Genet Program Evolvable Mach
DOI 10.1007/s10710-015-9253-5

Subtree semantic geometric crossover for genetic programming

Quang Uy Nguyen[1] · Tuan Anh Pham[2] · Xuan Hoai Nguyen[3] · James McDermott[4]

Received: 19 December 2014 / Revised: 23 September 2015
© Springer Science+Business Media New York 2015

Abstract The semantic geometric crossover (SGX) proposed by Moraglio et al. has achieved very promising results and received great attention from researchers, but has a significant disadvantage in the exponential growth in size of the solutions. We propose a crossover operator named subtree semantic geometric crossover (SSGX), with the aim of addressing this issue. It is similar to SGX but uses subtree semantic similarity to approximate the geometric property. We compare SSGX to standard crossover (SC), to SGX, and to other recent semantic-based crossover operators, testing on several symbolic regression problems. Overall our new operator out-performs the other operators on test data performance, and reduces computational time relative to most of them. Further analysis shows that while SGX is rather exploitative, and SC rather explorative, SSGX achieves a balance between the two. A simple method of further enhancing SSGX performance is also demonstrated.

✉ Quang Uy Nguyen
quanguyhn@gmail.com

Tuan Anh Pham
anh.pt204@gmail.com

Xuan Hoai Nguyen
nxhoai@gmail.com

James McDermott
jmmcd@jmmcd.net

[1] Faculty of IT, Le Quy Don Technical University, Hanoi, Vietnam

[2] Institute of IT, Military Academy of Logistics, Hanoi, Vietnam

[3] IT Research Centre, Hanoi University, Hanoi, Vietnam

[4] Lochlann Quinn School of Business, University College Dublin, Dublin, Ireland

Published online: 26 October 2015

Keywords Genetic programming · Semantics · Geometric crossover · Symbolic regression

1 Introduction

Genetic programming (GP) [18, 37] is an evolutionary paradigm with the objective of evolving programs that produce desired outputs for predefined inputs. To improve the performance of GP, a number of researchers have recently attempted to integrate semantic methods into GP. This integration typically happens by definition of new operators, which can be divided into two types: indirect and direct [39]. Indirect methods achieve desired semantic criteria by sampling and rejection. They alter the syntax of the parents [19, 23, 33, 34]. In contrast, direct methods generate new children with desired semantic criteria which incorporate the complete syntax of their parents without modification [29, 38]. Although recently proposed by Moraglio et al. [29], direct methods have received substantial interest from the GP research community [7, 31, 38]. Perhaps the main contribution of the approach is that it allows GP to search directly in the semantic space.

Although semantic geometric crossover (SGX) has been empirically shown to be superior to standard crossover (SC) in several studies [7, 29, 38], it has a shortcoming in the exponential growth of solution size. This exponential code growth can potentially hinder applications of SGX for several reasons. First, running GP with SGX requires excessive hardware resources and long running time. Second, the solutions found by SGX might be arbitrarily large and complex. While the first problem has been partly alleviated in recent implementations of SGX, e.g. [28, 38], the second remains. Since the objective of GP, in contrast to other black-box learning approaches, is to find solutions in the form of comprehensible structures, its usefulness in many applications often relies on the simplicity and understandability of the generated programs. For instance, in medical applications the goal might be to understand the relationships between genes as they affect particular diseases [10]. In other applications, the produced programs may have to run in an environment with limited resources such as wireless sensor networks [6]. In these applications, small and simple synthesized solutions are required. Therefore, finding small and comprehensible solutions is of great importance for the usefulness of GP.

In order to lessen the code growth problem in SGX, some GP researchers [22, 36] have used a version of the operator in which the geometric property is approximated, using a library of sub-programs of known semantics. However, these methods may increase the computation time of the system due to the library search process. In this paper, we introduce another way to address code growth in SGX by proposing a new semantic geometric crossover that works on subtrees. The new crossover not only helps to greatly reduce the complexity of the evolved solutions, compared to SGX, but also to further enhance GP performance. The main contributions and findings of the paper can be summarized as follows:

- A new semantic-based crossover, called subtree semantic geometric crossover (SSGX), is proposed and its performance is investigated. The experimental

results show that this operator helps to alleviate code growth in SGX and also to improve its performance.
- Further analysis shows that the geometric operator of Moraglio et al., SGX, has exploitative, local search behaviour, while the SC and other semantic-based operators have more global, explorative search. SSGX is the only operator of those tested that possesses both abilities (exploitation and exploration).
- Analysis of computation time shows that while using an additional library as in the random desired operator (RDO) and approximate geometric crossover (AGX) of Krawiec et al. [22, 36] increases the execution time of GP systems considerably, the overhead of SSGX is negligible.
- The design of SSGX allows for further improvements. In this paper, a simple method for improving SSGX is also introduced that leads to better performance.

In the next section, we present background. A brief review of the previous work on semantic operators in GP is given in Sect. 3. The proposed crossover, SSGX, is described in Sect. 4. The experimental settings are detailed in Sect. 5. The performance of SSGX is examined and compared with other crossover operators in Sect. 6. Section 7 analyses some crucial properties of the tested operators. The impact of parameters on the performance of SSGX is investigated in Sect. 8. In Sect. 9, a simple way to enhance the SSGX's performance is introduced and examined. Section 10 concludes the paper and highlights some potential future work.

2 Background

This section presents some necessary background for the research in this paper. First, a way to measure semantics is introduced. Next, a semantic distance is defined. Finally, a brief description of SGX [29] is presented.

2.1 Measuring semantics

The meaning of the term "semantics" varies between fields. In GP it is common to define the semantics of a program simply as its behaviour with respect to a set of input values. In this paper, we follow previous GP research [23, 26, 29, 34, 36] in defining the semantics of a program (individual) as its outputs for the problem's fitness cases. Formally, the semantics of a program is defined as follows:

Definition 1 Let $K = (k_1, k_2, \ldots k_N)$ be the fitness cases of the problem. The *program semantics* $S(P)$ of a program P is the vector of output values obtained by running P on all fitness cases.

$$S(P) = (P(k_1), P(k_2), \ldots, P(k_N)), \quad \text{for } i = 1, 2, \ldots, N.$$

This definition is valid for problems where a set of fitness cases is defined.

2.2 Semantic distance

Based on the above definition, a semantic distance between two trees or subtrees is often defined as the distance between their corresponding output vectors with respect to the vector of fitness case inputs. In this paper, we use the Manhattan metric normalized by the number of fitness cases of the problem as the semantic distance for two individuals.

Definition 2 The *semantic distance (SD)* between two trees (P_1 and P_2) is defined as follows.

$$SD(P_1, P_2) = \frac{|S(P_1) - S(P_2)|}{N} = \frac{\sum_{i=1}^{N} |P_1(k_i) - P_2(k_i)|}{N} \quad (1)$$

This definition is valid for programs whose output is a single real-valued number, as in symbolic regression.

2.3 Semantic geometric crossover

As mentioned, Moraglio et al. [29] proposed geometric semantic operators (crossover and mutation) for several problem domains in. Their idea is rooted in the unifying geometric theory of evolutionary algorithms [27]. The motivation for these operators is to allow GP to directly search in the semantic space. Formally, the geometric crossover (SGX) for real-valued symbolic regression is defined as follows [29].

Definition 3 Given two parent functions P_1, P_2: $\mathbb{R}^n \rightarrow \mathbb{R}$, the *geometric semantic crossover* returns the real function $P_3 = (T_R P_1) + ((1 - T_R)P_2)$ where T_R is a random real constant in [0, 1] or a random real function with codomain [0, 1].

If T_R is a random real constant, the crossover produces an offspring that lies on the line segment connecting the two parents in the semantic space with respect to the Euclidean distance. This is a "thin" line segment. However, if T_R is a random real function, the offspring lies on the line segment between the parents with respect to Manhattan distance. This is a "thick" line segment. Some previous research has shown that the Manhattan version of SGX performs better [7, 30], so that is the version investigated in this paper. The choice of T_R as a function rather than a constant here is linked to the choice of Manhattan rather than Euclidean distance for semantic distance and to the choice of mean absolute error for the fitness function, as opposed to root mean square error.

The SGX operator generates offspring that contain the complete structure of both parents. Therefore, the size of a child is more than the sum of the sizes of its parents. The exponential growth of the size of individuals makes this operator impractical in a naive implementation, though this may be partly addressed using automated simplification of offspring as in Moraglio et al. [29] or by implementation with

caching as proposed in [28, 38]. In Sect. 4, an alternative solution to this problem is proposed.

The SGX operator depends on several assumptions about the problem. As stated, it defines semantics as a vector of output values with respect to a pre-determined set of inputs, which may not be appropriate to all problems. It depends for its geometric property on the output being in a suitable metric space, and on the GP language being "functional", i.e. without side effects or program state. All of these assumptions are fulfilled for important problem classes including symbolic regression, but they do represent a limitation on SGX. The same is true of several other modern semantic crossover operators, including those in this paper. Alternative definitions of semantics which may allow the lifting of this limitation have been used in GP also [12, 16].

3 A review of semantic crossovers in GP

Semantics has been used in various ways in GP. In one strand of research, semantic information was often represented in the form of attribute grammars which can be used to eliminate individuals of poor fitness from the population [9] or to prevent the generation of semantically invalid individuals, e.g. [8, 41]. Later, Johnson advocated formal methods as a means to incorporate semantic information into the GP evolutionary process [13–15]. In Johnson's work, semantic information extracted through formal methods is used to quantify the fitness of individuals on some problems where the traditional sample-point-based fitness measure is unavailable or misleading.

The idea of defining the semantics of an individual as its output with respect to a set of inputs was perhaps first proposed by McPhee et al. [26]. They extracted semantic information from a Boolean expression tree by enumerating all possible inputs [26]. The semantic information was then used to analyse the semantic diversity of the population during the search process. Recently, semantic information has generally been used to design or guide genetic operators in GP. Thus, we will mainly focus our review on this use of semantics. A more detailed and comprehensive survey of semantic methods in GP is given by Vanneschi et al. [39].

Semantic methods for GP operators can be classified into two types: direct and indirect [39]. Direct methods effectively act directly on the semantics of individuals [29, 38], while indirect methods achieve their semantic goals indirectly by acting on syntax and then applying semantically defined survival criteria [19, 23, 33, 34].

In indirect methods, Beadle and Johnson [3] extracted semantics from expression trees on Boolean domains. They checked the semantic equivalence of the offspring produced by crossover with their parents and discarded the offspring if equivalent to their parents. This approach enhances semantic diversity in the evolving population, and consequently leads to improvements in GP performance [3]. The method of semantic equivalence checking is also applied to drive mutation [5] and guide the initialisation phase of GP [4], with a positive effect on performance.

Considering indirect semantic operators in real-valued domains, Nguyen et al. [32] proposed a crossover operator, called semantics aware crossover (SAC), aiming

to promote semantic diversity. SAC was based on checking for semantic equivalence of subtrees. SAC was then extended to semantic similarity based crossover (SSC) [34] to improve semantic locality. SSC selected subtrees for crossing over by checking semantic similarity but not semantic equivalence. In [33], the improved version of SSC, called the most semantic similarity based crossover (MSSC), was proposed, which better guarantees the semantic locality of crossover in GP. The performance of SSC and MSSC were shown to be superior to both SC and SAC [33, 34].

In other work, Krawiec et al. [20] used semantic information to guide crossover in a method that is similar to Soft Brood Selection (SBS) [1], known as approximating geometric crossover (AGC). A number of children are generated by a crossover operation, and the child most semantically similar to the parents is added to the next generation. Another semantic (subtree) crossover was proposed in [21, 23], namely, locally geometric semantic crossover (LGX). In LGX, two subtrees in the common shape of the parents are randomly chosen. Then a search is performed for a subprogram in a predefined library that is semantically intermediate to the selected subtrees. This subprogram (subtree) replaces the two chosen subtrees to generate two new offspring.

Among direct methods, Moraglio et al. [29] proposed an entirely new approach to designing semantic based crossovers in GP. Unlike other subtree crossovers, in SGX the offspring is created based on a convex combination of its parents. The crossover was tested on a class of problems and showed to perform very well. However, SGX can be very time- and memory-consuming as it keeps all parents in memory, accumulating their complexity (number of nodes) via the convex combination. More recently, Vanneschi at al. proposed a new implementation of this crossover that helps to reduce the space and time requirements, as claimed by the authors, to acceptable levels [38]. Although recently proposed by Moraglio et al. [29], direct methods have received substantial interest from the GP research community [7, 24, 31, 38]. For a comprehensive review and comparison of different geometric semantic crossovers in GP, the readers are recommended to see [35].

In an attempt to reduce the code growth observed with SGX, Krawiec and Pawlak [22] proposed AGX. The main idea behind AGX is to replace subtrees in parents with subtrees such that the offspring is semantically intermediate to the parents. This objective is obtained by first calculating the midpoint m of the semantic of parents. Then two crossover points p_1 and p_2 in the parents are selected. Last, a backpropagation procedure is called to find the semantics of the subtree (st) that when substituted for p_1 and p_2 will alter the semantics of the parents to approximate m. A library is used to find a subtree that is as close to st as possible. The experiments showed that the performance of AGX is superior to both SC and LGX on a wide range of problems [22, 36].

The idea of semantic backpropagation is then extended for designing operators in GP [36]. A new operator called RDO was proposed [36]. RDO used the same principle of backpropagation as AGX but with different desired semantics. In RDO, a parent is chosen by a selection scheme. Then a random subtree st is chosen. After that, backpropagation is again used to identify the semantics of the subtree that when substituted for st will produce a new child that matches the target semantics of

the problem (the target output of the training data). This operator performs very well on both real-valued and Boolean problems as reported in [36].

Although both AGX and RDO have been empirically shown to have superior performance to both LGX and SC, they have some limitations. First, these operators can be time consuming as a result of the backpropagation and library search processes. Second, RDO is not fully black-box: it is not usable if the target semantics of the problem is not explicitly defined (or calculated), as is common, for example, in some reinforcement learning and security problems [36]. Moreover, on problem domains like classification where the semantics of an individual may not be strongly correlated to the target semantics of the problem, the performance of RDO may suffer.

In the next section, we will present a novel crossover operator aiming to approximate the geometric semantic crossover operator, SGX. This crossover is implemented at the subtree level, leading to the reduction of code growth in SGX. Consequently, the new operator does not lead to code growth (compared to SGX) and executes rather faster (compared to AGX and RDO), while still achieving competitive performance.

4 Subtree semantic geometric crossover

The new crossover operator is called SSGX and is similar to the crossover of Moraglio at al. [29] except that it is implemented at the subtree level. It proceeds as follows. Two parents P_1 and P_2 and a probability value ϵ are selected. If a randomly generated number $R \in [0, 1]$ is less than ϵ, the novel geometric subtree crossover is performed. Otherwise, we execute the standard (subtree) crossover. This step means that a certain proportion of crossovers is executed by SC. In early experiments we implemented 100 % geometric subtree crossover and found that the population collapsed very quickly to individuals of very small sizes. For complex problems, if the individuals do not grow enough, the performance of GP deteriorates significantly.

In the cases where the novel geometric subtree crossover is conducted, a number (given by the parameter *MaxTrial*) of subtrees that satisfies a size constraint are randomly selected in P_1, excluding P_1 itself. We denote by St_1 the one of these that is most semantically similar to P_1. The objective of selecting St_1 in this way is to replace a parent of high fitness by a smaller subtree which approximates its semantics. The size constraint aims to control the code growth of the offspring. In this paper, we use a simple constraint where only a subtree with size in the range of $[\alpha, \beta]$ is chosen. The lower bound (α) aims to avoid the selection of very small subtrees (e.g. leaf nodes), that may disrupt the structure of the parents. The upper bound (β) is used to limit code growth.

We select St_2 in P_2 in the same way as St_1. Two offspring[1] are then generated by convex combination of the subtrees St_1 and St_2. In other words, C_1 and C_2 are the

[1] The reason for generating two offspring but not one as in the original version of SGX is to allow both SC and geometric crossover are executed in SSGX. Moreover this implementation makes SSGX consistent with conventional subtree-swapping crossovers.

two children, generated as: $C_1 = T_R St_1 + (1 - T_R) St_2$ and $C_2 = (1 - T_R) St_1 + T_R St_2$. T_R is a random function with codomain [0, 1], created by generating a random tree of maximum depth 2 and passing the result through the logistic function. Effectively, St_1 and St_2 approximate the semantics of P_1 and P_2 and are used instead of them. P_1 and P_2 themselves are not used in the crossover. If St_1 and St_2 were to duplicate the semantics of P_1 and P_2 precisely, then the semantic effect would be the same as that of the geometric semantic crossover proposed by Moraglio et al. [29], but with less code growth. In practice, St_1 and St_2 will only approximate the semantics of P_1 and P_2. Hence we can think of the procedure as a geometric semantic crossover preceded by a type of heuristic and inexact simplification, analogous to the exact post-operator simplification proposed by Moraglio et al. [29]. The details for SSGX are presented in Algorithm 1, where SD is the semantic distance between the two individuals as defined in Sect. 2 and *SizeOf*(*St*) function return the number of nodes of tree *St*.

Algorithm 1: Geometric Semantic Subtree Crossover. Parameters: P_1, P_2: parent programs, ϵ: threshold.

```
generate a random number R
if R< ε then
    Max=ExtremalValue;
    for Count=1 to MaxTrial do
        choose an arbitrary subtree St1 in P1 that satisfies size constraint;
        D=SD(St1, P1)
        if D<Max then
            Max=D;
            Subtree1=St1;
    Max=ExtremalValue;
    for Count=1 to MaxTrial do
        choose an arbitrary subtree St2 in P2 that satisfies size constraint;
        D=SD(St2, P2)
        if D<Max then
            Max=D;
            Subtree2=St2;
    execute geometric crossover based on Subtree1 and Subtree2;
Else execute standard subtree crossover
```

In Sect. 8 several values of *MaxTrial* will be tested to examine its impact on the performance of SSGX. Comparing to SGX [29], this crossover has some interesting properties. First, the geometric property is approximated at the subtree level, potentially helping to reduce the offspring size while still maintaining some of the advantages of geometric crossover (the results in Sect. 6 provide evidence for this). Second, only a certain portion of SSGX crossovers use the geometric method, i.e. the convex combination of subtrees. Therefore, we can use any version of subtree crossover such as SSC, MSSC [33, 34] or LGX [19] for the rest. In Sect. 9, we will investigate whether this can further enhance the performance of SSGX.

It is possible that a subtree selected as St_1 or St_2 in one generation will later be selected again to serve as St_1 or St_2 in a later generation, because each crossover works by preserving St_1 and St_2 as a subtree of the offspring. If this happens frequently, search may be inefficient due to revisiting old genetic material.

However, the probability of this occurring is not high, because in our experiments, only a certain portion (30 %) is geometric crossover, and the remainder (70 %) is SC. Mutation is also used to modify individuals. These reduce the probability that a subtree is reselected.

5 Experimental settings

In order to investigate the performance of SSGX, we tested it and several other operators on 10 regression problems. Amongst the tested problems, there are 6 GP benchmark problems taken from [25], originally proposed by Keijzer [17], and four real-valued problems drawn from the UCI repository [2]. The detailed descriptions of the problems are presented in Table 1.

The GP parameters used for our experiments are given in Table 2. The function set includes eight functions widely used in GP [18, 40]. The protected versions of division (/) and logarithm (log) function were used that return 1 when the denominator in the division is zero, or the argument in the logarithm is zero or less. The terminal set for each problem includes N variables X_1 to X_N corresponding to the number of variables (attributes) of the problems. The raw fitness is the sum absolute error on all fitness cases. Therefore, smaller values are better.

For all GP systems, the crossover rate is 0.9 and mutation rate is 0.1. A common version of SC [18] is used, in which the crossover point is executed with 90 % at nonterminal nodes, 10 % at leaf nodes. Standard subtree mutation is used on all GP

Table 1 All problems for testing different operators

Name	Definition	Training data	Testing data
(a) GP Benchmark regression problems			
Keijzer-4	$x^3 e^{-x} \cos(x) \sin(x)(\sin^2(x)\cos(x) - 1)$	R[0,10,100]	E[0.05,10.05,100]
Keijzer-6	$\sum_i^x \frac{1}{i}$	R[1,50,100]	E[1,120,100]
Keijzer-11	$xy + \sin((x-1)(y-1))$	R[-1,1,100]	E[0,1,100]
Keijzer-12	$x^4 - x^3 + \frac{y^2}{2} - y$	R[-1,1,100]	E[0,1,100]
Keijzer-14	$\frac{8}{2+x^2+y^2}$	R[-1,1,100]	E[0,1,100]
Keijzer-15	$\frac{x^3}{5} + \frac{y^3}{2} - x - y$	R[-1,1,100]	E[0,1,100]

Name	Attribute numbers	Training data	Testing data
(b) UCI regression problems			
Concrete slump test	7	50	53
CCPP	4	200	200
Wine quality red	11	250	250
Wine quality white	11	300	300

Table 2 Evolutionary parameter values

Parameter	Value
Population size	500
Generations	100
Selection	Tournament
Tournament size	7
Crossover probability	0.9
Mutation probability	0.1
Initial Max depth	6
Max depth	15
Max depth of T_R	3
Raw fitness	Mean absolute error on all fitness cases
Trials per treatment	30 independent runs for each value
Library size for AGX and RDO	1000
Mutation step size for SGX	0.001
Function Sets	+, −, *, /, sin, cos, exp, log

systems but SGX. The parameters for SSGX is as follows: *MaxTrial=20*, $\epsilon = 0.3$, $\alpha = 1$ and $\beta = 80$. These values were calibrated from early experiments for good performance. In Sect. 8, these parameters will be further examined. In SGX we used geometric semantic mutation by Moraglio et al. [29] since SGX performs better when accompanied with this mutation [29]. The mutation step size for SGX was set to 0.001, which was found to be the best value by Moraglio et al. [29]. We used the caching implementation described in [38] to reduce the computation time. The library size for RDO and AGX was set at 1000. This library was obtained by randomly generating 1000 semantically different individuals with max depth of 4. The library size of 1000 is intermediate between two tested values in previous literature [35]. For each problem and each parameter setting, 30 runs were performed.

We divided our experiments into four sets. The first aims to investigate the performance of SSGX in comparison with SC and other semantic-based crossovers. The second aims to analyse the solution size, execution time, and semantic effects of the tested operators. The third set attempts to test how sensitive the performance of SSGX is when its parameters are varied. Lastly, in the fourth set, we propose a scheme to further enhance the performance of SSGX. The results of these sets of experiments are detailed in the following sections.

6 Performance analysis

This section aims to analyse the performance of the new operator (SSGX) and to compare it with the SC and some recently proposed semantic-based crossovers. The tested semantic-based operators include SGX [29], AGX [22] and RDO [36].[2] In

[2] The list of all operators tested in this paper is presented in "Appendix 1".

order to compare these operators, three metrics including training error, testing error and running time were used.

The first analysis is the mean best fitness on the training data. These values are presented in Table 3.[3] In this table, K-I is short for Keijzer-I problem (I = 4, 6,...), and UCI-I is the abbreviation for the four UCI regression problems in Table 1 respectively (UCI-1 is Concrete Slump Test, ..., UCI-4 is Wine Quality White). For each problem, the smallest value among the five tested operators is printed in bold face in this table. Statistical tests are presented later.

It can be seen in Table 3 that all tested operators other than SGX achieved a smaller training error than SC. The training error of SGX was not always consistent. The value for SGX was considerably smaller than for SC on one problem (K-11), noticeably greater on four problems (K-4, K-14, K-15, UCI-1) and almost the same on others. This result seems surprising given previous good training error with SGX [29, 38]. However, the result is consistent with the result in a recent publication [35] where Pawlak et al. also reported poor performance of SGX on real-valued regression problems. One possible explanation is that SGX performs differently on different problems. Another is that SGX requires an unusually large number of generations (possibly with a smaller population) [29]. Further analysis (Sect. 7) also showed that while SGX more frequently produced children that are better than their parents in terms of the training fitness, the scale of semantic change from parents to children in SGX was much smaller than that of SC. Subsequently, this crossover moves rather slowly to the global optimum (best solution) in the semantic space. This supports the hypothesis that SGX requires an unusually large number of generations.

In contrast to SGX, other operators often achieved smaller values of the training error than SC. Amongst three operators (AGX, RDO and SSGX), RDO is the best and SSGX is the second best crossover, regarding training error. The performance on the training data of RDO and SSGX is often better than AGX and much better than other operators. The table shows that RDO is the "winner" (has the smallest training error) in eight out of ten problems (SSGX winning in one and SGX in one problem). Noticeably, increasing the library size in AGX and RDO may potentially achieve better training errors. However, even with a library of size 1000, it was found that the computation time for both AGX and RDO was considerably higher compared to SC and SSGX (around 8–10 times higher than SC and SSGX – see Table 6). Therefore, a greater library size seems prohibitive, especially in applications such as stream data mining or online learning.

The second metric used to measure the performance of the tested operators is their ability to generalize beyond the training data. The generalization ability is perhaps the most desirable property of a learner. In each run, the best solution was selected and evaluated on an unseen data set (the testing set). The median of these values across 30 runs was calculated and are shown in Table 4. This table is consistent with Table 3, confirming the inferior performance of SGX and the superiority of AGX, RDO and SSGX to SC. On the tested problems, the ability of

[3] In "Appendix 3", figures presenting the training error of the tested operators during the evolutionary process are shown.

Table 3 Mean best fitness on training data

XOvers	K-4	K-6	K-11	K-12	K-14	K-15	UCI-1	UCI-2	UCI-3	UCI-4
SC	4.86	2.21	7.13	10.3	6.88	2.99	200	2085	109	175
SGX	6.84	2.26	4.94	9.44	9.63	4.62	279	2079	**102**	170
AGX	6.35	1.47	6.92	6.20	7.51	1.89	216	748	113	175
RDO	**0.81**	**1.28**	**4.23**	**2.50**	5.50	**0.86**	**121**	**509**	109	**152**
SSGX	4.03	1.61	5.89	4.72	**3.32**	2.02	173	896	107	174

Lower is better and the best value on each problem is printed bold face

Table 4 Median of testing error of the tested operators

XOvers	K-4	K-6	K-11	K-12	K-14	K-15	UCI-1	UCI-2	UCI-3	UCI-4
SC	19.0	5.19	18.3	11.0	24.5	7.84	442	2085	220	231
SGX	20.7	58.2	37.9	39.6	90.3	64.5	515	49,150	565	657
AGX	3.69	**1.03**	9.96	12.6	16.2	4.70	**379**	**833**	239	209
RDO	**0.99**	1.15	**6.21**	8.16	12.7	**1.85**	441	841	240	217
SSGX	3.69	1.42	7.67	**6.27**	**8.59**	2.40	495	1072	**203**	**205**

Lower is better and the best value on each problem is printed bold face

SGX to generalize was often much worse than SC. Perhaps the poor performance SGX on the testing data is due to the fact that solutions obtained by SGX have very high complexity (see Sect. 7), which likely allowed overfitting according to the principle of Occam's razor [11]. However, Vanneschi et al. [38] argue that overfitting is bounded in SGX.

In contrast to SGX, three operators (SSGX, AGX, RDO) performed very well, compared to SC, on unseen data. The test error of the solutions produced by these operators was often much smaller than those of SC except on the UCI-1 problem. On UCI-1, while the testing error of AGX was smaller than SC, the value for both SSGX and RDO was worse. This suggests that on this problem both SSGX and RDO have overfitted. For comparison among SSGX, AGX and RDO, it can be seen that SSGX and RDO were often better than AGX with respect to generalization. SSGX was the winner (the operator that achieves the best result on the testing data) in four out of ten problems. AGX and RDO were each the winner in three problems.

For statistical comparison between multiple operators we used a Wilcoxon signed rank test with a Bonferroni correction factor of 100 (10 pairwise crossovers and 10 problems). The results are shown in Table 5 (Table 5a for for training error and Table 5b for testing error). In Table 5, if the operator in the row is significantly better (p-value $<0.05/100$) than the operator in the column on problem t, the problem t is presented in the corresponding cell. For instance, on the training data, RDO is significantly better than SC on all problems except UCI-3, so "All-{UCI-3}" is presented in the cell at row "RDO" and column "SC" of Table 5a.

On the training data, SC is superior to SGX on three problems (K-14,K-15,UCI-1) while SGX is not better than SC on any problem. AGX is significantly better than

Table 5 Statistical testing conducted on training error and testing error using Wilcoxon signed rank test and a Bonferroni correction factor of 100 (10 pairwise operators and 10 problems)

XOvers	SC	SGX	AGX	RDO	SSGX
(a) Results of statistical test on training error					
SC		K-14,K-15,UCI-1	UCI-3		
SGX			UCI-3		
AGX	K-6,UCI-2	K6,K-12,K-14 K-15,UCI-1,UCI-2			UCI-2
RDO	All-{UCI-3}	All-{UCI-3}	All		K-4,K-12,K-15 UCI-1,UCI-2,UCI-4
SSGX	K-12,K-14 UCI-1,UCI-2	All-{K-11,UCI-3}	K-14 UCI-1,UCI-3	K-14	
(b) Results of statistical test on testing error					
SC		All-{K-4,UCI-1}			
SGX					
AGX	K-4,K-6,K-11 UCI-2,UCI-4	All			
RDO	K-4,K-6,K-11 UCI-2,UCI-4	All-{UCI-1}	K-4		K-4
SSGX	All-{UCI-1,UCI-3}	All-{UCI-1}	K-12,K-14 UCI-3,UCI-4	UCI-3 UCI-4	

In the cells are the problems where the operator in the row is significantly better (p-value $< 0.05/100$) than the operator in the column

SC on K-6,UCI-2 and better than SGX on six out of ten problems. However, AGX is only better than SSGX on one problem (UCI-2) and not better than RDO on any tested problem. Conversely, RDO and SSGX are more frequently better than SC, SGX and AGX. Comparing between RDO and SSGX, Table 5 shows that RDO is better than SSGX on six problems while the converse is true in one. Apparently, RDO is the best crossover among the five tested operators.

On the test data, the performance of SSGX is more convincing. SSGX is significantly better than SC on all problems except UCI-1, UCI-3, and better than SGX on all problems except UCI-1. The test also shows that SSGX is better than AGX and RDO on four and two problems respectively, while RDO is better than SSGX on one problem (K-4) and AGX is not better than SSGX on any tested problem. The performance of AGX and RDO are also solid compared to SC and SGX on the unseen data. Both AGX and RDO are significantly better than SC on five out of ten problems and they are better than SGX on all tested problems except UCI-1 with RDO. SGX is the crossover that performed the worst on the test data. Overall, the results of the statistical tests show that although SSGX is the second best crossover on training data, it achieved better performance than all other crossovers on test data.

The last metric used to measure the performance of the five tested operators is their computational time. All operators were executed on the same computing

Table 6 Average running time of the tested operators measured in seconds

XOvers	K-4	K-6	K-11	K-12	K-14	K-15	UCI-1	UCI-2	UCI-3	UCI-4
SC	49.1	41.5	24.9	25.3	23.4	25.8	34.6	32.9	28.9	29.5
SGX	13.2	12.8	9.02	8.2	8.18	8.3	7.52	7.15	7.12	8.36
AGX	407	315	242	226	218	187	192	196	105	162
RDO	272	236	229	215	221	193	205	274	186	210
SSGX	39.7	36.8	27.7	26.2	25.6	26.6	26.6	25.1	28.3	30.5

Lower is better

platform (Operating system: Ubuntu 10.04 LTS, RAM 4G, CPU E7500 2.93GHz $\times$ 2). The source code of our GP system is available for download.[4] The computational time of a run measured by CPU time was recorded and the values were averaged over 30 runs. These values are presented in Table 6. SGX is the only operator that had running time considerably less than SC. This is not surprising since in these experiments, the implementation of SGX by Vanneschi et al. was used [7, 38]. In this implementation, caching prevents re-evaluation of individuals, considerably reducing execution time.

The running time of SSGX and SC is roughly equal. SSGX did not require longer running time compared to SC although it has an additional step of subtree searching. The result in the next section shows that by limiting the size of selected subtrees, the code growth with SSGX is less than with SC, which helps to limit runtime also. In contrast to SSGX, the two other semantic based crossovers required longer running time than SC. The running time of both AGX and RDO were much higher than other operators. Both AGX and RDO were 8 to 10 times larger in running time compared to SC and SSGX. This increase in computation time for AGX and RDO is one of the limitations of these operators. In some application domains such as online learning or stream data mining where the time constraint is critical, the slow execution of AGX and RDO might limit their usefulness. In other scenarios, the long training time may not be important.

Overall, the empirical results in this section show that the proposed crossover operator, SSGX, helps to enhance the performance of GP on unseen data in comparison with SC and some recently proposed semantic-based operators. Moreover, this superior performance (over SC and SGX) was achieved with much less computational time compared to AGX and RDO. Therefore, this crossover might be more suitable than AGX and RDO in application domains where time or computing resources are limited.

7 An analysis of semantic based operators

This section aims to analyse some crucial properties of SSGX and compare them with those of the other tested operators. The properties examined include the average size of the solutions, the average execution time of different phases in SSGX, AGX and RDO,

[4] https://github.com/jmmcd/GP-SSGX.

Table 7 Average size of solutions of the tested operators

XOvers	K-4	K-6	K-11	K-12	K-14	K-15	UCI-1	UCI-2	UCI-3	UCI-4
SC	150	117	101	103	87.6	90.1	261	190	147	164
SGX	10^{31}	10^{30}	10^{30}	10^{30}	10^{30}	10^{30}	10^{31}	10^{31}	10^{30}	10^{31}
AGX	173	102	89.5	80.6	46.1	54.9	119	147	16.2	82.6
RDO	201	147	109	104	89.0	76.1	174	252	46.1	154
SSGX	110	99.1	96.8	102	93.6	95.6	104	96.5	102	110

Lower is better

the semantic distance between children and their parents and the constructive rate of these operators. The results are detailed in the following subsections.

7.1 Solution size of the tested operators

The main design objective of SSGX is to limit the code growth phenomenon of SGX, helping to obtain simpler solutions. Therefore, it is important to examine if this objective is achieved. In each run, the best solution was selected and its size was recorded. These values were then averaged over 30 runs and the results are shown in Table 7.[5]

It can be seen from this table that SSGX achieved its main objective. The solutions obtained by SSGX were much smaller than those of SGX. While the solution size of SGX was very high and unacceptable in many real-world applications, the size of the solutions produced by SSGX was acceptably small. Interestingly, the solutions found by SSGX are frequently smaller than those of SC. There are three possible reasons explaining why SSGX achieve its objective in reducing solution size. The first is that only 30 % of SSGX uses the geometric combination, while the rest (70 %) uses SC. Second, the subtree selected for combination in SSGX was also limited by its size. Third, SSGX has a maximum depth of 15 whereas SGX has no such limit.

The table also shows that GP systems using AGX and RDO found solutions with sizes that are roughly equal and slightly higher than those of the GP system using SSGX except on UCI-3. On UCI-3, RDO and AGX achieved very small solutions. We investigated the reason behind this and found that these operators often achieved the best solution in the form of *function(constant)* e.g. $e^{1.609438}$ or $\log(-148.413159)$ after some generations. Perhaps this is the result of the constant search in AGX and RDO. In general, Table 7 demonstrates that implementing geometric crossover at the subtree level helps SSGX to find much less complicated solutions compared to SGX.[6]

7.2 Running time analysis of the tested operators

The results in Sect. 6 show that GP systems using AGX, RDO required longer execution time than the GP system using SC and SSGX. This subsection aims to

[5] The size of the solutions obtained by SGX in Table 7 is presented approximately.

[6] A sample tree output using SSGX with some remarks on its structure is given in "Appendix 2".

Table 8 Average running time of different phases in SC, AGX, RDO and SSGX

XOver	Phases	K-4	K-6	K-11	K-12	K-14	K-15	UCI-1	UCI-2	UCI-3	UCI-4
SC		49.1	41.5	24.9	25.3	23.4	25.8	34.6	32.9	28.9	29.5
AGX	LibGen	0.3	0.3	0.2	0.3	0.3	0.3	0.2	0.3	0.6	0.2
	Inversion	88.2	67.0	88.6	69.4	72.1	69.5	63.8	36.5	20.6	49.3
	LibSearch	271	235	123	136	127	113	108	127	92.7	112
	Total	407	315	242	226	218	187	192	196	135	162
RDO	LibGen	0.3	0.3	0.2	0.3	0.3	0.3	0.2	0.3	0.6	0.2
	Inversion	101	86.7	102	84.6	94.9	56.2	61.7	94.5	88.2	85.3
	LibSearch	134	117	97.1	119	108	114	125	123	94.8	106
	Total	272	236	229	215	221	193	205	274	186	210
SSGX	SubSearch	5.2	4.8	4.9	4.3	4.1	3.9	4.0	3.7	4.2	5.2
	Total	39.7	36.8	27.7	26.2	25.6	26.6	26.6	25.1	28.3	30.5

give a further analysis on the reasons behind this. It can be seen that for AGX and RDO, there are three extra stages that can potentially slow their execution time. These stages are: library generation, inversion, and searching in the library for a desired subtree. On the other hand, for SSGX, there is only one extra step, that is to search for a subtree (within a parent) that has as close semantics as possible to its parent and that satisfies the constraint on size. We measured the computational time of each stage in AGX, RDO and SSGX. The values were then averaged over the whole population and all runs. They are presented in Table 8.

In this table, there are four rows presenting the results of AGX and RDO. The rows "LibGen", "Inversion", and "LibSearch" present the average running time of library generation, inversion, and library search stages, respectively. For SSGX, the first row ("SubSearch") is the average running time for the subtree searching phase. The row "Total" presents the average running time of these operators (AGX, RDO and SSGX). For the sake of comparison, the results of SC are also presented.

The table shows that while the library generation stage in AGX and RDO was relatively fast, the inversion and the library search were both computationally expensive. The average running time of the inversion stage was often triple that of the whole evolutionary process of the GP systems using SC and SSGX. The values of the library search were five to six times higher than SC. Subsequently, both AGX and RDO were usually 8 to 10 times slower than SC and SSGX in our experiments. By contrast, the subtree searching produce in SSGX was rather fast. This process was equivalent to just 15–20 % of the average running time of the GP system using SC. This allowed SSGX to run as fast as SC on the tested problems.

7.3 Semantic distance between children and parents

The next property analysed is the semantic distance between parents and their children. The semantic distance measures distance between a pair of individuals (e.g. a parent and a child) in the semantic space. It is informative as to the balance between exploration and exploitation behaviour of the search. A larger distance

Table 9 Average distance semantics in various operators between children and parents

XOvers	K-4	K-6	K-11	K-12	K-14	K-15	UCI-1	UCI-2	UCI-3	UCI-4
SC	2.77	1.25	0.16	2.67	1.70	2.16	8.22	129	6.23	9.80
SGX	0.008	0.018	0.010	0.014	0.020	0.011	0.112	1.421	0.034	0.039
AGX	0.21	0.52	0.22	0.15	0.26	0.56	1.41	4.18	0.20	0.29
RDO	0.39	1.08	0.21	0.24	0.36	0.18	5.15	7.61	0.16	0.62
SSGX-Stan	3.41	6.25	0.82	1.24	2.71	0.97	56.2	152	12.5	14.5
SSGX-Geo	0.17	0.48	0.23	0.30	0.65	0.16	5.01	63.2	1.04	1.48
SSGX-Ave	2.62	4.21	0.62	1.07	2.03	0.80	39.8	117	8.33	9.35

Table 10 Average of constructive rate of the tested operators

XOvers	K-4	K-6	K-11	K-12	K-14	K-15	UCI-1	UCI-2	UCI-3	UCI-4
SC	5.63	7.18	5.88	7.02	5.08	7.24	12.7	8.59	10.3	11.0
SGX	85.6	85.2	88.4	89.2	90.2	86.5	69.3	67.6	89.1	86.3
AGX	8.12	10.7	9.31	10.5	11.2	12.0	14.9	20.6	18.2	17.5
RDO	7.09	8.25	7.47	8.41	9.35	8.01	12.92	19.2	15.7	13.1
SSGX-Stan	9.28	9.71	9.72	9.26	9.71	9.35	12.7	13.2	12.5	12.2
SSGX-Geo	5.64	10.1	4.75	6.32	7.21	6.35	8.58	7.91	8.11	8.06
SSGX-Ave	8.97	9.85	7.21	8.01	8.34	8.69	10.6	11.2	11.7	10.9

shows the ability to discover different areas in the search space, while a smaller move is evidence for the convergence of the algorithm to a specific subspace. Formally, the semantic distance between children and their parents is calculated as follow:

$$SD_X(P_1, P_2; C_1, C_2) = \frac{SD(P_1, C_1) + SD(P_2, C_2)}{2} \tag{2}$$

where P_1, P_2 are two parents for crossover and C_1, C_2 are the two children resulting from the crossover(C_1 has the same root as P_1 and C_2 has the same root as P_2).

This definition applies to all subtree crossover operators. Since RDO is similar to mutation (it has a single parent and a single offspring), for the above definition to be valid, P_2, C_2 and the factor of 2 are omitted. For SGX, since neither of C_1 and C_2 has the same root as either parent, a random child is selected and assigned as C_1 and the other is considered as C_2. The semantic distance is calculated and averaged over the number of crossovers, then over generations and the number of runs. The results are shown in Table 9. In this table (and also Table 10 below), there are three rows for the results of SSGX. The first row (SSGX-Stan) presents the results of the SC proportion in SSGX. The second row (SSGX-Geo) corresponds to the results of geometric crossover in SSGX and the last row is the average over both.

It can be seen that the semantic distance values between children and parents in SGX were very small. These values for SGX were often a hundred times smaller than those of SC. Small values are expected in that they suggest that the offspring

are semantically medial with respect to their parents, however they may also indicate slow search progress when using SGX. This could be one of the reasons for the unconvincing performance of SGX on some problems as shown in Sect. 6. All other operators created greater semantic change from parents to children. However, the scale of change also varied among them. The SC made greater moves in the semantic space than both AGX and RDO. Therefore, it seems that both AGX and RDO possess better semantic locality than SC, and their superior performance to SC might be attributed to this [33].

Perhaps the most interesting results in this table are those of SSGX. This is only the operator that possesses both exploration and exploitation ability simultaneously. Since SSGX includes both a SC and a geometric crossover component, they accomplish different tasks during the search process.[7] While the SC portion of operations aims to explore the search space, the geometric operator portion concentrates on exploiting specific areas. This balance provides a partial explanation for the convincing performance of SSGX compared to the other tested operators.

7.4 Constructive rate of the tested operators

The last property to be investigated is the constructive rate of the five tested operators. The constructive rate reflects how often an operator produces a child that is better than its parent in terms of the training error. With the same notation as in the definition of semantic distance, the constructive rate of an operator in each generation is calculated as follows:

$$Constructive\ rate\ (Op) = \frac{Count_{11} + Count_{12} + Count_{21} + Count_{21}}{4.(Number\ of\ crossovers)} \tag{3}$$

where $Count_{11}$ and $Count_{12}$ are the number of the crossovers in which C_1 is better than P_1 and C_1 is better than P_2 respectively. Similarly, we calculate $Count_{21}$, $Count_{22}$ for C_2. Op is the tested operator. For RDO, since it has a single parent generating a single offspring, $Count_{12}$, $Count_{21}$, $Count_{22}$ and the factor of 4 in the denominator are omitted. The values were averaged over all generations and over 30 runs. They are given in Table 10. This table shows that SGX was the only operator that frequently improved the fitness of individuals on the training data. The constructive rate of SGX was from 80 to 90 %. However, due to the fact that this crossover usually made a small change in the (semantic) search space, its performance on the training data was not always convincing.

For the other operators, the constructive rate was much less. These values of SC were usually <10 % on the tested problems. For AGX and RDO, the constructive rate was often slightly better than those of SC. These values of RDO are not as high as the results in [36] where the authors reported that RDO achieved about 70 % constructive rate even with a small library size. The reason for the different results could be that in [36], the constructive rate of RDO was calculated at only the first

[7] The result of SC in SSGX, SSGX-Stan is not identical to the result of SC. One possible reason is due to they are being executed in two populations with different diversity and structure. Future research will further examine this.

generation whereas in this paper, the constructive rate was averaged over all generations. The constructive rate values for SSGX was also slightly higher than those of SC, although the difference is marginal. Perhaps the better performance of SSGX compared to SC is due to its ability to execute both exploration and exploitation research (see Sect. 7.3).

8 Analysis of SSGX

As already discussed in Sect. 4, there are three factors that can potentially affect the performance of SSGX. The first is the number of trials (*MaxTrial*) to find a subtree that is semantically close to its parent. The second parameter is the proportion (ϵ) in which the geometric operator is executed. The last is the value of lower bound (α) and upper bound (β) that were used to control the size of the selected subtrees in SSGX. This section aims to analyse the impact of these factors on the performance of SSGX.

8.1 Impact of parameters to SSGX

This subsection examines the effect of two first parameters (*MaxTrial* and ϵ) on SSGX performance. To do this, we set these parameters with a starting value and then gradually increase it. Five values for each parameter were tested. For *MaxTrial*, they are 10, 15, 20, 25 and 30. For ϵ, five tested values are 0.1, 0.2, 0.3, 0.4, 0.5. When varying the value of *MaxTrial*, we fixed ϵ at 0.3. Similarly, when we tested various values of ϵ, we set *MaxTrial* at 20. The performance (the mean of the best fitness) of SSGX with different values of *MaxTrial* and ϵ are presented in Tables 11 and 12, respectively. The performance with SC is also shown for reference.

We can see from Table 11 that the training error of SSGX seems not to be very sensitive to *MaxTrial*. This value was not remarkably changed when the value of *MaxTrial* was increased from 10 to 30. A statistical test using Wilcoxon signed rank test and with a Bonferroni correction factor of 100 was used to verify if there is any significant difference (p-value $<0.05/100$) in SSGX's training error when *MaxTrial* was varied. The results of the test show that none of these differences is significant. However, the table shows that setting the value of *MaxTrial* at 20 helped SSGX to perform slightly more consistently than at other values and therefore it was used to

Table 11 Performance of SSGX (mean best training fitness) with different values of *MaxTrial*

MaxTrial	K-4	K-6	K-11	K-12	K-14	K-15	UCI-1	UCI-2	UCI-3	UCI-4
SC	6.00	2.24	9.06	13.7	9.61	4.08	210	2174	112	177.6
10	4.11	1.61	5.25	4.83	3.89	2.15	176	927	108	172
15	3.98	1.62	5.58	4.61	3.93	2.05	162	913	107	171
20	4.03	1.61	5.89	4.72	3.32	2.02	173	896	107	174
25	4.02	1.68	5.72	4.93	3.40	2.12	162	906	105	175
30	4.74	1.72	6.16	5.09	3.76	2.80	170	901	108	174

Table 12 Performance of SSGX (mean best training fitness) with various values of ϵ

ϵ	K-4	K-6	K-11	K-12	K-14	K-15	UCI-1	UCI-2	UCI-3	UCI-4
SC	6.00	2.24	9.06	13.7	9.61	4.08	210	2174	112	177
0.1	4.86	1.69	5.98	4.78	3.62	2.70	173	883	105	167
0.2	4.97	1.59	5.07	4.98	3.00	2.14	178	867	103	166
0.3	4.03	1.61	5.89	4.72	3.32	2.02	173	896	107	174
0.4	4.48	2.15	5.63	5.57	3.29	2.96	170	894	106	171
0.5	5.45	2.37	7.34	7.91	4.40	3.01	181	929	106	172

measure SSGX's performance when this crossover was compared with other operators in Sect. 6.

Table 12 shows a more apparent impact on the training error of SSGX when ϵ was modified. Values above 0.5 were ruled out in preliminary experiments. Comparing between 0.1, 0.2 and 0.3, it could be seen that SSGX performed quite consistently, with some decrease in performance at 0.4 and 0.5. Overall, the results in this section show that the training error of SSGX was not very sensitive to *MaxTrial* (values from 10 to 30 were all good) while the proportion of the geometric operator in SSGX should be no > 0.3 for good performance of SSGX on the tested problems. This analysis provides some guidelines for GP practitioners to select appropriate values for SSGX's parameters. We leave the self-adaptation of this parameter for a future study.

8.2 Impact of subtree size constraint with SSGX

This subsection investigates the impact of the size constraint for selected subtrees on SSGX performance. We tested three schemes with the different constraints on size as follows:

- SSGX-S1: There was no size constraint on selected subtrees.
- SSGX-S2: Using lower bound ($\alpha = 1$) to avoid selecting leaf nodes.
- SSGX-S3: Used both lower and upper bounds, with $\alpha = 1$ and $\beta = 80$

The result of SSGX with different constraint in the size of selected subtrees is presented in Table 13. In this table, three metrics including mean of the best fitness on training data, median of testing error and size of solutions were used to analyze the impact of size constraint of selected subtrees on SSGX. The results for SC are also shown in this table for reference. It can be seen from this table that three different schemes for controlling the size of selected subtrees in SSGX had only a slight impact on SSGX performance. Apparently, the mean best fitness on the training data and the median of testing data of SSGX-S1, SSGX-S2 and SSGX-S3 are mostly equal. However, controlling the size of the subtrees in SSGX, as with SSGX-3, helps this scheme find solutions that are considerably smaller than with other schemes.

Table 13 Performance of SSGX with different subtree size control

Metrics	XOvers	K-4	K-6	K-11	K-12	K-14	K-15	UCI-1	UCI-2	UCI-3	UCI-4
Mean	SC	4.86	2.21	7.13	10.3	6.88	2.99	200	2085	109	175
	SSGX-S1	3.34	1.73	5.54	4.33	3.33	1.65	168	890	107	173
	SSGX-S2	4.01	1.69	5.47	4.68	3.08	1.99	163	931	107	173
	SSGX-S3	4.03	1.61	5.89	4.72	3.32	2.02	173	896	107	174
Median	SC	19.0	5.19	18.3	11.0	24.5	7.84	442	2085	220	231
	SSGX-S1	3.80	1.46	8.21	5.61	5.45	2.31	497	1007	203	202
	SSGX-S2	3.87	1.63	7.45	5.79	5.63	3.13	494	1152	204	202
	SSGX-S3	3.69	1.42	7.67	6.27	8.59	2.40	495	1072	203	205
Size	SC	150	117	101	103	87.6	90.1	261	190	147	164
	SSGX-S1	210	127	117	112	108	107	145	120	127	155
	SSGX-S2	185	125	125	116	114	126	173	133	132	135
	SSGX-S3	110	99.1	96.8	102	93.6	95.6	104	96.5	102	110

9 Enhancing the performance of SSGX

In the design of SSGX (Sect. 4), only a certain proportion of SSGX operations execute the geometric recombination. The remainder revert to SC. This suggests an opportunity to use other advanced crossovers in place of SC. In this section, two recently-proposed semantic-based crossovers, SSC [34] and the MSSC [33] are used as the replacements for SC in the implementation of SSGX. Examining the performance of SSGX when SC component is replaced by other advanced operators such as LGX, AGX and RDO are left for the future search.

The configuration of SSC and MSSC in this experiment is as follows. The lower bound semantic sensitivity for both SSC and MSSC was set at 0. The upper bound for SSC was 0.6. The number of trials for selecting a pair of subtrees in both SSC and MSSC was 10. These values are common for good performance of these operators [33]. The performance of SSGX when combined with SSC (SSGX-SSC) and MSSC (SSGX-MSSC) is compared to SC, SSC, MSSC and SSGX. The mean of training error is presented in Table 14 and the median of testing error is shown in Table 15.

Table 14 shows that the two new implementations of SSGX (SSGX-SSC and SSGX-MSSC) helped to further improve its training error. The mean of best fitness of SSGX-SSC and SSGX-MSSC are often smaller than those of other methods. Comparing between SSGX-SSC and SSGX-MSSC, the table shows that SSGX-MSSC was often better than SSGX-SSC. Moreover, SSGX-MSSC was often the best operator in terms of the training error among all tested operators on the tested problems. The results on the testing data was consistent with the results on the training data, showing the better performance of SSGX when combined with SSC and MSSC. Table 15 shows that the median of the testing error of SSGX-SSC and SSGX-MSSC was often smaller than that of SSGX and of other operators.

Table 14 Mean best fitness on training data of SSGX-SSC and SSGX-MSSC

XOver	K-4	K-6	K-11	K-12	K-14	K-15	UCI-1	UCI-2	UCI-3	UCI-4
SC	4.86	2.21	7.13	10.3	6.88	2.99	200	2085	109	175
SSC	4.72	2.17	7.70	10.1	5.60	2.52	195	2015	106	175
MSSC	4.39	1.82	6.61	9.06	5.11	2.65	189	1851	105	172
SSGX	4.03	1.61	5.89	4.72	3.32	2.02	173	896	107	174
SSGX-SSC	4.00	1.56	**3.84**	4.32	**2.21**	1.82	**153**	**806**	103	168
SSGX-MSSC	**1.91**	1.52	**2.62**	**2.65**	**1.71**	**1.01**	**151**	**772**	101	**165**

Significant differences of SSGX-SSC and SSGX-MSSC versus SSGX are printed in bold face

Table 15 Median of testing error of SSGX-SSC and SSGX-MSSC

XOver	K-4	K-6	K-11	K-12	K-14	K-15	UCI-1	UCI-2	UCI-3	UCI-4
SC	19.0	5.19	18.3	11.0	24.5	7.84	442	2085	220	231
SSC	14.8	2.44	14.2	10.8	13.5	4.96	398	1289	218	215
MSSC	12.9	2.75	13.2	9.81	11.2	5.55	389	1134	219	218
SSGX	3.69	1.42	7.67	6.27	8.59	2.40	495	1072	203	205
SSGX-SSC	4.31	1.31	**4.52**	7.40	7.46	3.14	535	1162	195	203
SSGX-MSSC	**2.07**	1.35	**4.69**	**3.52**	**4.72**	2.27	492	1011	**189**	203

Significant differences of SSGX-SSC and SSGX-MSSC versus SSGX are printed in bold face

We conducted statistical tests between SSGX-SSC and SSGX-MSSC versus SSGX using Wilcoxon-signed rank test and with a Bonferroni correction factor of 20 (2 pairwise crossovers and 10 problems). The significant difference (p value $< 0.05/20$) between SSGX-SSC and SSGX-MSSC versus SSGX is presented bold face in Tables 14 and 15. On the training data, SSGX-SSC is better than SSGX on four problems while SSGX-MSSC is better than SSGX on all but K-6 and UCI-3. On the test data, SSGX-SSC was significantly better than SSGX on one problem (K-11) and SSGX-MSSC was significantly better than SSGX on four problems (K-4, K-12, K-14, UCI-3).

Generally, the results in this section show that SSGX's performance can potentially be further enhanced by using advanced crossover operators in place of the SC proportion of SSGX operations. In this section, two semantic-based operators are implemented in that proportion. The new versions lead to improved performance of SSGX in both situations. In the future, other advanced crossovers such as LGX, AGX and RDO will be tested and the performance of SSGX will be examined with these versions.

10 Conclusions and future work

In this paper, a new semantic crossover (SSGX) for GP was proposed. SSGX is inspired by semantic geometric crossover (SGX) [29] but implemented at the subtree level. The objective of SSGX is to address the code growth problem in SGX.

The proposed crossover was tested on a number of regression problems including six GP benchmark problems and four UCI regression problems. The experimental results were compared with other crossovers including SC, semantic geometric crossover (SGX), AGX by Krawiec et al. [23] and RDO by Pawlak et al. [36]. The results showed the advantages of the new crossover in improving GP performance on unseen data compared to other operators. Moreover, the main objective of SSGX in reducing code growth of SGX is also achieved.

Further experiments were conducted to analyse some important aspects of the tested operators. The analysis showed that the computational time of SSGX is not as high as some library-based operators such as AGX and RDO. Moreover, this analysis showed that SGX carries out exploitative, local search, whereas SC and others mostly execute in a more global search manner (focusing on exploration). SSGX is the only operator that possesses both abilities (exploration and exploitation). This may be one reason for its superior performance.

After that, the sensitivities of SSGX's parameters were examined. The results showed that while the number of trials for selecting a subtree that has semantics similar to its parents does not considerably affect SSGX's performance, the proportion of SSGX crossovers in which a geometric operator is used should not be too large (not > 30 %). The impact of the constraint on size of the selected subtree was also investigated and the results showed that this technique helps to considerably reduce the code growth in SGX. Finally, a technique was introduced to further enhance SSGX's performance by using as the non-geometric operator in SSGX two advanced crossovers, SSC and the MSSC. The results showed that these two schemes lead to improved performance of SSGX.

There are a number of research areas for future work which arise from this paper. First, we want to propose some techniques to allow the values of SSGX's parameters (*MaxTrial*, ϵ, α, and β) to be self-adapted during the evolutionary process. In this paper, these values have been tuned and analysed. However, allowing them to be self-adapted could further improve the performance of SSGX. Second, we would like to combine SSGX with some other advanced crossovers. In Sect. 9, SSC and MSSC have been used to enhance SSGX's performance. However, other crossovers could also be used in the proportion of SSGX that is currently implemented using SC.

Next, it will be very interesting to extend the applications of SSGX to other problem domains such as Boolean and classification problems. To date, semantic-based operators are often designed for regression problems only. Thus, it will be very interesting to investigate the performance of semantic-based crossovers including SSGX on classification problems. Finally, at the theoretical level, we want to conduct a deeper analysis of the role of standard subtree crossover and geometric crossover in SSGX during the evolutionary process to gain a better understanding of the behavior of this operator.

Acknowledgments This research is funded by Vietnam National Foundation for Science and Technology Development (NAFOSTED) under Grant Number 102.01-2014.09.

Appendix 1: List of operators tested

See Table 16.

Table 16 The operators tested in the paper

Abbreviation	Meaning
SC	Standard crossover
SSC	Semantic similary-based crossover
MSSC	The most semantic similarity-based crossover
SGX	Semantic geometric crossover
AGX	Approximate geometric crossover
RDO	Random desired operator
SSGX	Subtree semantic geometric crossover
SSGX-SSC	SSGX combined with SSC
SSGX-MSSC	SSGX combined with MSSC

Appendix 2: Sample output

As stated in Table 7, the trees output using GP with SSGX have a mean size close to 100 nodes. A typical output achieved on the UCI-I problem is shown:

```
add(mul(div(1,add(1,ep(sub(0,X2)))),add(mul(div(1,add(1,ep(sub(X3,X4)))),
    add(sub(ep(ep(sub(sin(X5),X3))),div(1,add(sin(sin(X7)),X3))),
    log(add(sub(mul(mul(X7,X1),X7),log(1)),div(0,1))))),mul(1,mul(div(1,
    add(1,ep(ep(sub(X2,X3))))),add(div(1,1),log(X7)),div(sin(1),
    add(sin(sin(X7)),X3))),X7))),mul(sub(1,div(1,add(1,ep(sub(div(1,
    add(sin(ep(X4)),ep(ep(sub(X3,add(sin(sin(X7)),X3)))))),X2))))),
    log(mul(X7,mul(X3,X3)))))
```

This tree shows several typical features. The root of the tree is `add(mul(.),mul(.))`, which indicates the geometric crossover template $T_R St_1 + (1 - T_R) St_2$. This pattern occurs at the root of many solution trees. However, here it occurs just once in the tree, whereas with SGX this pattern is ubiquitous. The pattern `div(1,add(1,ep(sub(0,X2))))` also occurs near the root, indicating the logistic mapping $1/(1 + e^{-X_2})$ applied to a random T_R (here $T_R = X_2$). Again, this pattern occurs in many of the solution trees. In some cases the pattern occurs partially, since after creation through crossover it can be altered by later crossover or mutation. However, the pattern is again not ubiquitous in the tree: in this case, it occurs fully once, and partially 4 times.

Appendix 3: Figures

This appendix presents figures for the results in Sects. 6 and 7. Figure 1 shows the mean best fitness of the five crossovers over the course of the evolutionary process on K-6, K-11, K-14 and UCI-1. Overall, all operators but SGX performed better

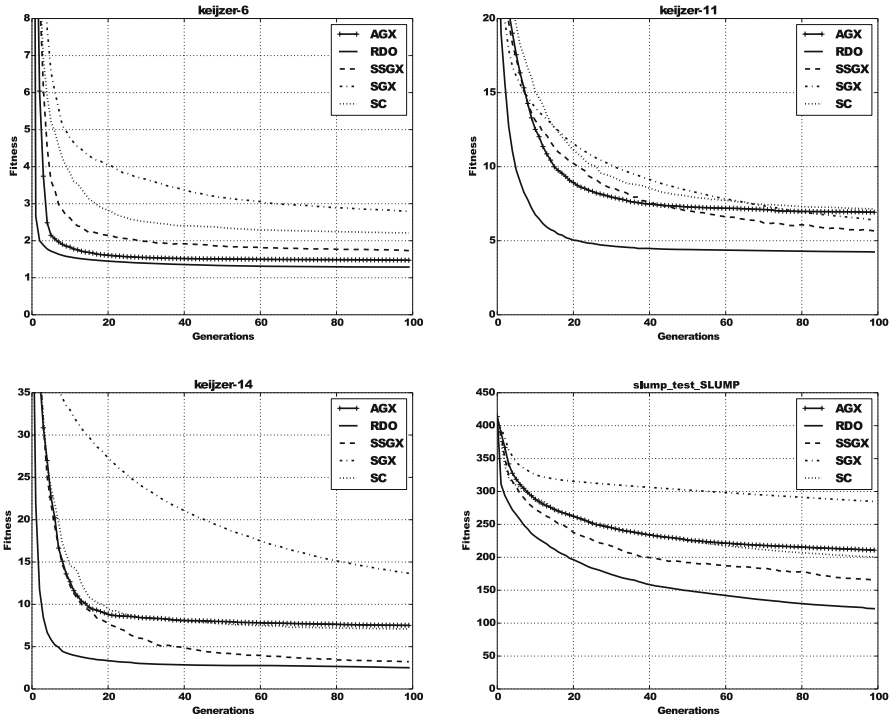

Fig. 1 Mean best fitness of five operators over the generations

than SC during the evolutionary process. However, RDO and AGX tended to converge early. These operators quickly improved the error on the training data after around twenty generations but made little progress after that point. Conversely, SSGX kept improving the fitness until the end of the evolution (generation 100).

Figure 2 presents the average of the size of individuals over the generations of the four operators (we exclude SGX here since the size of individuals of SGX is too high to be shown) on the same four problems. It can be seen that RDO is the crossover that usually grew fastest. SSGX grew fast at the beginning of the evolutionary process (about 20 generations). However, after that point, the operator did not grow as fast as others. Among the four operators, AGX is the operator that grew least.

The semantic distance between parents and children with the five tested operators is presented in Fig. 3. SGX is the operator that created the smallest changes in semantics during the evolutionary process. The semantic distance between children and their parents in SGX quickly reduced to nearly zero after about five generations. The semantic step with AGX and RDO was also much less than SC and SSGX, but this value was not as small as with SGX. Two other operators (SC and SSGX) made a greater move in the semantic space, and SSGX is the operator that make the largest change. However, this value was averaged over both standard and geometric portions of the crossover. In SSGX, the semantic change in the geometric portion

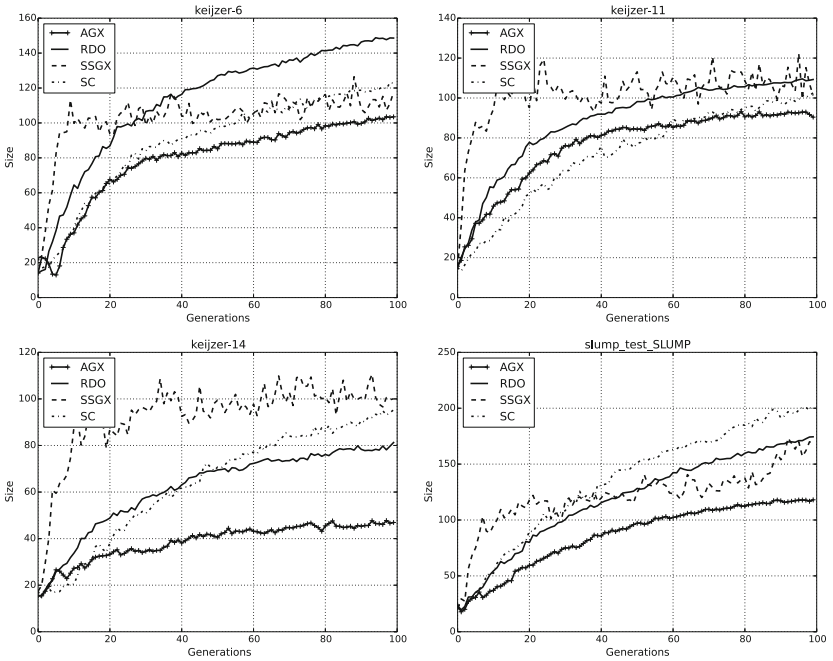

Fig. 2 Average of population size over the generations

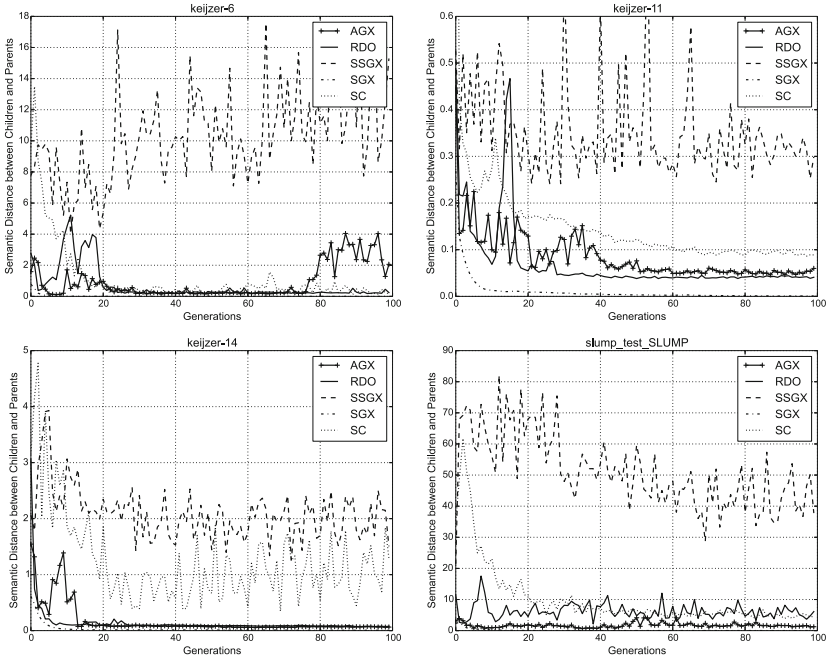

Fig. 3 Average of semantic distance between offspring and parents over the generations

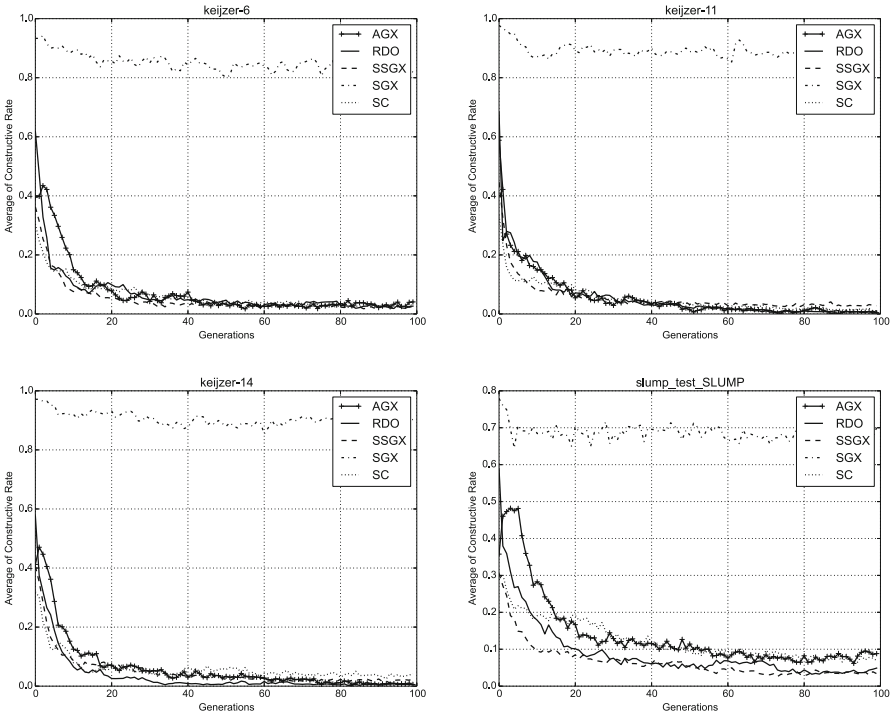

Fig. 4 Average of constructive rate over the generations

was often much smaller than that with SC (see Table 9). Consequently, SSGX can exploit the search space better than SC.

Figure 4 shows the constructive rate of five operators on the same four problems. It can be seen that the constructive rate of SGX was far higher than other crossovers. However, since its search step was much smaller, the performance of SGX was not always better than SC as shown in Sect. 6. Comparing the four subtree crossovers, Fig. 4 shows that their constructive rate is mostly equal.

References

1. L. Altenberg, The evolution of evolvability in genetic programming, in *Advances in Genetic Programming, chapter 3*, ed. by K.E. Kinnear Jr (MIT Press, Cambridge, 1994), pp. 47–74
2. K. Bache, M. Lichman, UCI machine learning repository (2013). http://archive.ics.uci.edu/ml
3. L. Beadle, C.G. Johnson, Semantically driven crossover in genetic programming. In *Proceedings of the IEEE World Congress on Computational Intelligence* (IEEE Press, 2008), pp. 111–116
4. L. Beadle, C.G. Johnson, Semantic analysis of program initialisation in genetic programming. Genet. Program. Evolvable Mach. **10**(3), 307–337 (2009)
5. L. Beadle, C.G. Johnson, Semantically driven mutation in genetic programming. In ed. by A. Tyrrell. *2009 IEEE Congress on Evolutionary Computation, Trondheim, Norway, 18-21 May 2009. IEEE Computational Intelligence Society*, (IEEE Press), pp. 1336–1342
6. A. Boukerche, *Algorithms and Protocols for Wireless Sensor Networks* (Wiley-IEEE Press, Cambridge, 2008)

7. M. Castelli, D. Castaldi, I. Giordani, S. Silva, L. Vanneschi, F. Archetti, D. Maccagnola, An efficient implementation of geometric semantic genetic programming for anticoagulation level prediction in pharmacogenetics. In *Proceedings of the 16th Portuguese Conference on Artificial Intelligence, EPIA 2013*. Lecture Notes in Computer Science, vol. 8154 (Springer, Sept. 9-12, 2013), pp. 78–89
8. R. Cleary, M. O'Neill, An attribute grammar decoder for the 01 multi-constrained knapsack problem. In *Proceedings of the Evolutionary Computation in Combinatorial Optimization*, (Springer Verlag, 2005), pp. 34–45
9. M. de la Cruz Echeanda, A. O. de la Puente, M. Alfonseca, Attribute grammar evolution. In *Proceedings of the IWINAC 2005*, (Springer Verlag, Berlin Heidelberg, 2005), pp. 182–191
10. E. Glaab, J. Bacardit, J.M. Garibaldi, N. Krasnogor, Using rule-based machine learning for candidate disease gene prioritization and sample classification of cancer gene expression data. PLoS One **7**, 1–18 (2012)
11. P.D. Grunwald, *The Minimum Description Length Principle* (MIT Press, 2007)
12. P. He, L. Kang, C.G. Johnson, S. Ying, Hoare logic-based genetic programming. Sci. China Inform. Sci. **54**(3), 623–637 (2011)
13. C.G. Johnson, Deriving genetic programming fitness properties by static analysis. In *Proceedings of the 4th European Conference on Genetic Programming (EuroGP2002)*, (Springer, 2002), pp. 299–308
14. C.G. Johnson, *What can automatic programming learn from theoretical computer science*. In *Proceedings of the UK Workshop on Computational Intelligence*, (University of Birmingham, 2002)
15. C.G. Johnson, Genetic programming with fitness based on model checking. In *Proceedings of the 10th European Conference on Genetic Programming (EuroGP2002)*, (Springer, 2007), pp. 114–124
16. G. Katz, D. Peled, Genetic programming and model checking: Synthesizing new mutual exclusion algorithms. In *Automated Technology for Verification and Analysis*. Lecture Notes in Computer Science, vol. 5311 (Springer, 2008), pp. 33–47
17. M. Keijzer, Improving symbolic regression with interval arithmetic and linear scaling. In *Proceedings of EuroGP'2003* (Springer-Verlag, 2003), pp. 70–82
18. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (The MIT Press, Cambridge, 1992)
19. K. Krawiec, Medial crossovers for genetic programming. In *Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012*. LNCS, vol. 7244 (Springer Verlag, Malaga, Spain, 11–13 Apr. 2012), pp. 61–72
20. K. Krawiec, P. Lichocki, Approximating geometric crossover in semantic space. In ed. by F. Rothlauf, *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, (ACM, 2009), pp. 987–994
21. K. Krawiec, T. Pawlak, Quantitative analysis of locally geometric semantic crossover. In *Parallel Problem Solving from Nature - PPSN XII*. Lecture Notes in Computer Science, vol. 7491 (Springer, Taormina, Italy, Sept. 1–5 2012), pp. 397–406
22. K. Krawiec, T. Pawlak, Approximating geometric crossover by semantic backpropagation. In *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference (GECCO 2013)*, (ACM, Amsterdam, The Netherlands, 6-10 July 2013), pp. 941–948
23. K. Krawiec, T. Pawlak, Locally geometric semantic crossover: a study on the roles of semantics and homology in recombination operators. Genet. Program. Evolvable Mach. **14**(1), 31–63 (2013)
24. A. Mambrini, L. Manzoni, A comparison between geometric semantic GP and cartesian GP for boolean functions learning. In *Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion (GECCO 2014)*, (ACM, Vancouver, BC, Canada, 12-16 July 2014), pp. 143–144
25. 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, Genetic programming needs better benchmarks. In *Proceedings of GECCO 2012*, (ACM, Philadelphia, 2012)
26. N. McPhee, B. Ohs, T. Hutchison, Semantic building blocks in genetic programming. In *Proceedings of 11th European Conference on Genetic Programming* (Springer, 2008), pp. 134–145
27. A. Moraglio, Geometric unification of evolutionary algorithms. In *European Graduate Student Workshop on Evolutionary Computation* (Budapest, Hungary, 10 Apr. 2006), pp. 45–58
28. A. Moraglio, An efficient implementation of GSGP using higher-order functions and memoization. In ed. by C. Johnson, K. Krawiec, A. Moraglio, M. O'Neill, *Semantic Methods in Genetic Programming*. Workshop at Parallel Problem Solving from Nature 2014 conference (Ljubljana, Slovenia, 13 Sept. 2014)

29. A. Moraglio, K. Krawiec, C.G. Johnson, Geometric semantic genetic programming. In *12th International Conference on Parallel Problem Solving from Nature (PPSN)*. Lecture Notes in Computer Science, vol. 7491 (Springer, Taormina, Italy, 2012), pp. 21–31
30. A. Moraglio, A. Mambrini, Runtime analysis of mutation-based geometric semantic genetic programming for basis functions regression. In *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference (GECCO 2013)*, (ACM, 2013), pp. 989–996
31. A. Moraglio, J. Togelius, S. Silva, Geometric differential evolution for combinatorial and programs spaces. Evolut. Comput. **21**(4), 591–624 (2013)
32. Q.U. Nguyen, X.H. Nguyen, M. O'Neill, Semantic aware crossover for genetic programming: the case for real-valued function regression. In *Proceedings of the 12th European Conference on Genetic Programming (EuroGP 2009)*, (Springer, April 2009), pp. 292–302
33. Q.U. Nguyen, X.H. Nguyen, M. O'Neill, R.I. McKay, N.P. Dao, On the roles of semantic locality of crossover in genetic programming. Inform. Sci. **235**, 195–213 (2013)
34. Q.U. Nguyen, X.H. Nguyen, 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)
35. T.P. Pawlak, B. Wieloch, K. Krawiec, Review and comparative analysis of geometric semantic crossovers. Genet. Program. Evolvable Mach. **16**(3), 351–386 (2015)
36. T.P. Pawlak, B. Wieloch, K. Krawiec, Semantic backpropagation for designing search operators in genetic programming. IEEE Trans. Evolut. Comput. **19**(2), 326–340 (2015)
37. R. Poli, W. B. Langdon, N. F. McPhee. *A Field Guide to Genetic Programming*. http://lulu.com, and http://www.gp-field-guide.org.uk, (2008)
38. L. Vanneschi, M. Castelli, L. Manzoni, S. Silva, A new implementation of geometric semantic GP and its application to problems in pharmacokinetics. In *Proceedings of the 16th European Conference on Genetic Programming, EuroGP 2013*. LNCS, vol. 7831 (Springer Verlag, Vienna, Austria, 3-5 Apr. 2013), pp. 205–216
39. L. Vanneschi, M. Castelli, S. Silva, A survey of semantic methods in genetic programming. Genet. Program. Evolvable Mach. **15**(2), 195–214 (2014)
40. D.R. White, J. McDermott, M. Castelli, L. Manzoni, B.W. Goldman, G. Kronberger, W. Jaskowski, U.-M. O'Reilly, S. Luke, Better GP benchmarks: community survey results and proposals. Genet. Program. Evolvable Mach. **14**(1), 3–29 (2013)
41. M.L. Wong, K.S. Leung, An induction system that learns programs in different programming languages using genetic programming and logic grammars. In *Proceedings of the 7th IEEE International Conference on Tools with Artificial Intelligence*, (1995)