operator is used along with LGX to provide exploration. Thus, rather than choosing the procedure strictly according to Eq. (5), a set of k closest neighbors of s_m are determined, and one of them is selected at random and planted into the parent programs. The neighborhood size k is the parameter of the method.

Note also that usually no procedure from the library brings $||s(p) - s_m||$ to zero. The library is intended to store relatively simple procedures, which cannot be expected to express any semantics (see the next section). For this reason, it may be argued that the differences between the k closest neighbors are quantitative rather than qualitative.

3.3 The library of procedures

The library, designed to provide semantically diverse subprograms for LGX, is prepared prior to the evolutionary run using the instruction set I provided with the

task. The choice of procedures to be gathered in L could be made in many different ways. Here, for simplicity, L is filled with all trees of height at most h built from instructions from I, where h is a parameter of the method.

For every procedure $p \in L$, we calculate its semantics $s(p)$, i.e., the vector of outcomes produced by p for the considered set of fitness cases. Any two procedures p_1, p_2 having 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 the procedures that duplicate the semantics of other procedures. For each subset of procedures having the same semantics, only the shortest one remains in L.

To provide fast retrieval of semantically similar procedures (Formula (5)), we employ a *multidimensional index*, a data structure designed for geographic and spatial databases. We chose *kd-trees* [5] due to very good performance of nearest neighbor search in multidimensional data sets. The typical time of insertion of m points into the *kd*-tree is $O(m \log {}^2 m)$, however it can be reduced to $O(m \log m)$ with the linear-time median searching algorithm (eg. [6]). The nearest neighbor query to the tree consisting m elements takes $O(m^{1-1/d})$ time in worst case, where d is the dimensionality of indexed space, but only $O(\log m)$ on average [7, 25].

The above three steps of library preparation (generating the procedures, calculating their semantics, and building the index) introduce certain computational overhead. However, a library, once prepared for a given instruction set and a set of program inputs comprised by fitness cases, can be reused in multiple runs and problems, in which case this overhead can effectively become negligible. Also, for typical symbolic regression tasks this process turns out to take mere seconds on contemporary hardware, even for several dozens of thousands of procedures (see Sect 5).

Alternatively, the library could be filled up during the run, by collecting the subprograms created by evolution. Although this could lower the cost of library preparation, it would also make its contents dependent on the course of evolution, and make the analysis of performance less objective. For this reason, we decided to prepare the entire library before the run.

4 Related work

There are several contributions in the past GP literature that share some common elements with the proposed LGX operator. They can be grouped along two main features of LGX: the use of a library, and the semantically-aware manipulation of programs.

With respect to the former, LGX can be likened to, e.g., run transferable libraries (RTL, [41]), which are repositories of program fragments collected in separate GP runs, intended to be applied to different problem instances. However, contrary to LGX, RTL requires a series of preliminary GP runs, because fragment's inclusion in the library depends on its long-term frequency of occurrence in an evolving population. Also, the collected modules are then re-used at random, i.e., RTL does not involve any directed (e.g., semantic-driven) operator to pick the procedures. In [15], we demonstrated the usefulness of a similar approach on a task of visual

learning. Rosca and Ballard [40] create an analogous library on-the-fly, during a single evolutionary run. To update the library at every generation, they use a sophisticated mechanism for assessing subroutine utility, and employ entropy to make decision when a subroutine should be created. However, as in RTL, the subroutines are used in an undirected way, without considering their semantic properties. Haynes [10] integrated a distributed search of genetic programming-based systems with collective memory, however he utilizes a library only for detection of redundancy of GP search agents that operate in parallel. Galvan Lopez et al. [8] reused code using a special encapsulation terminal, causing code fragments to be stored implicitly as fragments of individuals in population (in contrast to LGX, where the library is fixed and explicit). Other approaches involving some form of library include reuse of assemblies of parts within the same individual [11], identifying and re-using code fragments based on the frequency of occurrences in the population [12], and explicit expert-driven task decomposition using layered learning [1, 13].

Considering the semantic aspects, we review here the approaches that rely on the understanding of program semantics introduced in Sect. 2, even if some of them do not explicitly refer to the idea of semantics. McPhee et al. [28] were probably the first to study the impact of crossover on program semantics and so-called semantic building blocks. They defined, for Boolean problems, the semantic properties of components that form offspring in tree-swapping crossover, i.e., subtrees and contexts (partial trees with a hanging branch), and observed how they change with evolution time. Beadle and Johnson [3] used program semantics to increase diversity of programs, proposing a method that produces initial populations of semantically unique programs, and confronted it with standard population initialization techniques (which typically produce many semantically identical solutions). This idea was further explored by Jackson [14]. Subsequently, Beadle and Johnson applied a similar idea the mutation operator [2] and crossover [4], forbidding creation of offspring that was semantically identical to parents. The proposed methods proved superior to standard GP in terms of the number of fully successful programs evolved.

The approaches cited above used semantics primarily to *detect* differences in program outcomes. Measuring semantic *similarity* was a natural further step in exploitation of semantic information. Semantically-aware crossover (SAC, [35]) and semantic similarity-based crossover (SSC, [45]) proposed by Nguyen et al. employ semantic similarity to make decision whether a pair of subtrees in parent solutions should be swapped. In the cited work, SAC and SSC have been reported to outperform the standard tree-swapping crossover. These operators serve as reference methods in the experimental part of this paper and will be detailed there.

Yet another step in development of semantically-aware methods involves explicit exploitation of selected properties of semantic space. Such properties have been studied by Moraglio [33, 30] and resulted in the idea of geometric offspring as presented in Sect. 3.1. In [19], Krawiec and Lichocki proposed using a trial-and-error approach (essentially a form of brood selection) to design a crossover operator that is approximately geometric. In [32], Moraglio et al. considered properties of semantic spaces for different metrics and proposed a syntax-based technique for designing exact semantically geometric crossovers.

In the context of these contributions, LGX remains unique in combining the three elements: selection of procedures w.r.t. their semantic properties, homologous character of crossover, and the use of a library of procedures.

5 Experimental analysis

The experiments presented here have two main goals: (i) assessment of LGX's performance and (ii) verification of the rationale discussed in Sect. 3.1. To attain (ii), we implement three control crossover operators, chosen to assess the impact of semantic mediality and homology on LGX operation.

To verify the importance of semantic mediality, we use two operators that are homologous but ignore the semantics of programs. The first of these is the homologous tree-swapping crossover (**GPH**), called one-point crossover by Poli and Langdon in [37]. It computes the common region of parents in the same manner as LGX (Sect. 3.2), selects a single locus in it, and swaps the parents' subtrees rooted at the selected locus. Thus, GPH chooses the locus like LGX but affects the code at that locus like standard tree-swapping GP crossover, and thus ignores semantics of subprograms.

The second operator, *random crossover* (**RX**), is also homologous and selects the affected locus like LGX. It also uses the same library L for replacing the code at the locus. However, RX's choice of procedures from L is purely random. As in LGX, the chosen procedure is pasted into *both* parents. Thus, RX uses the same repertoire of code fragments as LGX, but, similarly to GPH, its choice of procedure is not influenced by program semantics. Moreover, that choice does not depend at all on the replaced subtrees, so RX can be considered as a form of mutation.

To verify the importance of homology, we introduce **NHX**, an operator that is non-homologous but locally medial. When applied to a pair of parents, NHX proceeds as LGX except for the choice of the locus to be affected. To make this choice, NHX works as the standard tree-swapping crossover, i.e., selects the subtrees p'_1 and p'_2 to be swapped independently in both parents. For this, it uses the same probability distribution as LGX (Sect. 3.2). Then it proceeds as LGX, i.e., finds the semantically most medial procedure in the library (Formulas 4 and 5) and inserts it into both parents. Thus, NHX works like LGX without the concept of homology.

To provide a more complete picture for the performance of particular operators, we use also the following control methods:

GPX The canonical tree swapping crossover [17].

SAC *Semantic aware crossover* [35]. SAC picks at random two subtrees in the parents and checks whether their semantic distance is greater than a predefined threshold (called semantic sensitivity), in which case it swaps them. Otherwise, it picks another subtrees and tries again, however for a limited number of times.

SSC *Semantic similarity-based crossover* [45], an advanced version of SAC. Rather than relying on a single parameter, SAC swaps subtrees if their

semantic distance is within a range limited by two parameters, lower and upper bound on semantic sensitivity.

For convenience, we summarize the properties of considered operators in Table 1.

5.1 The setup

W apply the compared methods to 12 univariate symbolic regression problems taken from [27] and [23], shown in Table 2: three polynomials, three rational functions, and six irrational functions. For each configuration, 100 runs are carried out, each starting from a different initial population of size 1,024. Fitness is minimized and defined as the absolute error of individual's output w.r.t. the desired output, summed for the 20 fitness cases drawn uniformly (equidistantly) from a given interval (Table 2). In other words, fitness is defined as in Formula (1), with the metric $||\cdot||$ being the city-block distance between individual's output and the target t, calculated by applying a formula from Table 2 to the training set. A run is considered as success if it finds a program that has fitness smaller than 10^{-6}.

As we compare the methods both in terms of elapsed generations (Sect. 5.2) and time-complexity (Sect. 5.3), we employ two stopping criteria. A run is terminated if it lasts for at least 250 generations *and* at least 200 s. If an ideal solution is found earlier, a run is assumed to have the ideal fitness (zero) for the remaining generations.

The detailed setup is given in Table 3. Note that most methods do not involve mutation. The exceptions are GPX and GPH, where the tree-swapping crossover alone cannot supply the population with new genetic material. For the population sizes used here this can easily lead to stagnation, so we provide these setups with the standard mutation operator that replaces a randomly selected subtree with a subtree generated using the ramped-half-and-half method [26]. The probability of crossover is the same for all setups to ensure fair comparison. Other parameters are set to the defaults of the ECJ package [26] that was used as the experimental framework.

For the methods involving a library of procedures L, we consider two settings of the maximum procedure height. For $h = 3$, the assumed instruction set results in 289 procedures, but only 209 of them are semantically distinct, so $|L| = 209$. For $h = 4$, these figures amount to 269217 and 104469, respectively. We append the value of h to method's name (e.g., LGX_4 for $h = 4$) to indicate which version we

Table 1 The properties of the compared crossover operators

		Semantic mediality	
		Absent	Present
Homology	Absent	GPX, SAC, SSC	NHX
	Present	GPH, RX	LGX

Table 2 The benchmarks used for experimental comparison

Problem	Definition (formula)	Training set	Test set
Sextic	$x^6 - 2x^4 + x^2$	$U[-1, 1, 20]$	$R[-1, 1, 20]$
Septic	$x^7 - 2x^6 + x^5 - x^4 + x^3 - 2x^2 + x$	$U[-1, 1, 20]$	$R[-1, 1, 20]$
Nonic	$x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x$	$U[-1, 1, 20]$	$R[-1, 1, 20]$
R1	$(x + 1)^3/(x^2 - x + 1)$	$U[-1, 1, 20]$	$R[-1, 1, 20]$
R2	$(x^5 - 3x^3 + 1)/(x^2 + 1)$	$U[-1, 1, 20]$	$R[-1, 1, 20]$
R3	$(x^6 + x^5)/(x^4 + x^3 + x^2 + x + 1)$	$U[-1, 1, 20]$	$R[-1, 1, 20]$
Nguyen-5	$\sin(x^2)\cos(x) - 1$	$U[-1, 1, 20]$	$R[-1, 1, 20]$
Nguyen-6	$\sin(x) + \sin(x + x^2)$	$U[-1, 1, 20]$	$R[-1, 1, 20]$
Nguyen-7	$\log(x + 1) + (x^2 + 1)$	$U[0, 2, 20]$	$R[0, 2, 20]$
Keijzer-1	$0.3x\sin(2\pi x)$	$U[-1, 1, 20]$	$R[-1, 1, 20]$
Keijzer-4	$x^3 e^{-x}\cos(x)\sin(x)(\sin^2(x)\cos(x) - 1)$	$U[0, 10, 20]$	$R[0, 10, 20]$
Keijzer-9	$\log(x + \sqrt{x^2 + 1})$	$U[0, 100, 20]$	$R[0, 100, 20]$

$U[a, b, n]$ means n fitness cases selected *uniformly* from the given interval $[a, b]$. $R[a, b, n]$ means n *randomly* chosen points from the interval $[a, b]$

Table 3 Experimental setup

Parameter	LGX, NHX, RX	SAC, SSC	GPX, GPH
Instruction set	$\{+, -, \times, /, \sin, \cos, \exp, \log, x\}$[a]		
Population size	1024		
Initial max tree depth	6		
Max tree depth	17		
Selection	Tournament selection		
Tournament size	7		
Trials per experiment	100 independent runs		
Termination condition	250 generations and at least 200 s of total time		
Crossover probability	0.9		
Mutation probability	0.0	0.0	0.1
Reproduction probability	0.1	0.1	0.0
Max tree depth in library	{3,4}	–	–
Neighborhood size	8	–	–
Semantic sensitivity	–	0.0500[b]	–
Lower bound semantic sensitivity	–	0.0001[c]	–
Upper bound semantic sensitivity	–	0.4000	–

[a] The division and logarithm are protected. It is assumed that division by 0 returns 0 and logarithm calculates absolute value of its argument

[b] The value comes from [35]

[c] The values for lower and upper bounds of semantic sensitivity come from [44]

are discussing. Preliminary experiments have shown that setting LGX's neighborhood size to $k = 8$ works well, so this value has been adopted in this experiment.

Overall, there are 10 methods, 12 benchmarks and 100 runs for each configuration, which implies 12,000 evolutionary runs, each lasting at least 200 s. The following sections describe in detail selected aspects of obtained results.

5.2 Performance over generations

The graphs of best-of-generation fitness with 0.95 confidence intervals for all analyzed methods are shown in Figs. 2 and 3, with the methods involving semantic mediality shown in red, and the remaining ones in green. Note that the range of ordinate varies from problem to problem.

5.2.1 Evolution dynamics and performance on categories of problems

LGX is unquestionable winner in terms of best-of-run fitness on the polynomial problems. Even for the easiest *Sextic* problem there are noticeable differences between the methods. Almost all runs of LGX_3 solve it in 75 generations or less. In the first part of runs, the runner up is NHX_3, but around generation 120 it is overtaken by LGX_4 (though the final difference is not statistically significant). It is worth noting that good results are achieved by SAC as well. SSC converges faster, but is worse and comparable with GPH at the end of run.

For the more difficult *Septic* and *Nonic*, the situation is quite different. The best results are achieved now by LGX_4, but finally LGX_3 catches up at the end. In both cases RX_4 is almost as good as LGX_3, which can be due to having access to a larger library. Note that in these two cases NHX_4 is worse than its random counterpart RX_4. For NHX and RX employed with small library ($h = 3$), the results are inverted. SAC is a bit worse than NHX_4 in both cases, while SSC is substantially inferior.

The rational problems (R1, R2, R3) seem to be easy for LGX_4, which noticeably beats the competition in terms of speed of convergence and best-of-run fitness. For R1 and R2, for the initial part of runs the second place is taken by NHX_4, but at the end it is caught up by LGX_3, causing this pair to be statistically indistinguishable by the end. For R3, the second place is taken by GPX for most part of run, however finally it is overtaken by RX_3. Despite that, the difference of their results is not statistically significant. The R2 problem is somehow unusual, with all semantically medial methods achieving similar level of performance and the non-medial methods doing much worse (including the semantically-aware, albeit non-medial SAC and SSC).

The *Nguyen* problems seem to be easy for LGX/NHX methods, especially for $h = 3$. Both methods typically reach optimum in less than 50–100 generations. Surprisingly, for the harder of these problems, NHX_3 overtakes LGX_3 but not for long time. This can be explained by NHX's ability to affect deeper tree parts than LGX, which can lead to bigger and fitter programs. Both methods employed with the bigger library ($h = 4$) yield slightly worse results. All *Nguyen* problems also seem to be easy for SAC and SSC. GPH loses in all these problems.

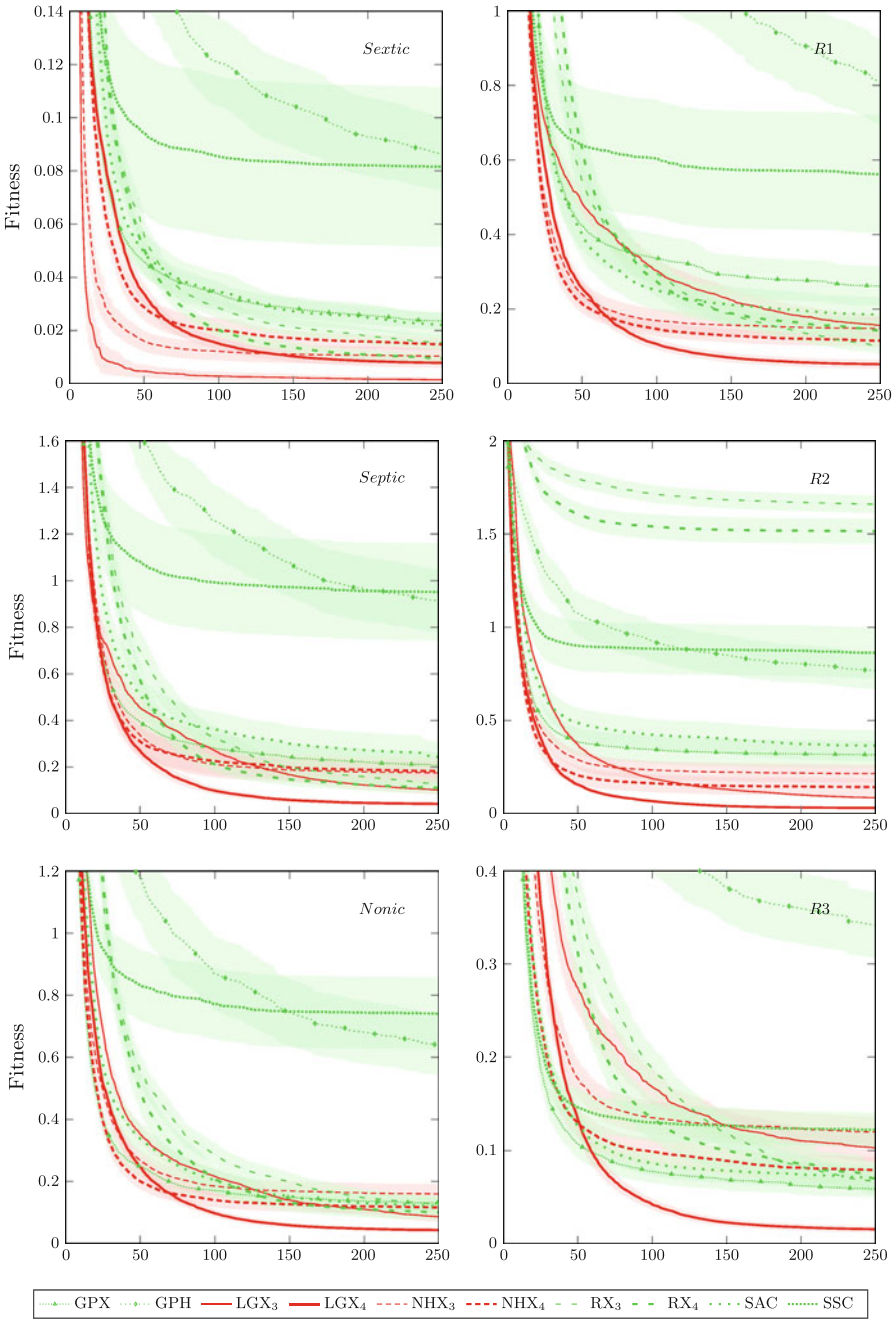

Fig. 2 Best-of-generation fitness averaged over 100 runs with 0.95 confidence intervals (I)

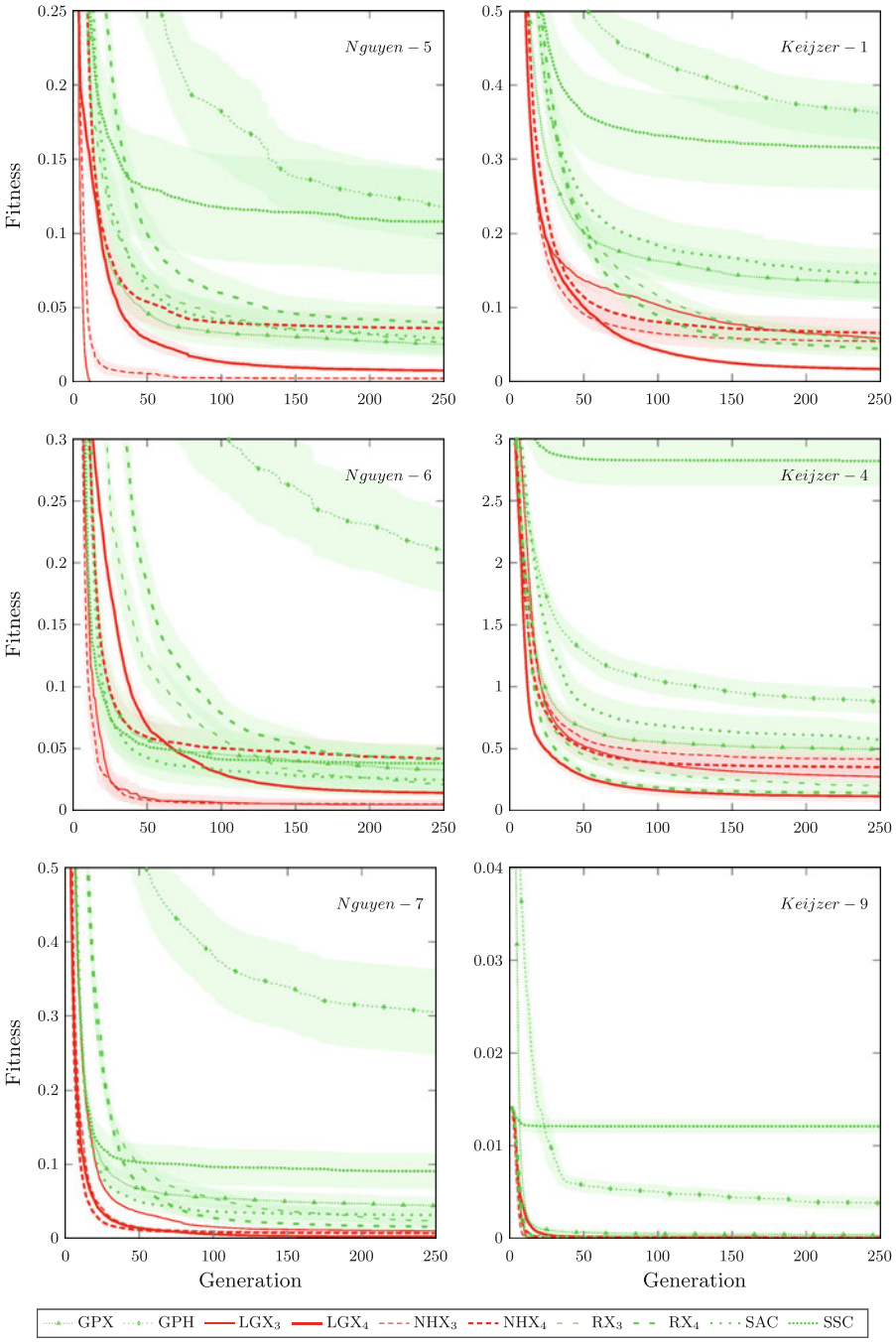

Fig. 3 Best-of-generation fitness averaged over 100 runs with 0.95 confidence intervals (II)

For *Keijzer*-4, LGX_4 is leading for almost entire run, with RX_4, its random counterpart, running up. While *Keijzer*-1 does not significanly discriminate RX_4, LGX_3 and NHX_3, *Keijzer*-4 seems to consistently tell them apart over entire evolution time. The order of plots does not change during evolution (except for LGX_3 and NHX_4, which switched around the 90th generation, however their previous performance was almost the same). This quite complex benchmark (cf. Table 2) seems to be very useful, as opposed to *Keijzer*-9, which is easy for all methods.

To complement this picture, in Table 4 we present the success rate of particular methods, i.e., the percentage of runs that achieve the best-of-run fitness below 10^{-6}. The methods that involve semantic mediality clearly tend to lead with respect to this performance indicator.

5.2.2 *Significance*

To verify whether the observed differences between the methods are significant, we employed Friedman's test for multiple achievements of a series of subjects on the average of best-of-run fitness. The outcome of the test was $p = 2.58 \times 10^{-8}$, which allowed us to reject the null hypothesis and claim that there is at least one significant difference between methods. To find out which pairs of methods differ significantly, a post-hoc analysis was carried out, the results of which we present in Table 5 as the p values of symmetry test for pairs of methods. Figure 4 visualizes the table as an outranking graph, with arcs corresponding to the statistically significant outranking ($\alpha = 0.05$). The graph is clearly bipartite, with the 'good' methods on the one side and 'bad' ones (outranked by at least one other) on the other.

Post-hoc analysis clearly shows that none of the control methods is superior to LGX. This holds for small ($h = 3$) as well as big library ($h = 4$). On the other hand, LGX is significantly better than GPX, SSC, and, in case of LGX_4, to SAC and GPX. There are also some signs that LGX is superior to RX and NHX, however this result does not manifest significant at the number of considered benchmarks.

Table 4 Success rate of the best-of-run individuals

Problem	GPX (%)	GPH (%)	LGX_3 (%)	LGX_4 (%)	NHX_3 (%)	NHX_4 (%)	RX_3 (%)	RX_4 (%)	SAC (%)	SSC (%)
Sextic			**85**	3	31		6	1	3	2
Nonic	**1**									
Nguyen-5	8	3	**100**	1	73	1	6		4	3
Nguyen-6	50	9	**91**	3	78	3	22	1	53	48
Nguyen-7	6				**12**	1			5	2
Keijzer-4				**1**	**1**			**1**		
Keijzer-9	41		**91**	24	71	34	63	56	75	2

The best method marked in bold. Empty spaces mean 0 %. The omitted problems have not been solved by any method

Table 5 Post-hoc analysis of Friedman's test, conducted on best-of-run fitness, visualized as an outranking table

	GPX	GPH	LGX_3	LGX_4	NHX_3	NHX_4	RX_3	RX_4	SAC	SSC
GPX		0.310					0.899	0.899	1.000	0.487
GPH										1.000
LGX_3	0.149	**0.000**			0.980	0.804	0.958	0.958	0.125	**0.000**
LGX_4	**0.010**	**0.000**	0.997		0.582	0.236	0.486	0.486	**0.008**	**0.000**
NHX_3	0.840	**0.002**					1.000	1.000	0.804	**0.006**
NHX_4	0.987	**0.017**			1.000		1.000	1.000	0.980	**0.039**
RX_3		**0.004**								**0.010**
RX_4		**0.004**					1.000			**0.011**
SAC		0.351					0.872	0.871		0.535
SSC										

Values represent probability of erroneously judging the method in row as outranking the method in column. Lower values represent stronger confidence in outranking. The p values in bold represent significant results ($\alpha = 0.05$)

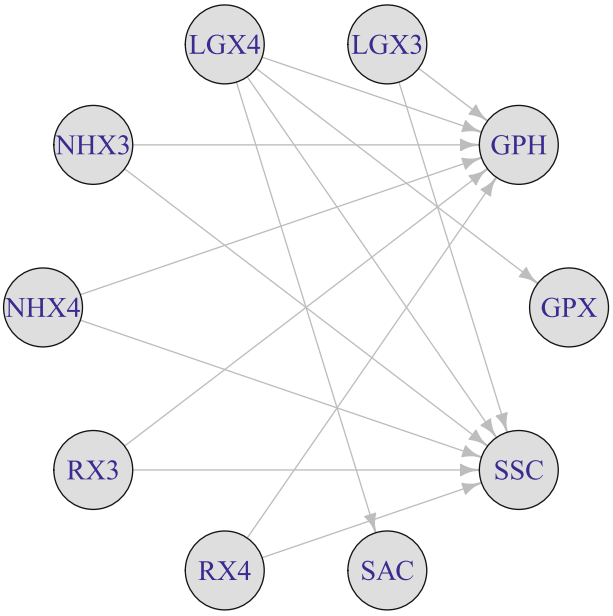

Fig. 4 Outranking graph built upon results of post hoc analysis of Friedman's test

GPH is inferior to almost all other methods, including RX, the other homologous crossover. A possible explanation is that RX constantly supplies the population with diversified genetic material from the library, while GPH cannot enrich the gene pool as it only swaps the subtrees between parents. However, RX does not significantly beat any other operator except for SSC. The columns for RX_3 and RX_4 in Table 5

suggest that RX could be defeated by some other methods (LGX_4, SAC) given larger sample size (number of benchmarks). This lets us conclude that the operators that rely only on homology tend to perform substantially worse than LGX.

NHX performs almost as well as LGX. No other method is significantly better than it. This indicates that relying on semantically medial procedures can be beneficial. However, as opposed to LGX, NHX has problems beating the standard crossover, GPX. Also, LGX shows signs of superiority to NHX, though this difference is insignificant at the number of benchmarks used. This suggests that being semantically medial is good for a crossover operator, although not as good as being medial *and* homologous, a combination of features that is implemented here by LGX. This can be also seen in several fitness plots (Figs. 2, 3). The fact that combination of semantic mediality and homology is synergetic confirms the motivations presented in Sect. 3.1 and is the main result of this study.

The next conclusion is that the methods that rely on semantic mediality (LGX, NHX) produce monotonous fitness plots with narrow confidence intervals (Figs. 2, 3). The variance of best-of-generation fitness individuals at a given generation is for these methods often several times smaller than for the other approaches. This feature, particularly evident for LGX, makes the methods more predictable and allows for approximate performance prediction.

The results for SAC and SSC suggest that using semantic distances between parents' subtrees to decide whether they should be swapped is not very effective. In particular, it is surprising to see that SSC, the more sophisticated operator, yields to many more methods than SAC. We attribute these results to sensitivity of these methods to parameters. Though the parameters specific to SAC and SSC are set as in [35, 45], the other, generic evolutionary parameters common for all methods are set to ECJ's defaults to ensure fair comparison. These defaults are different from the values used in the cited work (see Table 3). Also, concerning the success rates, we employ here a much more demanding definition of success (fitness $<10^{-6}$), compared to the condition used in [35] (error on each single fitness case <0.01).

In general, using a large library ($h = 4$) seems to pay off for LGX only for difficult problems. Apparently, for the easier benchmarks (*Sextic*, *Nguyen*-5, *Nguyen*-6), the semantic diversity provided by the small library of roughly 200 procedures is sufficient for the search to quickly converge to good solutions.

5.3 Performance over time

LGX incurs an additional computational cost of creating the library and searching for the semantically similar procedures. While the former turns out to be low (0.1–6 s compared to 200 s of entire run), the latter cannot be ignored. Every act of crossover requires a *kd*-tree query, which for the assumed crossover probability 0.9 and population size 1,024 implies $0.9 \times (1{,}024/2) = 460$ queries per generation, and 115,200 queries per 250-generations run. This urges us to ask the question: will LGX maintain its performance with the same time allocated to all methods?

To answer this question, in Figs. 5 and 6 we plot the mean fitness of the best-of-generation individual as a function of run time expressed in seconds. For clarity, we use logarithmic time scale, so the label '10^2' marks *half* of the run time, because the

plots tended to overlap and flatten out early when plotted against a linear scale. The plots do not take into account the cost of library building.

The plots reveal that the methods that use the big library ($h = 4$) are the slowest and achieve competitive results only after tens of seconds. The obvious reason for this is the high computational cost of querying the *kd*-tree built upon the library of over 100,000 procedures. However, LGX_3 and NHX_3 that use roughly 500 times fewer procedures exhibit good time-wise performance, leading for the greater part of runs in most benchmarks, proving that a moderately sized library can be queried effectively.

RX_4 uses the library, but does not need to query the *kd*-tree to retrieve procedures. Therefore, it typically converges much faster than its medial counterparts LGX_4 and NHX_4, but rarely manages to maintain this advantage at the end of the allocated time.

The methods that do not use the library are usually fast, and their best-of-generation fitness is typically located between the curves for LGX_3 and LGX_4 for most part of the runs (with *Nonic* and *R3* being exceptions).

To complement this picture, in Table 6 we present the number of generations processed by a single 200-s run, averaged over all benchmarks. LGX_4 and NHX_4 reach only a few dozen of generations, while the remaining methods process several hundreds of them. RX_4, equipped with the same library as LGX_4 and NHX_4 is few times faster than them, since its choice of procedures is purely random and thus there is no need to search the index.

It is interesting to note that *kd*-tree querying is not the only reason for the differences between the library-equipped methods shown in Table 6. The other is the greater average procedure (subtree) height for the larger library. Consider RX_3 and RX_4: neither of them uses *kd*-tree, but the former is about four times faster. This is because the latter inserts higher (on average) procedures into parents, which causes the offspring programs to grow larger, which in turn increases the evaluation time.

The most important result of this experiment is that, despite the extra time needed for querying, the final performance of LGX_3 and NHX_3 are not outstanding from the other methods. In particular, LGX_3 usually achieves the best fitness and typically does so faster than the other methods. The computational effort invested in searching for semantically medial procedure in the library seems to be compensated by the capability of finding well-performing solutions. LGX_4 works noticeably slower, however it has the potential to overtake LGX_3 in longer trials (which could be observed in Figs. 2 and 3). This, together with low variance of fitness, can make LGX attractive not only from theoretical viewpoint, but also in practical scenarios, for instance when it is critical to produce a well-performing solution in a limited time.

5.4 Impact on tree size

The compared crossover operators differ in the way they choose the locus to be affected, and in the source of code fragments they insert into parent solutions. To investigate how this affects the size of evolved programs, we traced the average

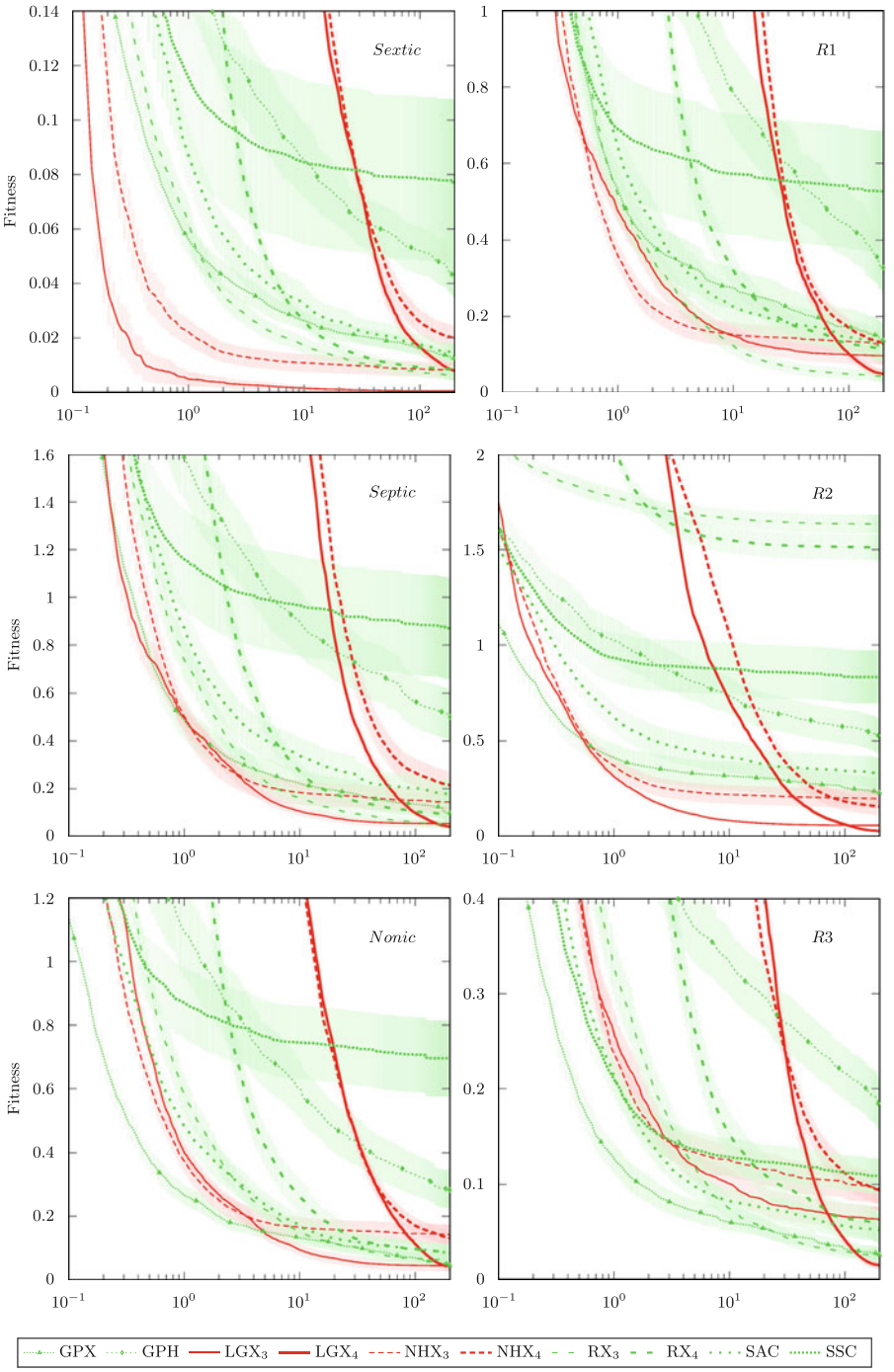

Fig. 5 Best-of-generation fitness as a function of time ($\log_{10}$), averaged over 100 runs (I)

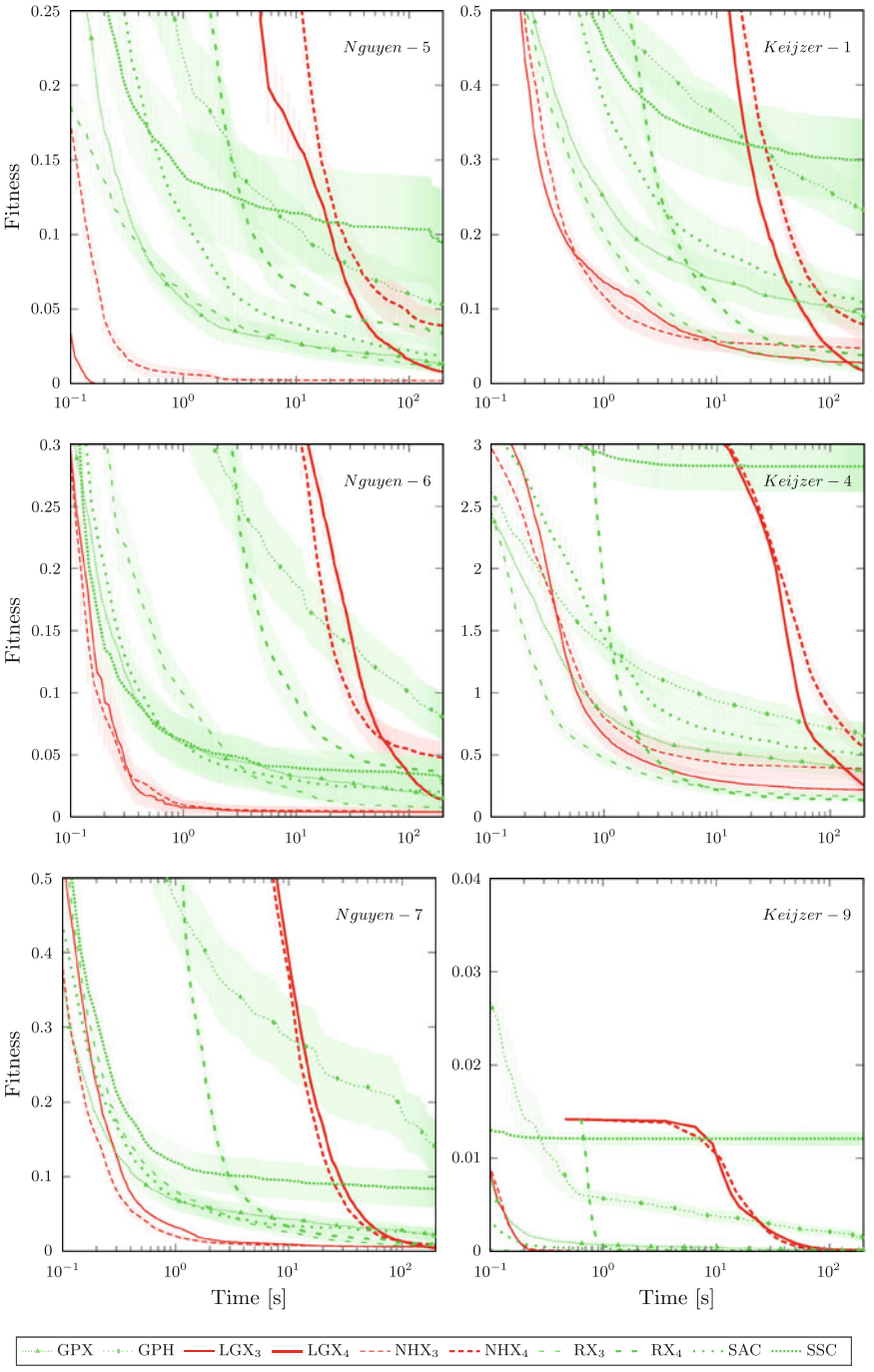

Fig. 6 Best-of-generation fitness as a function of time ($\log_{10}$), averaged over 100 runs (II)

Table 6 Number of generations processed by methods within 200 s (min, max, average and 0.95 confidence interval)

	GPX	GPH	LGX_3	LGX_4	NHX_3	NHX_4	RX_3	RX_4	SAC	SSC
Min	948.0	1650.0	1137.0	56.0	1118.0	57.0	819.0	314.0	421.0	620.0
Avg	2011.0	3739.8	1577.8	85.5	1971.0	80.0	1705.9	427.4	1209.7	2783.3
Cf	±143.8	± 245.6	±69.5	±3.0	±97.3	±1.3	±137.4	±21.7	±105.1	±553.4
Max	5746.0	6738.0	4457.0	151.0	3078.0	97.0	3926.0	1051.0	2981.0	19294.0

number of nodes of individuals in population, and present it with confidence intervals in Figs. 7 and 8.

The main observation resulting from these plots is that they look very alike, particularly concerning the ranking of methods at the end of evolution. In general, RX_4 generates the largest trees, up to 700 nodes. The second is LGX_4, with the final size usually between 400 and 600 nodes. The smallest trees, of 100 nodes on average, are produced by LGX_3, GPH or SSC, depending on the problem. It is interesting to note that, although these three methods seem to be very similar in this respect, LGX_3 typically attains better fitness than GPH and SSC (Sects. 5.2 and 5.3).

In general, the methods that use the big library ($h = 4$) tend to produce larger trees than their counterparts for $h = 3$. This is not surprising, given the fact that the average procedure size in the former library is substantially larger than in the latter.

Despite using the same library, RX produces much bigger trees than LGX. This might suggest that the former selects longer procedures from the library. To verify this, we gathered statistics on the number of tree nodes in inserted procedures for all library-based methods, and show them in Table 7. As the average procedure size turns out to be very similar for RX and LGX, we conclude that the inclination of the former to produce larger trees has to stem from a more sophisticated interplay between semantics of subprograms and the dynamics of evolutionary process, investigation of which is beyond the scope of this study.

Figures 7 and 8 reveal also that NHX produces bigger trees than LGX for $h = 3$, while for $h = 4$ the reverse happens. This is an effect of several factors. On one hand, given a pair of parents p_1 and p_2, LGX, by selecting the loci only from the common region, inserts procedures at shallower locations than NHX. This is because the common region of p_1 and p_2 by definition cannot contain deeper loci than those present in p_1 and p_2. However, by affecting the same locus in the parent solutions, and inserting *the same* procedure at that locus, LGX causes the trees in population to become more and more structurally similar as evolution proceeds (note that the LGX setup does not involve mutation). In a long run, the common region for any two parents in the population is likely to embrace *all* or almost all loci in parents. This in turn allows LGX to operate deeper in the tree, and amplify bloat. The other factor is that, as Table 7 reveals, NHX uses on average slightly smaller procedures than LGX.

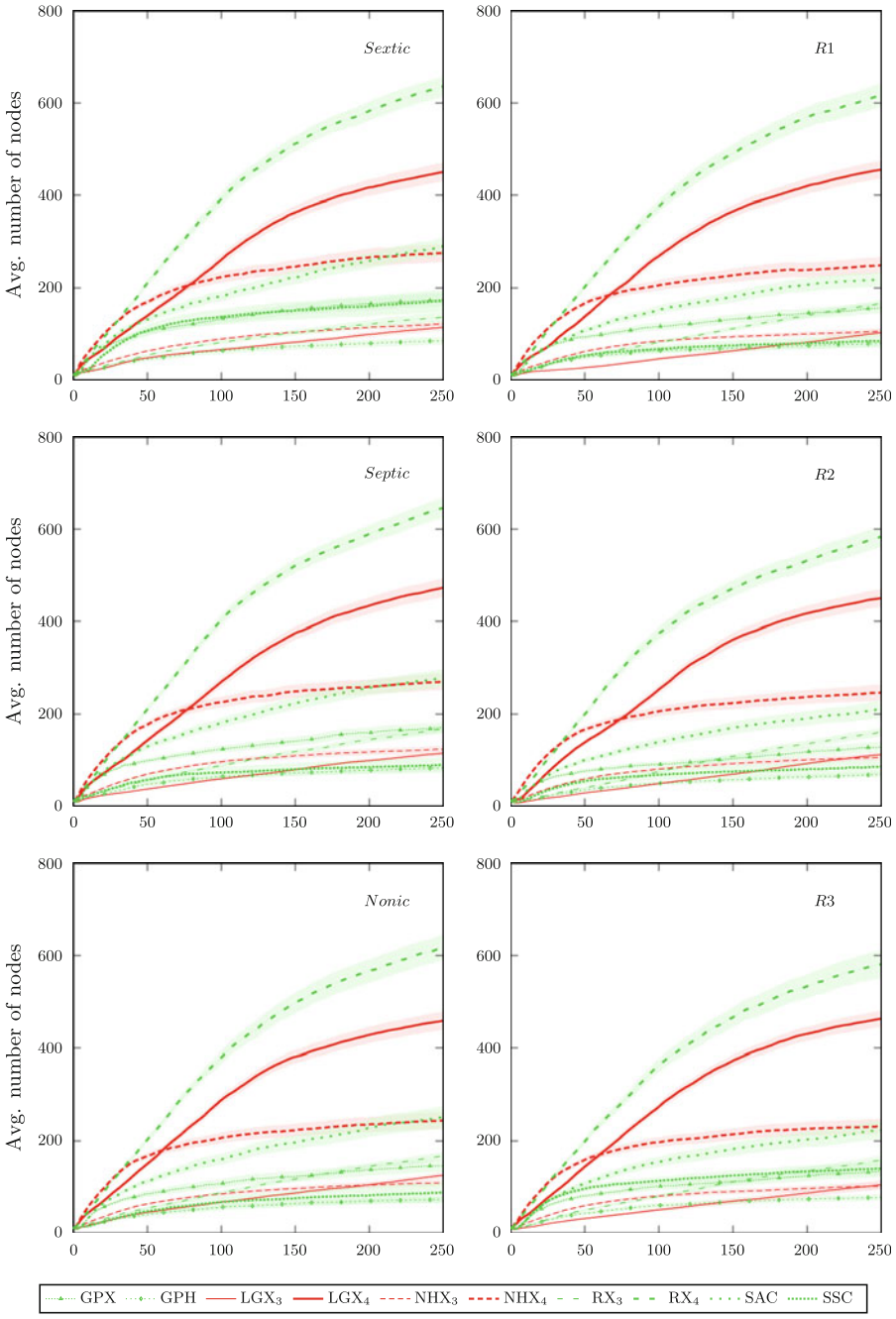

Fig. 7 Tree size over generations (I)

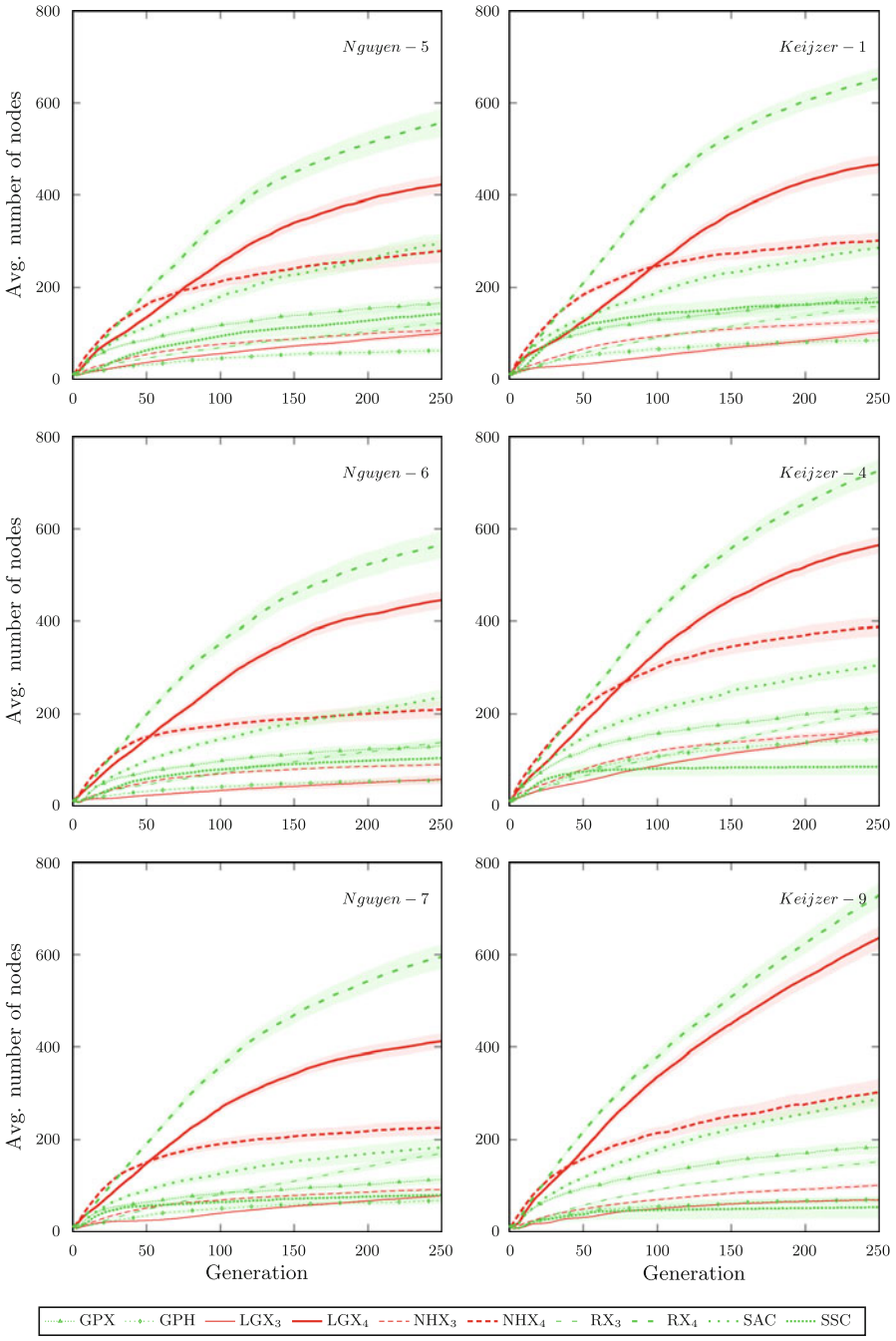

Fig. 8 Tree size over generations (II)

Table 7 The size (number of tree nodes) of the inserted procedure, averaged over all applications of operators during entire evolutionary runs

h	RX_h	LGX_h	NHX_h
3	5.108	5.082	4.876
4	11.062	11.054	10.866

5.5 Generalization to the test set

Large trees produced by some methods, in particular by RX_4 and LGX_4 (Figs. 7, 8), raise concerns about their generalization ability. In machine learning, complex hypotheses are typically associated with higher risk of overfitting. To verify how prone to this risk are the evolved programs, we decided to evaluate the best-of-run individuals on separate test sets of 20 benchmark-specific fitness cases, shown in the last column of Table 2. For each benchmark and run, the best-of-run program is applied to these cases and its quality expressed in the same terms as fitness, i.e., absolute error.

Comparison of the test-set errors presented in Table 8 to the values of fitness achieved at the end of runs (Figs. 2, 3) leads to conclusion that all methods overfit, i.e., commit greater error than on the training set. However, in absolute terms, LGX_3 and LGX_4 attain the lowest errors on the test set for most problems. The variance of error (not shown here for brevity) is also typically the lowest for LGX. The fact that LGX_4 often reaches lower test-set error than the other methods (and the lowest for two benchmarks) is particularly interesting, as the trees it produces typically belong to the largest ones (Figs. 7, 8). Contrary to the common wisdom of machine learning, the trees evolved by LGX, built in a semantically-aware and homologous way, tend to extrapolate better despite being larger. This is consistent with the results obtained in a recent study on controlling bloat using operator equalisation [42], where programs evolving in a population, despite growing in size and improving fitness, tended to keep their average performance on the test set at a constant level.

6 Discussion

The most evident conclusion that can be drawn from the presented experimental evidence is that an important factor that determines the performance of considered operators is the library of procedures. In Fig. 4, the library-equipped operators (LGX, RX, NHX) outrank many other methods, while never being outranked themselves. The likely reason for this is the elimination of semantic duplicates that takes place during library preparation (Sect. 3.3). As it is known from past research (see, e.g., [24], ch. 7), the distribution of semantics of randomly generated trees is very uneven. Elimination of semantic duplicates flattens that distribution and causes all the available semantics of procedures of height up to h to be used equally frequently. Thus, LGX, RX, and NHX are more likely to paste a semantically ‘rare’

Table 8 Errors committed by the best-of-run individuals (as of 250 generation) on test sets (averages over 100 runs)

Problem	GPX	GPH	LGX_3	LGX_4	NHX_3	NHX_4	RX_3	RX_4	SAC	SSC
Sextic	0.024	0.086	**0.002**	0.091	10^{13}	0.044	0.029	0.106	0.092	0.106
Septic	0.207	0.914	**0.096**	0.214	0.197	0.390	0.220	10^{13}	0.366	0.776
Nonic	0.130	0.639	**0.104**	0.217	0.150	0.226	10^{13}	0.828	0.199	0.577
R1	0.261	0.809	0.159	0.181	0.145	0.185	**0.124**	40.32	0.238	0.515
R2	0.316	0.767	0.092	**0.091**	0.245	0.357	10^{5}	10^{13}	0.451	0.958
R3	**0.059**	0.341	0.090	0.144	0.225	0.139	0.238	0.661	10^{13}	0.179
Nguyen-5	0.025	0.118	**0.000**	0.013	0.003	0.040	0.030	10^{13}	0.046	0.092
Nguyen-6	0.033	0.210	**0.004**	0.033	0.004	0.041	0.019	0.129	0.026	10^{13}
Nguyen-7	0.044	0.305	0.008	**0.005**	0.007	0.007	0.043	10.90	0.056	0.085
Keijzer-1	0.134	0.362	**0.092**	0.108	0.106	1.381	0.103	67.36	10^{13}	0.335
Keijzer-4	**0.492**	0.881	1.363	13.27	1.838	10^{13}	1.675	30.24	54.42	10^{13}
Keijzer-9	**0.000**	0.004	0.003	0.592	0.005	0.064	0.159	4.160	0.011	0.192

For errors greater than 100, only the order of magnitude is given

procedure than an operator that would generate an equally big procedure in a purely syntactic way (e.g., using the ramped half-and-half method). This in turn increases the likelihood for the offspring programs to reach previously unexplored semantics, and may help obtaining better fitness.

Nevertheless, LGX outranks more methods than NHX and RX in Fig. 4, and shows some inclination to beat them in Table 5, which suggests that there are some other important factors that shape our results. To identify them, let us reflect on the overall effect of LGX on an evolving population. LGX inserts *the same* procedure into both of parent programs, making them syntactically and semantically identical at the selected locus and all the loci in the subtree rooted at that locus. Thus, over multiple generations, by acting on randomly selected common loci in various pairs of parent solutions, LGX drives the individuals in population to become more and more similar in the syntactic sense and, implicitly, also semantically. In practice however, this never leads to a population-wide convergence. Firstly, LGX chooses one of the $k = 8$ closest semantic neighbors from the library *at random*, so it can produce a different offspring even when applied twice to the same pair of parents. Secondly, a single act of crossover affects only a pair of individuals, so it may take a long time for all individuals in population to converge at a particular locus.

The extent to which individuals in a population evolving under LGX become syntactically similar is however of secondary importance. What matters is that this similarity has certain 'semantic grounding', resulting from the fact that every application of LGX inserts a procedure with semantics that is 'medial' with respect to semantics of the parents' subtrees (Eq. 4). As the experiments demonstrate, this grounding is essential. If it was not, LGX would not prove superior to RX on most benchmarks (Figs. 2, 3). Like LGX, RX is 'convergent' in the sense that it pastes *the same* procedure at a common locus in both parents. Compared to LGX, it does even more intense exploration as it draws a procedure from the entire library, while

LGX chooses one of only $k = 8$ semantically most similar procedures. Yet RX clearly converges more slowly, and in most cases attains worse fitness at the end of run.

Thus, making offspring semantically identical at a homologous locus is only a part of LGX's success. What matters as well is the semantic relationship between the pasted procedure and the subtrees replaced in parents. LGX performs better because it makes the subprograms at the affected locus not only identical, but also semantically medial with respect to parents' subprograms (Eq. 4). This change propagates to the outputs of the offspring programs, with certain likelihood causing them to become semantically more similar to each other (see the discussion on monotonicity in Sect. 3.1). Proving this hypothesis is beyond the scope of this paper; however, in [18], we demonstrated experimentally, for linear programs, that crossover operators that are semantically medial at the level of subprograms tend to produce offspring that are more medial than the offspring built using the standard code-swapping crossovers. That, in turn, causes them perform better. In this way LGX, while explicitly considering only semantic properties of subprograms (i.e., locally), has some features of semantically geometric crossover [30, 32].

The above observations allow us to link these results to the issue of problem decomposition. The convergence of homologously located subprograms towards a common semantics can be viewed as if such subprograms aimed at solving a *subproblem* of the original problem (cf. Sect. 3.1). In other words, LGX enables 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. In this sense, LGX facilitates discovery of subproblems within the problem, which in turn could allow explicit problem decomposition.

Finally, it is interesting to note that, on the genotypic level, labeling LGX a *crossover* operator is questionable, as it does not explicitly *exchange* genetic material between the parent solutions. However, it 'blends' the parents on the semantic, phenotypic level. In this sense, LGX can be regarded as a memetic search operator, but driven by the semantic properties of programs rather than directly by the fitness function.

7 Conclusions

The overall conclusion of this study is that crossover operators that use a library of semantically unique procedures as a source of genetic material improve the efficiency of evolutionary search when compared to other crossover operators (including some semantic-aware ones). Locally geometric semantic crossover (LGX), the operator that uses such a library and is both homologous and semantically medial is the best option among the library-equipped operators considered here, causing the evolved programs to generalize better. The extra computational overhead it involves can be canceled out by its semantically-aware character if the library is not too big. Finally, LGX is not very sensitive to initial conditions (low variance of fitness and test-set performance), which can help avoiding running it multiple times.

LGX and the related concepts have been demonstrated here for programs represented as trees, and verified on symbolic regression benchmarks. However, homologous and semantically medial crossover can be easily generalized. Homology can be meaningfully defined for many program representations (see Sect. 5.3 in [39] for review). Concerning semantic mediality, any domain for which (i) semantics is computable, (ii) semantic metric is available, and (iii) semantic 'blending' can be expressed, can be subjected to this approach. Concerning requirement (i), defining program semantics as a vector of outputs is now widely accepted in GP [44, 34] and technically convenient. Meeting the second requirement is often easy, as fitness function usually *is* a metric, or is based on such. Concerning (iii), although for some semantic spaces (e.g., Hamming space), Eq. (4) cannot be used to explicitly and uniquely determine the midpoint between semantics of parents' subprograms, semantic blending can be expressed in alternative ways, for instance based on triangle inequality or divergence from equidistance, which we studied in [18].

Apart from moving to other problem domains, our future work on this topic will focus on verifying some of the hypotheses formulated in Sect. 6, particularly the supposed binding of semantic subproblems to different loci in program trees. Designing other search operators with the help of the concepts introduced here is yet another interesting research direction. Ongoing preliminary research suggests that an indexed library can serve as a useful source of genetic material for a semantically-aware mutation operator. Next, for the sake of LGX's simplicity, we decided here to populate its library in a very straightforward way. More sophisticated sampling procedures may be considered, e.g., such that provide a more uniform distribution of procedures in the semantic space.

Acknowledgments This study has been supported by National Science Centre Grant No. DEC-2011/01/B/ST6/07318.

References

1. A. Bajurnow, V. Ciesielski, in *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*. Layered learning for evolving goal scoring behavior in soccer players (IEEE Press, Portland, 2004), pp. 1828–1835. URL http://goanna.cs.rmit.edu.au/vc/papers/cec2004-bajurnow.pdf
2. L. Beadle, C. Johnson, in *Proceedings of the IEEE World Congress on Computational Intelligence, IEEE Computational Intelligence Society*, ed. by J. Wang. Semantically driven crossover in genetic programming (IEEE Press, Hong Kong, 2008), pp. 111–116 doi:10.1109/CEC.2008.4630784
3. L. Beadle, C.G. Johnson. Semantic analysis of program initialisation in genetic programming. Genet. Program. Evolvable Mach. **10**(3), 307–337 (2009) doi:10.1007/s10710-009-9082-5. URL http://www.springerlink.com/content/yn5p45723l6tr487
4. L. Beadle, C.G. Johnson, in *IEEE Congress on Evolutionary Computation*, ed. by A. Tyrrell. Semantically driven mutation in genetic programming (IEEE Computational Intelligence Society, IEEE Press, Trondheim, 2009), pp. 1336–1342. doi:10.1109/CEC.2009.4983099

5. J.L. Bentley, Multidimensional binary search trees used for associative searching. Commun. ACM. **18** (1975)
6. M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest, R.E. Tarjan, Time bounds for selection. J. Comput. Syst. Sci. (1973)
7. J.H. Friedman, J.L. Bentley, R.A. Finkel, An algorithm for finding best matches in logarithmic expected time. ACM Trans. Math. Softw. **3**(3), 209–226 (1977)
8. E. Galvan Lopez, R. Poli,C.A. Coello Coello, in *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings, LNCS*, eds. by M. Keijzer, U.M. O'Reilly, S.M. Lucas, E. Costa, T. Soule. Reusing code in genetic programming, vol. 3003, (Springer, Coimbra, 2004), pp. 359–368. URL http://delta.cs.cinvestav.mx/ccoello/conferences/eurogp04.pdf.gz
9. D.E. Goldberg, *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. (Kluwer, Norwell, 2002)
10. T. Haynes, in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, eds. by J.R. Koza, K. Deb, M. Dorigo, D.B. Fogel, M. Garzon, H. Iba, R.L. Riolo. On-line adaptation of search via knowledge reuse, (Morgan Kaufmann, Stanford University, CA, USA 1997) p. 156–161. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.3381
11. G.S. Hornby, J.B. Pollack, Creating high-level components with a generative representation for body-brain evolution. Artif. Life **8**(3):223–246 (2002) doi: 10.1162/106454602320991837. URL http://www.demo.cs.brandeis.edu/papers/hornby_alife02.pdf
12. D. Howard, in *Genetic Programming Theory and Practice*, chap. 10, eds. by R.L. Riolo, B. Worzel. Modularization by multi-run frequency driven subtree encapsulation (Kluwer, The Netherland, 2003) pp. 155–172.
13. W.H. Hsu, S.J. Harmon, E. Rodriguez, C. Zhong, in *Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference*, ed. by M. Keijzer. Empirical comparison of incremental reuse strategies in genetic programming for keep-away soccer (Seattle, Washington, 2004). URL http://www.cs.bham.ac.uk/wbl/biblio/gecco2004/LBP010.pdf
14. D. Jackson, in *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010, LNCS*, eds. by A.I. Esparcia-Alcazar, A. Ekart, S. Silva, S. Dignum, A.S. Uyar. Phenotypic diversity in initial genetic programming populations, vol. 6021, (Springer, Istanbul, 2010), pp. 98–109. doi:10.1007/978-3-642-12148-7_9
15. W. Jaskowski, K. Krawiec, B. Wieloch, in *GECCO '07: Proceedings of the 9th Annual Conference on Genetic and evolutionary computation*, eds. by D. Thierens, H.G. Beyer, J. Bongard, J. Branke, J.A. Clark, D. Cliff, C.B. Congdon, K. Deb, B. Doerr, T. Kovacs, S. Kumar, J.F. Miller, J. Moore, F. Neumann, M. Pelikan, R. Poli, K. Sastry, K.O. Stanley, T. Stutzle, R.A. Watson, I. Wegener. Genetic programming for cross-task knowledge sharing, vol. 2, (ACM Press, London, 2007) pp. 1620–1627. doi: 10.1145/1276958.1277281. URL http://www.cs.bham.ac.uk/wbl/biblio/gecco2007/docs/p1620.pdf
16. K.E. Kinnear Jr., in *Proceedings of the 1994 IEEE World Conference on Computational Intelligence*. Fitness landscapes and difficulty in genetic programming, vol. 1, (IEEE Press, Orlando, 1994) pp. 142–147. doi:10.1109/ICEC.1994.350026. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/kinnear.wcci.ps.Z
17. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. (MIT Press, Cambridge, 1992)
18. K. Krawiec, in *Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012, LNCS* vol. 7244, eds. by A. Moraglio, S. Silva, K. Krawiec, P. Machado, C. Cotta. Medial crossovers for genetic programming (Springer, Malaga, Spain 2012), pp. 61–72 doi:10.1007/978-3-642-29139-5_6
19. K. K.Krawiec, P. Lichocki, in *GECCO '09: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, eds. by G. Raidl, F. Rothlauf, G. Squillero, R. Drechsler, T. Stuetzle, M. Birattari, C.B. Congdon, M. Middendorf, C. Blum, C. Cotta, P. Bosman, J. Grahl, J. Knowles, D. Corne, H.G. Beyer, K. Stanley, J.F. Miller, J. van Hemert, T. Lenaerts, M. Ebner, J. Bacardit, M. O'Neill, M. Di Penta, B. Doerr, T. Jansen, R. Poli, E. Alba. Approximating geometric crossover in semantic space (ACM, Montreal 2009), pp. 987–994. URL http://doi.acm.org/10.1145/1569901.1570036
20. K. Krawiec, T. Pawlak, in *GECCO '12: Proceedings of the 2012 GECCO Conference Companion on Genetic and Evolutionary Computation*, Locally geometric semantic crossover, (ACM, Philadelphia, 2012), (to appear)

21. K. Krawiec, B. Wieloch, Analysis of semantic modularity for genetic programming. Found. Comput. Decisi. Sci. **34**(4), 265–285 (2009). URL http://fcds.cs.put.poznan.pl/FCDS2/ArticleDetails.aspx?articleId=219
22. K. Krawiec, Wieloch, B, in *GECCO 09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, eds. by G. Raidl, F. Rothlauf, G. Squillero, R. Drechsler, T. Stuetzle, M. Birattari, C.B. Congdon, M. Middendorf, C. Blum, C. Cotta, P. Bosman, J. Grahl, J. Knowles, D. Corne, H.G. Beyer, K. Stanley, J.F. Miller, J. van Hemert, T. Lenaerts, M. Ebner, J. Bacardit, M. O'Neill, M. Di Penta, B. Doerr, T. Jansen, R. Poli, E. Alba. Functional modularity for genetic programming (ACM, Montreal 2009) pp. 995–1002. URL http://doi.acm.org/10.1145/1569901.1570037
23. K. Krawiec, B. Wieloch, in *IEEE Congress on Evolutionary Computation (CEC 2010)*. Automatic generation and exploitation of related problems in genetic programming, (IEEE Press, Barcelona, 2010). doi:10.1109/CEC.2010.5586120
24. W. Langdon, R. Poli, *Foundations of Genetic Programming*. (Springer, Berlin, 2002)
25. D.T. Lee, C.K. Wong, Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. Acta. Inform. **9**, 23–29, (1977)
26. S. Luke, *The ECJ Owner's Manual—A User Manual for the ECJ Evolutionary Computation Library*, zeroth edition, online version 0.2 edn. (2010). URL http://www.cs.gmu.edu/eclab/projects/ecj/docs/manual/manual.pdf
27. J. McDermott, D.R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaśkowski, K. Krawiec, R. Harper, K.D. Jong, U.M. O'Reilly, in *GECCO '12: Proceedings of the 2012 GECCO Conference on Genetic and Evolutionary Computation*. Genetic programming needs better benchmarks, (ACM, Philadelphia, 2012). (to appear)
28. N.F. McPhee, B. Ohs, T. Hutchison, in *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008, Lecture Notes in Computer Science*, vol. 4971, eds. by M. O'Neill, L. Vanneschi, S. Gustafson, A.I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, E. Tarantino. Semantic building blocks in genetic programming (Springer, Naples, 2008), pp. 134–145. doi:10.1007/978-3-540-78671-9_12
29. T. Mitchell, *Machine Learning*. (McGraw-Hill, New York, 1997)
30. A. Moraglio, *Towards a geometric unification of evolutionary algorithms*. Ph.D. thesis, Department of Computer Science, University of Essex, UK (2007). URL http://eden.dei.uc.pt/moraglio/Thesis_final.pdf
31. A. Moraglio, in *Foundations of Genetic Algorithms*, eds. by H.G. Beyer, W. Langdon. Abstract convex evolutionary search (ACM, Schwarzenberg, Austria 2011), pp. 151–162. doi:10.1145/1967654.1967668
32. A. Moraglio, K. Krawiec, C. Johnson, in *The 5th Workshop on Theory of Randomized Search Heuristics, ThRaSH'2011*, eds. by C. Igel, P.K. Lehre, C. Witt. Geometric semantic genetic programming (Copenhagen, Denmark 2011). URL http://www.thrash-workshop.org/slides/moraglio.pdf
33. A. Moraglio, R. Poli, in *Genetic and Evolutionary Computation—GECCO-2004, Part I, Lecture Notes in Computer Science*, vol. 3102, K. Deb, R. Poli, W. Banzhaf, H.G. Beyer, E. Burke, P. Darwen, D. Dasgupta, D. Floreano, J. Foster, M. Harman, O. Holland, P.L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, A. Tyrrell. Topological interpretation of crossover (Springer, Seattle, 2004), pp. 1377–1388. URL http://privatewww.essex.ac.uk/amoragn/gecco2004fin.PDF
34. A. Moraglio, S. Silva, in *GECCO '11: Proceedings of the 13th Annual Conference on Genetic and evolutionary computation*, eds. by N. Krasnogor, P.L. Lanzi, A. Engelbrecht, D. Pelta, C. Gershenson, G. Squillero, A. Freitas, M. Ritchie, M. Preuss, C. Gagne, Y.S. Ong, G. Raidl, M. Gallager, J. Lozano, C. Coello-Coello, D.L. Silva, N. Hansen, S. Meyer-Nieberg, J. Smith, G. Eiben, E. Bernado-Mansilla, W. Browne, L. Spector, T. Yu, J. Clune, G. Hornby, M.L. Wong, P. Collet, S. Gustafson, J.P. Watson, M. Sipper, S. Poulding, G. Ochoa, M. Schoenauer, C. Witt, A. Auger. Geometric nelder-mead algorithm on the space of genetic programs (ACM, Dublin, Ireland 2011), pp. 1307–1314. doi:10.1145/2001576.2001753
35. Q.U. Nguyen, X.H. Nguyen, M. O'Neill, in *Proceedings of the 12th European Conference on Genetic Programming, EuroGP 2009, LNCS*, vol. 5481, eds. by L. Vanneschi, S. Gustafson, A. Moraglio, I. De Falco, M. Ebner. Semantic aware crossover for genetic programming: The case for real-valued function regression (Springer, Tuebingen 2009), p. 292–302. doi:10.1007/978-3-642-01181-8_25
36. M. Pelikan, *Hierarchical Bayesian Optimization Algorithms*. (Springer, Berlin, 2005)

37. R. Poli, W.B. Langdon, *Genetic programming with one-point crossover and point mutation. Tech. Rep. CSRP-97-13*, (University of Birmingham, School of Computer Science, Birmingham, B15 2TT, UK, 1997). URL ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1997/CSRP-97-13.ps.gz
38. R. Poli, W.B. Langdon, Schema theory for genetic programming with one-point crossover and point mutation. Evol. Comput. **6**(3), 231–252 (1998). doi: 10.1162/evco.1998.6.3.253. URL http://cswww.essex.ac.uk/staff/poli/papers/Poli-ECJ1998.pdf
39. R. Poli, W.B. Langdon, N.F. McPhee, *A field guide to genetic programming*. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk (2008). URL http://www.gp-field-guide.org.uk. (With contributions by J. R. Koza)
40. J.P. Rosca, D.H. Ballard, in *Advances in Genetic Programming 2, chap. 9*, P.J. Angeline, K.E. Kinnear, Jr. Discovery of subroutines in genetic programming, (MIT Press, Cambridge, 1996) pp. 177–202. URL ftp://ftp.cs.rochester.edu/pub/u/rosca/gp/96.aigp2.dsgp.ps.gz
41. C. Ryan, M. Keijzer, M. Cattolico, in *Genetic Programming Theory and Practice II, chap. 7*, eds. by U.M. O'Reilly, T. Yu, R.L. Riolo, B. Worzel. Favorable biasing of function sets using run transferable libraries. (Springer, Ann Arbor 2004), pp. 103–120. doi:10.1007/0-387-23254-0_7
42. S. Silva, in *GECCO '11: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, eds. by N. Krasnogor, P.L. Lanzi, A. Engelbrecht, D. Pelta, C. Gershenson, G. Squillero, A. Freitas, M. Ritchie, M. Preuss, C. Gagne, Y.S. Ong, G. Raidl, M. Gallager, J. Lozano, C. Coello-Coello, D.L. Silva, N. Hansen, S. Meyer-Nieberg, J. Smith, G. Eiben, E. Bernado-Mansilla, W. Browne, L. Spector, T. Yu, J. Clune, G. Hornby, M.L. Wong, P. Collet, S. Gustafson, J.P. Watson, M. Sipper, S. Poulding, G. Ochoa, M. Schoenauer, C. Witt, A. Auger. Reassembling operator equalisation: a secret revealed (ACM, Dublin, Ireland 2011), pp. 1395–1402. doi:10.1145/2001576.2001764
43. H. Simon, *The Sciences of the Artificial*. (MIT Press, Cambridge, 1969)
44. N.Q. Uy, N.X. Hoai, M. O'Neill, R.I. McKay, E. Galvan-Lopez, Semantically-based crossover in genetic programming: application to real-valued symbolic regression. Genet. Program. Evolvable Mach. **12**(2), 91–119 (2011) doi:10.1007/s10710-010-9121-2
45. N.Q. Uy, M. O'Neill, X.H. Nguyen, B. McKay, E.G. Lopez, in *9th International Conference, Evolution Artificielle, EA 2009, Lecture Notes in Computer Science*, vol. 5975, P. Collet, N. Monmarche, P. Legrand, M. Schoenauer, E. Lutton. Semantic similarity based crossover in GP: The case for real-valued function regression (Springer, Strasbourg, France 2009), pp. 170–181. doi: 10.1007/978-3-642-14156-0