Semantic Backpropagation for Designing Search Operators in Genetic Programming

Tomasz P. Pawlak, Bartosz Wieloch, Krzysztof Krawiec, *Member, IEEE*

***Abstract*—In genetic programming, a search algorithm is expected to produce a program that achieves the desired final computation state (desired output). To reach that state, an executing program needs to traverse certain intermediate computation states. An evolutionary search process is expected to autonomously discover such states. This can be difficult for nontrivial tasks that require long programs to be solved. The semantic backpropagation algorithm proposed in this paper heuristically inverts the execution of evolving programs to determine the desired intermediate computation states. Two search operators, Random Desired Operator and Approximately Geometric Semantic Crossover, use the intermediate states determined by semantic backpropagation to define subtasks of the original programming task, which are then solved using an exhaustive search. The operators outperform the standard genetic search operators and other semantic-aware operators when compared on a suite of symbolic regression and Boolean benchmarks. This result and additional analysis conducted in this study indicate that semantic backpropagation helps evolution at identifying the desired intermediate computation states, and makes the search process more efficient.**

***Index Terms*—program synthesis, semantics, reversible computing, problem decomposition, mutation, geometric crossover**

I. Introduction

THE objective in an automated programming task is to synthesize a program that exhibits certain desired behavior. The desired behavior is typically defined by specifying the desired program output as a function of program input. This can be done either explicitly, by enumerating all relevant input-output pairs (or a sample thereof), or implicitly, by defining an objective that program responses have to optimize.

Apart from the trivial cases, the desired input-output behavior cannot be attained by applying a single instruction to the input data. A combination of instructions is necessary: a sequence, tree, or graph, depending on the programming paradigm. The combinations of instructions (syntax) determine program behavior (semantics) in a complex manner. This characteristic, known as ruggedness of the fitness landscape [1] or low causality [2], makes it hard to design automatic programming algorithms that scale well with task complexity.

A convenient approach to solving programming tasks is to pose them as search problems: a search algorithm runs programs, observes how they behave, and uses that information to direct the search process. In genetic programming (GP) this process involves the bio-inspired mechanisms of variation and selection, and a fitness function that quantifies the degree to which the actual program output matches the desired output (the *target* in the following).

Conventionally in GP, fitness depends only on the ultimate effect of program execution; the intermediate effects, like the values calculated by the subtrees of a program tree, or the transient states of registers in linear GP, are ignored. This selectivity is commonly accepted: in the end, it is only the final outcome that determines whether the programming task has been solved or not (or how well a program approximates the desired output). This remains however in stark contrast to the process of programming as exercised by humans. When faced with a nontrivial task, programmers split it into subtasks and attempt to solve them independently or semi-independently. This strategy often proves effective, because human programmers often know in advance which *intermediate execution states* are desired and which are not. For instance, for a programming task 'design an algorithm that calculates the median of an array of numbers', such a desired intermediate is the input array sorted in ascending or descending order.

The ability to decompose tasks allows human programmers to excel on tasks that automatic programming methods still struggle to solve. It is then highly desirable to equip the GP algorithms with an analogous capability, which we postulated in our previous studies [3], [4], [5].

In this paper, we propose a method that determines which intermediate computation states are desirable when solving a given programming task. The method exploits the fact that program execution can be to some extent reversed: for any computation state (e.g., the state of working memory or other executing environment), the number of other states it can be achieved from by executing a single instruction is limited (although potentially large). It is therefore possible to revert (albeit in some cases ambiguously) the effects of execution of an instruction sequence for a given *output*. This process of *semantic backpropagation* allows us to determine a desired intermediate memory state (*subtarget*), which in turn defines a specific programming subtask. The space of programs that need to be considered to solve a subtask is a subspace of the original search space, and can be thus searched more efficiently.

The main contribution of this paper is the demonstration how this idea (presented in Sections III and IV) can be utilized in mutation and crossover operators (Sec. V). Compared to [6], [7] where we originally proposed these operators, here we present a generalized version of semantic backpropa-

The authors are with the Institute of Computing Science, Poznan University of Technology, Poznań, Poland, e-mails: tpawlak@cs.put.poznan.pl, bwieloch@cs.put.poznan.pl, krawiec@cs.put.poznan.pl.

gation, embed the search operators into a common formal framework, and examine them in detail on a wide range of benchmarks (Sec. VI).

II. Related work

This paper focuses on program behavior, reflecting growing interest in *semantic GP*. Rather than focusing on programs, semantic GP focuses on investigating the *effects* of program execution, and on using knowledge about such effects to designing more effective GP search algorithms. Besides the fundamental analytical studies conducted by Langdon [8], McPhee *et al.* were to our knowledge the first to study the impact of crossover on program semantics and semantic building blocks [9]. They defined 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 for Boolean problems. In another early study, Beadle and Johnson [10] proposed a semantically driven crossover operator for Boolean problems that guarantees the offspring to be semantically distinct from both parents. In [11] Jackson analyzed semantic diversity in the initial population. More recently, Galvan-Lopez *et al.* came up with a tournament selection that discouraged semantic duplicates [12].

Nguyen *et al*. [13] considered two semantic crossover operators for symbolic regression, one that permits crossover only if the subtrees to be exchanged in the parents are semantically more distant than a given lower limit (a parameter of the method), and another with an additional upper limit on the distance. Later, Nguyen *et al.* [14] proposed an operator that from a set of valid pairs of subtrees to be exchanged in the parents, chooses the pair with the smallest but over the lower limit distance, dropping so the upper limit parameter. The representation of semantics adopted there, i.e., a vector of numbers returned by a program for all training examples (fitness cases), is currently the most widely used one in semantic GP and employed also in this paper. Driven by similar intentions, though without explicitly referring to program semantics, Day and Nandi used binary strings to characterize how individuals in a population cope with particular fitness cases, and designed a mating strategy that exploits that information [15].

The recent works by Moraglio *et al.* [16], [17] follows a qualitatively different approach. The key observation that underpins their approach is that the fitness function in GP is typically defined as the distance between the program's semantics and a predefined target semantics. This turns the set of semantics into a metric space that has certain geometric properties which can be exploited to make the search process more efficient. The authors proposed two search operators that rely on this principle, including an exact geometric crossover that is guaranteed to produce offspring which are semantically intermediate with respect to their parents (i.e., combines their behaviors). This is achieved by merging the parent programs into an offspring using an additional syntactic structure. In an analogous endeavor, Krawiec and Pawlak [18] proposed a crossover operator designed to be approximately geometric. Some of these geometric operators serve as control methods in Section VI. In the context of these studies, the methods proposed in this paper remain original in attempting problem decomposition on the semantic level.

As it will become clear in Section IV, semantic backpropagation uses concepts similar to those of reversible computing [19], a paradigm which assumes that the effects of computation can be always reversed. In that framework, every instruction must implement a transition function that transforms the current state of the execution environment (e.g., registers, memory) to the other state in a one-to-one way, i.e. injectively. No wonder it became an area of interest for many researchers in quantum computing, and was also studied in genetic programming. Langdon in [20] analyzed the fitness distribution of reversible programs and found it to be normal. In [21], Multi Expression Programming [22] and Fredkin gate have been employed to solve even-parity problems.

Unfortunately, reversible computing is mainly a theoretical domain, because it assumes a strong, injective notion of reversibility, while the conventional programming languages abound in instructions that are irreversible. For this reason we proposed here a more relaxed notion of program *inversion*, which allows the instructions to implement many-to-one mappings (surjections).

III. Background

A. Program semantics

As mentioned in the Introduction, when solving a programming task we are primarily interested in program behavior (what programs 'do'). Prior to formally defining a programming task, one must thus specify first what program behavior is. To this aim, we introduce the concept of program semantics.

Let $p \in P$ be a program, i.e., a sequence (or other structure) of symbols from a given programming language P. When applied to an input $in \in I$, a program p produces[1] certain output $p(in)$. In this way, a program realizes certain mapping from the set of inputs I into the set of outputs O, which we denote as $p : I \to O$.

Definition 1. *Semantic mapping* is a function $s : P \to S$ mapping any program from P to the *semantic space* S, which has the following property:

$$s(p_1) = s(p_2) \iff \forall in \in I : p_1(in) = p_2(in)$$

Let us summarize the properties of semantics that result from this definition. Firstly, every program has exactly one semantics. Secondly, two or more programs can have the same semantics. Thirdly, programs that behave differently (i.e., produce different outputs for one or more inputs) have different semantics.

The semantic space S enumerates all possible behaviors of programs for all considered inputs. The mapping s implicitly partitions the programs into semantic equivalence classes, so that every element in S corresponds one-to-one to a unique combination of outputs generated by programs. In this sense, semantics is *complete* in capturing the entire information on

[1] We consider only programs that halt for all inputs $in \in I$.

program behavior: when $s(p)$ is known, nothing more is needed to determine p's behavior.

Definition 1 does not specify how semantics is represented. It can be any formal object that meets the conditions specified there, including the denotational semantics or operational semantics used in the theories of formal languages. Note that, the code of a program (i.e., a sequence of symbols) cannot be considered as its semantics, unless the semantic mapping is bijective (which it never is in practice).

In this paper, we adopt the convention that is common in GP and assume that a programming task is specified by a finite training sample of *fitness cases*, each being a pair consisting of program input and the corresponding desired program output. A fitness case is then a pair from $I \times O$. We also assume that I contains *only* the inputs present in the given set of fitness cases, conforming to the learning-from-examples paradigm of machine learning, where the learner has no access to other (testing) examples. Given these assumptions, we can define the semantics more concretely:

Definition 2. Semantics $s(p)$ of a program p is the vector[2] of values from O obtained by running p on all inputs from I:

$$s(p) = [p(in)]_{in \in I} = [p(in_1), \ldots, p(in_l)] \tag{1}$$

where $l = |I|$ is the number of fitness cases.

This representation of semantics, prevailing in the earlier studies on semantics in GP [23], [16], [13], [18], [7] conforms to Def. 1. In particular, it is complete in the above sense: because $s(p)$ explicitly enumerates program behavior for all inputs, it allows us to trivially determine program output for any element from $in \in I$ by simply selecting the appropriate element from the vector.

B. Programming tasks

The properties of semantic space, in particular the relations between its elements, are essential for the approach proposed in this paper. The key observation that propels the recent developments in semantic GP is that the semantic space is inherently endowed with a structure. This structure is imposed on S by the programming task itself, more specifically by the fitness function that gauges the discrepancy between the actual program behavior and the desired one.

Definition 3. Given a set of programs P (defined implicitly by programming language), a *programming task* (*task* for short) consists in finding a program $p \in P$ that minimizes

$$f(p) = d(s(p), t) \tag{2}$$

where $t \in S$ is the *target semantics* (*target* for short) that describes the desired behavior of a program, and $d : S \times S \to \mathbb{R}$ is a metric, called *semantic distance*. For a program p such that $f(p) = 0$ we say that it *solves* the task t.

From now on, we use the terms 'target' and 'programming task' interchangeably, because, other things being equal, a target uniquely identifies the programming task.

[2]We use the terms 'vector' and 'tuple' exchangeably, depending on the context. In both cases, the i[th] element of vector/tuple corresponds to i[th] test (fitness case).

As an illustration, let us consider the class of univariate symbolic regression tasks, where programs are real-valued functions of a single real independent variable, i.e., $I \subset \mathbb{R}$ and $O \subset \mathbb{R}$. The task is specified by $l = |I|$ fitness cases (in_i, t_i) from $I \times O$, where $in_i \in I$ is program input and t_i defines the corresponding desired output[3]. The semantics $s(p)$ of a program p is a vector of l real numbers obtained by applying p to the elements of I. The minimized fitness of a program is defined by, e.g., the Euclidean or Manhattan distance d between program's semantics and the target t.

Most of programming tasks considered in GP are defined similarly in terms of fitness cases and semantic distance. Exceptions are the tasks for which the target is not explicitly specified, like evolving controllers (e.g., pole balancing, artificial ant, robotics). For such tasks, fitness assessment is implicit, e.g., based on results of simulations, and semantic methods that refer to target cannot be directly applied. Of two operators introduced in this paper, one (RDO, Section V-B) requires the knowledge of target, while the other (AGX, Section V-C) does not, so the approach presented here can be applied to a wide range of GP tasks.

IV. Inversion of program execution

A. The rationale for inversion

Programming tasks (Def. 3) are in general difficult to solve, because the mapping from the combinatorial program space to the semantic space is usually very complex. However, when semantics is a vector of outputs produced by a program for particular fitness cases (Eq. 1), the semantic mapping $s : P \to S$ ceases to be a black-box monolith. In such a case, each component of the vector that forms the semantics is a result of a sequential process, where particular program instructions process the outcomes of their predecessors, and execution of the last instruction completes the calculation of semantics (for a single fitness case). The decomposability of this process is the key to the methods presented in this paper.

We assume that any program p can be decomposed into its *prefix* $p_{(1)}$ and *suffix* $p_{(2)}$, which in terms of Reverse Polish Notation that we adopt here will be written as

$$p = [p_{(1)}\, p_{(2)}]$$

Execution of such a compound program for an input datum in involves applying the suffix $p_{(2)}$ to the outcome produced by the prefix $p_{(1)}$, i.e., $p(in) = p_{(2)}(p_{(1)}(in))$. In this formulation, we abstract from program representation. For sequences of instructions, prefixes and suffixes take the form of sequences as well. For tree programs without side effects (which we limit our attention to in this paper), a prefix is any subtree of a program, while a suffix is a program tree with a single subtree removed (termed *context* in [9]). Regardless of representation, we assume that prefix is a well-formed (sub)program that can be executed. Therefore, every prefix will also have a specific semantics as described in Def. 1.

[3]The set I contains only the inputs that occur in the training set. This however does not prevent the evolved programs to be applicable to other inputs (from $\mathbb{R} \setminus I$) to, e.g., assess program's generalization capability.

If the considered task t is solvable, there exists a program p^* such that $s(p^*) = t$. Let $p^*_{(1)}$ be a prefix of p^* and $p^*_{(2)}$ the corresponding suffix. The key observation is that $p^*_{(1)}$ determines the semantics $s(p^*_{(1)})$ that can be treated as a target for another programming task. We say that $p^*_{(1)}$ determines a *subtarget* $t' = s(p^*_{(1)})$. Any program p_s that solves the *subtask* defined by subtarget t' can be used as a substitute for $p^*_{(1)}$ in p^*. Formally, if $s(p_s) = t'$, then the program $[p_s\, p^*_{(2)}]$ solves task t, i.e., $s([p_s\, p^*_{(2)}]) = t$.

Similarly, the suffix $p^*_{(2)}$ can be said to determine a *set* of subtasks, i.e., a set of such semantics t'' that $s([t''\, p^*_{(2)}]) = t$. The subtask t' determined by the prefix is one of them, but there can be more such subtasks t'' because of the many-to-one operation of program instructions. We formalize this concept as *desired semantics* in Section IV-B.

Because every solution to a subtask forms a prefix of at least one solution to the original task, and any (proper) prefix is shorter than the program it is part of, it is reasonable to hypothesize that a subtask can be easier to solve than the original task. Assuming that the computational cost of solving a programming task is a monotonically increasing function of the length of the shortest solution[4], solving a subtask instead of the original task may bring substantial savings.

However, the subtasks above were derived from a given *optimal* program, i.e., a program that is already known to solve the original task. The practical question is: can we define a subtask of a task without solving the latter one in the first place?

We argue that useful subtask candidates can be derived even from non-optimal programs. Consider the population of an evolutionary process (or any other incremental search algorithm). It is likely that some programs in such a population include subsequences of instructions that occur in the sought optimal program. In particular, some of such 'correct' subsequences may form the suffixes of programs. It is worth noting here that already early studies on semantic GP, particularly by McPhee *et al.* [9], emphasized the importance of identifying correct program suffixes.

Let p be a program with a correct suffix, i.e., a suffix that is a part of one of optimal programs p^*, i.e., $p = [p_{(1)}\, p^*_{(2)}]$. Let us assume that we can invert the execution of $p^*_{(2)}$, i.e., *directly* calculate *any* subtarget[5] t'' such that $s([t''\, p^*_{(2)}]) = t$. Then, t'' would define a subtask that could be easier to solve for the reason given above.

The semantic backpropagation presented in the next sections is an effective heuristic for finding such subtasks.

B. Semantic backpropagation

The semantic backpropagation algorithm finds the subtargets for a given target and a suffix of a program represented as a tree. To attain this goal, it inverts the execution of the suffix for the desired output specified by the target.

[4]Consider a brute force algorithm that attempts to solve a programming task by generating all programs, starting from the shortest ones and gradually increasing their length. Given two programming tasks having the shortest solutions p and $[p\, p']$ respectively, such algorithm will find a solution for the former task earlier. In this sense a subtask can be expected to be easier to solve than the entire task. Obviously, this argument will not always be valid for more sophisticating search algorithms like GP.

[5]There are in general many such subtargets.

Algorithm 1 Semantic backpropagation algorithm for tree programs. Parameters: t: target semantics, p: program tree, n: the node in p to be reached by the backpropagation. $\textsc{Child}(a, n)$ returns the child of node a on the path leading from a to n and $\textsc{Pos}(a, n)$ returns the child's position in the list of arguments of node a. $\textsc{Invert}(a, k, o)$ returns the set of the desired values for kth argument of node a and desired output o, as specified in Table I.

```
1:  function SemanticBackpropagation(t, p, n)
2:      for all t_i ∈ t do                          ▷ For each fitness case
3:          D_i ← {t_i}
4:          a ← p
5:          while a ≠ n ∧ D_i ≠ ∅ ∧ ∗ ∉ D_i do
6:              k ← Pos(a, n)
7:              D' ← ∅
8:              for all o ∈ D_i do
9:                  D' ← D' ∪ Invert(a, k, o)
10:             D_i ← D'
11:             a ← Child(a, n)
12:     return (D_1, D_2, ...)
13: end function
```

Because in general there may be infinitely many subtargets t'' such that $s([t''\, p^*_{(2)}]) = t$, the algorithm does not guarantee to determine *all* of them, but only a subset thereof. To efficiently represent such a subset, we use *desired semantics*.

Definition 4. The *desired semantics* for a programming task t and suffix $p_{(2)}$ is a tuple $D = (D_i)$ of sets D_i corresponding to particular components of t, where each D_i contains the desired values defined by the suffix $p_{(2)}$ for the ith fitness case, i.e.:

$$D_i = \{x : p_{(2)}(x) = t_i\} \tag{3}$$

In other words, D_i contains a set of values x that cause the suffix to reach the target on the ith fitness case. Such values form the desired output for the ith fitness case of subtargets defined by $p_{(2)}$ for t. D_i can contain any set of such outputs that fulfill Eq. 3, not necessarily all of them.

Algorithm 1 presents the SemanticBackpropagation procedure which heuristically calculates a desired semantics determined by a given suffix. Although the inversion process requires only the program suffix, we assume for convenience that the arguments of the procedure are a program tree p and a node n in that tree. The node n unambiguously identifies the suffix (see Fig. 1).

For a given target t, $\textsc{SemanticBackpropagation}(t, p, n)$ computes the desired semantics associated with the suffix determined by p and n. For each fitness case t_i independently, it carries out traversal over the nodes on the path from the root of p to n. It starts by propagating the ith component of the original target, t_i, through the instruction located at the root node. The inversion of the consecutive instructions on the path is carried out by the loop in lines 5–11. For every o in the working set of desired values D_i, the call of the Invert

Table I: Definition of $\text{INVERT}(a, k, o)$ used in our experiments for the symbolic regression and the Boolean domain. For a subtree a (an instruction node with one or two children that returned c_1 and c_2) the formulae determine the desired value for the first ($k = 1$, center column) and second ($k = 2$, right column) argument, given the desired value of the whole subtree a.

Subtree a	$\text{INVERT}(a, 1, o)$	$\text{INVERT}(a, 2, o)$
Symbolic regression domain		
$c_1 + c_2$	$o - c_2$	$o - c_1$
$c_1 - c_2$	$o + c_2$	$c_1 - o$
$c_1 \times c_2$	$o/c_2 \quad c_2 \neq 0$ $* \quad c_2 = 0 \wedge o = 0$ $\emptyset \quad c_2 = 0 \wedge o \neq 0$	$o/c_1 \quad c_1 \neq 0$ $* \quad c_1 = 0 \wedge o = 0$ $\emptyset \quad c_1 = 0 \wedge o \neq 0$
c_1 / c_2	$o \times c_2 \quad c_2 \neq \pm\infty$ $* \quad c_2 = \pm\infty \wedge o = 0$ $\emptyset \quad c_2 = \pm\infty \wedge o \neq 0$	$c_1 / o \quad c_1 \neq 0$ $* \quad c_1 = 0 \wedge o = 0$ $\emptyset \quad c_1 = 0 \wedge o \neq 0$
$\exp(c_1)$	$\log o \quad o \geq 0$ $\emptyset \quad otherwise$	—
$\log \lvert c_1 \rvert$	$\{-e^o, e^o\}$	—
$\sin c_1$	$\{\arcsin o - 2\pi, \arcsin o\} \quad \lvert o \rvert \leq 1$ $\emptyset \quad otherwise$	
$\cos c_1$	$\{\arccos o - 2\pi, \arccos o\} \quad \lvert o \rvert \leq 1$ $\emptyset \quad otherwise$	
Boolean domain		
c_1 and c_2	$o \quad c_2$ $* \quad$ not c_2 and not o $\emptyset \quad$ not c_2 and o	$o \quad c_1$ $* \quad$ not c_1 and not o $\emptyset \quad$ not c_1 and o
c_1 or c_2	$o \quad$ not c_2 $* \quad c_2$ and o $\emptyset \quad c_2$ and not o	$o \quad$ not c_1 $* \quad c_1$ and o $\emptyset \quad c_1$ and not o
c_1 nand c_2	not $o \quad c_2$ $* \quad$ not c_2 and o $\emptyset \quad$ not c_2 and not o	not $o \quad c_1$ $* \quad$ not c_1 and o $\emptyset \quad$ not c_1 and not o
c_1 nor c_2	not $o \quad$ not c_2 $* \quad c_2$ and not o $\emptyset \quad c_2$ and o	not $o \quad$ not c_1 $* \quad c_1$ and not o $\emptyset \quad c_1$ and o

function determines the desired values for the kth child of the current node a, where the child is the next node on the path.

Let us illustrate this procedure with an example, where SEMANTICBACKPROPAGATION is called for target $t = [-2, 0, 0]$, program p shown in Fig. 1, and n being the node marked in blue. The semantics of subtrees in p are shown in black boxes. Let us consider the first fitness case, the desired output for which is -2. The algorithm starts at the root node. Because n is in the right subtree of the root, the algorithm will determine the set D_1 for the right argument of the root instruction ('$-$'). Currently, that node calculates $1-(-1) = 2$. The algorithm attempts to find out what should be the value of the right argument (say, x) to make the outcome meet the target -2, i.e., $1-x = -2$. This can be obtained by inverting that calculation, and so the algorithm ends up with $x = 1-(-2) = 3$. This value becomes the first element of the desired semantics propagated to the right child of the root node (blue dotted box), together with the values calculated independently for the remaining two fitness cases. The resulting desired semantics $(3, 2, 3)$ becomes the starting point for the subsequent step of backpropagation (instruction $\times$).

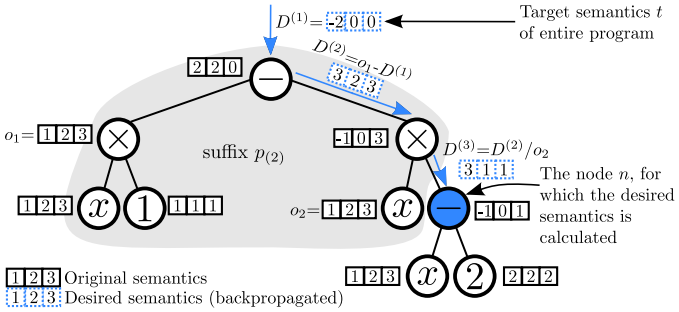


Figure 1: Semantic backpropagation example. Black vectors are the actual semantics at each program node (i.e., the semantics of the corresponding subtree). The desired semantics (the blue dotted vector) is propagated from the root node to the blue node n.

The function INVERT, defined in Table I, carries out the inversion of the execution of a single instruction at a root of the subtree a with respect of its kth argument for the specific context formed by all the remaining arguments. If a has two or more arguments, the result of INVERT depends not only on the output of a, but also on the values supplied by the other arguments of a (the context). For instance, INVERT applied to the first fitness case and the root node of the program shown in Fig. 1, i.e., $\text{INVERT}(c_1 - c_2, 2, o)$, operates on $c_1 = 1$ and $o = -2$. Accordingly to the second row of Table I, it returns $\text{INVERT}(1 - c_2, 2, -2)$, i.e., $c_1 - o = 1 - (-2) = 3$.

For such bijective instructions, inversion is unambiguous, i.e., $\text{INVERT}(a, k, o)$ returns a single value. Otherwise, INVERT can be ambiguous in returning more than one desired value. For instance, INVERT returns two values for $\log|\cdot|$. The periodic functions can be inverted in infinitely many ways, so for sin and cos we limit the set of desired values to two arbitrarily chosen ones.

Another form of ambiguity results from ineffective code. Consider the program $c_1 \times c_2$ where $c_2 = 0$. No matter what we substitute for c_1, the output of the program does not change. If the desired output equals zero, the set of desired values comprises all real numbers. In such cases, INVERT returns the special symbol '$*$'.

$\text{INVERT}(a, k, o)$ returns an empty set if no value passed as the kth argument to a can make it return o. For instance, the exponential function cannot produce negative values.

In these two cases (INVERT()=$\emptyset$ or $\{*\}$), further propagation cannot change the contents of D_i, so SEMANTICBACKPROPAGATION stops the traversal of the path (line 5 in Alg. 1) and the algorithm proceeds to the next fitness case. Finally, the algorithm gathers into a tuple the sets D_i of desired values computed for each fitness case, and returns that tuple as the desired semantics.

C. Impact of program inversion on fitness landscape

Technically, semantic backpropagation propagates only the specific desired values derived from the target (i.e., certain *points* in the semantic space). However, the resulting desired semantics becomes a subtarget for a separate search process using the search operators we define in the next section.

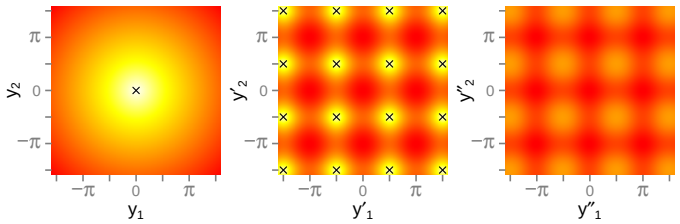


Figure 2: Inverse transformation of the original fitness landscape (left) by the program $[x \sin \cos]$. Instructions are considered in reverse order: $\cos^{-1}$ (b) and $[\cos^{-1} \sin^{-1}]$ (c). Heatmaps present the fitness of the entire program as a function of semantics at a particular point of execution. White is the best fitness, darker (red) area is the worst. Crosses mark the target of the original task in (a) and the subtargets in (b).

That search process will be driven by semantic distance from the subtarget. It is then essential to understand how the semantic distance from the target and the semantic distance from a subtarget relate to each other.

To provide such insight, we consider an example of a univariate symbolic regression problem with the fitness function (Eq. 2) being the Euclidean distance. For the sake of simplicity, we consider only two fitness cases and assume the target $t = (0, 0)$. Consistently with Section III-A, the semantic space here is $\mathbb{R}^2$ and the fitness landscape hovering over it is an 'upside-down' Euclidean cone in three dimensions, with apex at point $(0, 0)$. This fitness landscape is presented as a heatmap in Fig. 2a. The coordinates of every point in this heatmap correspond to the outputs (y_1, y_2) returned by a hypothetical program for the two fitness cases, and the color of the point reflects program's fitness.

We will demonstrate now how, by transforming the desired outputs, semantic backpropagation also implicitly transforms the entire fitness landscape (i.e., not only the points in Fig. 2a, but also the corresponding fitness values). Consider the program suffix $\cos$. For every point (y_1, y_2) in the heatmap in Fig. 2a, we can find all points $(y'_1, y'_2) \in \mathbb{R}^2$ such that $\cos(y'_1) = y_1$ and $\cos(y'_2) = y_2$, and label them with the same fitness (color) as (y_1, y_2). The resulting heatmap, shown in Fig. 2b, is the transformed fitness landscape as seen by the programs prepended to the suffix $\cos$. For any program p, the color of the point $s(p) \in \mathbb{R}^2$ in Fig. 2b reflects the fitness of the compound program $[p \cos]$. That landscape has multiple subtargets (marked by crosses), since cosine is periodic.

This backpropagation of the fitness landscape continues as SEMANTICBACKPROPAGATION proceeds on the path from the tree root to the selected node n (Alg. 1). Given the heatmap shown in Fig. 2b, we can perform an analogous calculation for, e.g., instruction $\sin$, considering so the program suffix $[\sin \cos]$. The fitness landscape of the subtask defined by this suffix is shown in Fig. 2c. However, this time the subtargets do not propagate from Fig. 2b to Fig. 2c: the global optima disappear, since $\nexists y'' : \cos(\sin(y'')) = 0$. Thus, no program of the form $[p \sin \cos]$, where p is an arbitrary subprogram, can be a solution to this programming task.

As the example illustrates, the number of subtargets can grow with suffix length (one target in Fig. 2a, many subtargets in Fig. 2b). That growth can be exponential in function of the number of fitness cases and suffix length. However, the desired semantics, by storing the desired output for each fitness case *independently*, can capture the subtargets in a memory-efficient way; the 16 subtargets in Fig. 2b require only eight values in the desired semantics: $D = (\{-\frac{3\pi}{2}, -\frac{\pi}{2}, \frac{\pi}{2}, \frac{3\pi}{2}\}, \{-\frac{3\pi}{2}, -\frac{\pi}{2}, \frac{\pi}{2}, \frac{3\pi}{2}\})$.

The key observation following from the above example is that semantic backpropagation transforms the entire fitness landscape. Given a program p with semantics located somewhere in Fig. 2b, by modifying that program so that its semantics becomes closer to any of the subtargets, we cause the semantics of the complete program $[p \cos]$ to approach the original target t in Fig. 2a. This is the main motivation for designing search operators that employ semantic backpropagation.

V. SEMANTIC BACKPROPAGATION IN SEARCH OPERATORS

A. Common working principle

We propose two genetic search operators that employ semantic backpropagation: a unary *Random Desired Operator* (RDO) that can be considered as a form of mutation, and *Approximately Geometric Semantic Crossover* (AGX). Both start with selecting a random node n in the parent program p, which divides p into a prefix and suffix. Then, they call SEMANTICBACKPROPAGATION to determine a subtask. Next, the operators attempt to solve the subtask by an exhaustive search of a library of programs described in Section V-D. The best program found in this way replaces the corresponding prefix in p, yielding the offspring.

The main difference between RDO and AGX lies in what they set as the starting semantics in the backpropagation process, i.e., the argument t in the SEMANTICBACKPROPAGATION(t, p, n) call. RDO uses the original target provided by a task. AGX employs a 'synthetic' target, and so can be applied to tasks for which the original target is not explicitly known.

B. Random Desired Operator

The *Random Desired Operator* (RDO), presented in Alg. 2, which we first proposed in [7], follows the above working principle. A random node n is first selected in the parent program p. Next, SEMANTICBACKPROPAGATION determines the desired semantics D associated with the selected node. D stores a set of subtargets and thus defines a programming subtask. The subtask is solved in line 4 by calling LIBRARYSEARCH(L, D) described in Section V-D. LIBRARYSEARCH(L, D) returns a program from the library that is the closest to the desired semantics D in terms of the measure adopted there. Due to the limited size of the library, LIBRARYSEARCH is not guaranteed to actually *solve* the subproblem D, i.e., to find a program in L that exactly matches D.

Note that RDO does not check whether D contains empty sets and searches the library also in such cases. This may seem futile, as no program (whether from library or not) can match such a D perfectly. However, in such cases LIBRARYSEARCH

Algorithm 2 RDO search operator. Parameters: t: target semantics, p: parent tree, L: library of programs. $\text{REPLACE}(p, n, p')$ replaces the subtree rooted in node n in tree p with the tree p'.

```
1: function RDO(t, p, L)
2:     n ← SELECTRANDOMNODE(p)
3:     D ← SEMANTICBACKPROPAGATION(t, p, n)
4:     p' ← LIBRARYSEARCH(L, D)
5:     o ← REPLACE(p, n, p')
6:     return o
7: end function
```

Algorithm 3 AGX crossover operator. Parameters: p_1, p_2: parent programs (trees), L: library of programs.

```
1: function AGX(p1, p2, L)
2:     m ← MIDPOINT(s(p1), s(p2))
3:     o1 ← RDO(m, p1, L)
4:     o2 ← RDO(m, p2, L)
5:     return {o1, o2}
6: end function
```

will find a program p' that matches D as well as possible on the other fitness cases, and RDO will paste it into the offspring. Behavior on the non-matched fitness cases can hopefully be fixed by RDO mutations in the subsequent generations.

C. Approximately Geometric Semantic Crossover

RDO uses as its goal the original target of the search process, t. This is particularly useful in domains where the desired program outcome is explicitly given as a part of task formulation (which we assumed to this point of the paper, particularly in Eq. 2). The typical formulation of symbolic regression and Boolean function synthesis belong to this category.

However, there are tasks in which the target is not explicitly known to the search algorithm, and the fitness function is a black-box that can only be queried for particular programs. For instance, it may be the case that the target contains confidential information and cannot be revealed to the experimenter.

At first sight, the reasoning presented so far is not applicable to such tasks. Indeed, $\text{RDO}(t, p, L)$ cannot be called because t is not known to the search algorithm and thus cannot be propagated back through p. However, a distance-based fitness function (Eq. 2) induces a conic fitness landscape no matter whether the location of the cone apex is known to the algorithm or not. Such fitness landscapes, unimodal and devoid of plateaus, are in general easy to search.

In particular, geometric crossover operators [24] have been demonstrated to perform particularly well on this kind of problem. A recombination operator is a geometric crossover under metric d if its offspring are in the d-metric segment between its parents. Geometric crossover can be easily adopted in semantic GP because, with program semantics represented as a vector of outputs, the semantic space is naturally a vector space. For some metric, geometric crossover offers attractive convergence properties. E.g., for the Euclidean metric, the offspring on the segment cannot be worse than the worst of the parents (see [25] for a more thorough explanation).

The *Approximately Geometric Semantic Crossover* (AGX), which we originally proposed in [6], is an operator that combines semantic backpropagation with geometric crossover. Presented in Algorithm 3, AGX chooses a point m on the segment connecting the semantics of the parents, and attempts to produce a program that matches m by applying RDO to each of the parents, with the target set to m. In other words, uninformed about the true target, AGX uses a point on a segment between parents' semantics as a surrogate target.

Finding a midpoint m of the segment spanning the semantics of parent programs p_1 and p_2 is domain-dependent. For numeric semantics and Euclidean metric, $\text{MIDPOINT}(s(p_1), s(p_2))$ returns $m = (s(p_1) + s(p_2))/2$. For binary vectors and Hamming metric, MIDPOINT returns an arbitrarily chosen point m that is (i) located on the segment (i.e. $d(s(p_1), m) + d(m, s(p_2)) = d(s(p_1), s(p_2))$) and (ii) possibly equidistant to the endpoints of the segment (i.e., $|d(s(p_1), m) - d(m, s(p_2))| \leq 1$).

D. Solving subtasks by library search

RDO and AGX use the parent individuals to derive the subtasks from the original programming task. In Section IV-A, we provided evidence that subtask solutions can be shorter than solutions to the original task. Therefore, to solve a subtask, rather than using a sophisticated heuristic like GP, we resort to simpler means and perform exhaustive search in a set of programs with precomputed semantics, which we term a *library*. This process hides under the LIBRARYSEARCH call in Algorithm 2. Given a library L and a desired semantics D, $\text{LIBRARYSEARCH}(L, D)$ calculates the best match between the components of D and the semantics of every program in the library, i.e., finds a program p in L that minimizes:

$$\arg\min_{p \in L} \min_{y \in D_1 \times \ldots \times D_l} d\left(y, s(p)\right) \tag{4}$$

When minimizing this expression, we discard from the Cartesian product all sets D_i such that $D_i = \emptyset$ or $* \in D_i$. In consequence, the distance $d(y, s(p))$ is calculated only on the remaining (well-defined) components of semantics $s(p)$.

Note that posing a programming subtask in this way is different from the formulation of the original programming task (Def. 3), where the target of the search process was a *single* vector (combination of desired output values). Here, a program that minimizes the distance to *any* of the subtargets in D is sought.

After finding the program in L that minimizes Formula 4, we verify whether a constant semantics would give an even better match. By doing so, we not only expect to sometimes find a constant that reduces the matching distance d, but also provide the population with an additional influx of potentially useful constants (ERCs) and reduce bloat. We calculate the constant that minimizes the overall divergence from the desired values on all fitness cases by solving the following special case of Formula 4:

$$\arg\min_{c} \min_{y \in D_1 \times \ldots \times D_n} d(y, [c]) \tag{5}$$

where $[c]$ denotes a vector of l components, all set to c. As in Formula 4, the Cartesian product involves only the well

defined D_i components. This expression is minimized over c varying in the domain characteristic for the problem ($\mathbb{R}$ for symbolic regression, and $\{0, 1\}$ for the Boolean problems). If value of (5) is smaller than value of (4), LIBRARYSEARCH returns c, otherwise it returns the program found in the library.

We solve both these problems by separate minimization in each dimension of Formula 4, which can be done in a polynomial time thanks to the properties of Minkowsky distance. The details are presented in the Appendix[6].

We consider two sources of programs for libraries. A static library L is filled up with all program trees up to certain height[7] limit and does not change during evolutionary search. A *dynamic* library contains all subtrees collected from all individuals in the current population and thus varies along the search process.

A library, whether static or dynamic, stores only the semantically unique programs. If two candidate programs have the same semantics, only the shorter one is included in the library. Note that the verification of semantic uniqueness, intended to keep the library size at bay and reduce program bloat, is done without any threshold[8]. Albeit that, this is the main computational cost of library generation. Thus, maintaining a dynamic library is typically more time consuming.

VI. THE EXPERIMENT

A. Setup

To assess the benefits of semantic backpropagation, we compare the performance of RDO (Sec. V-B) and AGX (Sec. V-C) to the following reference search algorithms:

GPX: GP using standard subtree crossover (90%) and subtree mutation (10%) by Koza [26]. Standard crossover produces offspring by swapping two randomly selected subtrees in parent programs. Mutation replaces a randomly selected subtree in the parent with a randomly generated one. We use mutation to improve GPX's results and make it a more challenging opponent (GP without mutation achieved notably worse results in a preliminary series of runs).

LGX: GP using only the Locally Geometric Semantic Crossover [18]. Given two parent trees, LGX draws a crossover locus only from their structurally homologous region, like the one-point crossover by Langdon [27]. Next, it calculates the midpoint of the segment connecting the semantics of the subtrees rooted at the chosen locus in both parents. Then it calls LIBRARYSEARCH to find the program that is semantically most similar to the midpoint, and pastes that program into parents at the selected locus (see [18] for more details). We chose LGX as a control approach because its design is motivated by the same geometric properties that found AGX, and it also uses a library of programs. Moreover, in [18] we compared LGX with Semantic Aware Crossover and Semantic Similarity Crossover by Nguyen *et al.* [13], so by using LGX as a reference we can also indirectly compare RDO and AGX to those operators.

We apply these two control setups and GP running RDO and AGX to 18 commonly used benchmarks that represent two groups: symbolic regression tasks and Boolean function synthesis tasks (Table II). The symbolic regression benchmarks come from [26], [28], while the Boolean benchmarks are taken from [26], [29].

Instruction sets used in symbolic regression and Boolean benchmarks are listed in Table III. For GPX, ERCs in symbolic regression are random constants drawn from the interval $[-1, 1]$. For RDO, AGX and LGX, constants are generated by algorithm LIBRARYSEARCH described in Section V-D. On the other hand for Boolean benchmarks, to maintain consistency with the symbolic regression setup, we allow the programs to

Table II: Benchmarks used in the experiment. For symbolic regression training set contains 20 fitness cases selected *equidistantly* from the given range, whereas test set contains 20 *uniformly* drawn points from the same range.

Symbolic regression benchmarks		
Problem	Definition (formula)	Range
Septic	$x^7 - 2x^6 + x^5 - x^4 + x^3 - 2x^2 + x$	$[-1, 1]$
Nonic	$\sum_{i=1}^{9} x^i$	$[-1, 1]$
R1	$(x + 1)^3/(x^2 - x + 1)$	$[-1, 1]$
R2	$(x^5 - 3x^3 + 1)/(x^2 + 1)$	$[-1, 1]$
R3	$(x^6 + x^5)/(x^4 + x^3 + x^2 + x + 1)$	$[-1, 1]$
Nguyen-6	$\sin(x) + \sin(x + x^2)$	$[-1, 1]$
Nguyen-7	$\log(x + 1) + \log(x^2 + 1)$	$[0, 2]$
Keijzer-1	$0.3x \sin(2\pi x)$	$[-1, 1]$
Keijzer-4	$x^3 e^{-x} \cos(x) \sin(x)(\sin^2(x) \cos(x) - 1)$	$[0, 10]$

Boolean benchmarks			
Problem	Instance	Bits	Fitness cases
even parity	Parity-5	5	32
	Parity-6	6	64
	Parity-7	7	128
multiplexer	Multiplexer-6	6	64
	Multiplexer-11	11	2048
majority	Majority-6	6	64
	Majority-7	7	128
comparator	Comparator-6	6	64
	Comparator-8	8	256

Table III: Evolutionary parameters.

Parameter	Value
Stopping condition	100 generations or optimal solution found
Population size	1024
Fitness function	*Symbolic regression:* Mean of absolute errors *Boolean domain:* Hamming distance
Initialization method	Ramped Half-and-Half algorithm, height range $2 - 6$
Duplicate retries	100 (until accepting a syntactic duplicated individual)
Selection method	Tournament selection, tournament size 7
Max program height	17
Node selection	RDO, AGX: Equal depth probability[a] LGX: Homologous selection GPX: Koza-I (90% nonterminal nodes, 10% leafs) [26]
Instructions	*Symbolic regression*: x, $+$, $-$, $\times$, $/^b$, sin, cos, exp, $\log^b$, ERC *Boolean domain*: $D1...D11$ (inputs depend on a problem instance), and, or, nand, nor, ERC
Number of runs	30

[a] SELECTRANDOMNODE(p) in Alg. 2 draws with uniform probability a random number r from the interval $[1, height(p)]$, picks at random a node in p at depth r and returns it. We found that this selector leads to less bloat than the conventional Koza-style selectors.

[b] log and / are protected. log is defined as $\log |x|$; / returns 0 if divisor is 0.

[6] Available as supplementary material online.

[7] Typically, albeit incorrectly, referred as 'tree depth' in GP literature.

[8] If there is no threshold, semantic uniqueness of the candidate program can be determined in $O(1)$ time, e.g. using hashtable. In contrary threshold forces us to take into account distribution of semantics of programs already present in the library. Thus it can be done, using e.g. binary tree, in $O(\log |L|)$ time, where $|L|$ is size of the library.

Table IV: Cardinality of static libraries of trees up to height 3, depending on the number of inputs in the Boolean task.

# of inputs	5	6	7	8	9	10	11
Cardinality	1072	2524	5126	9370	15836	25192	38194

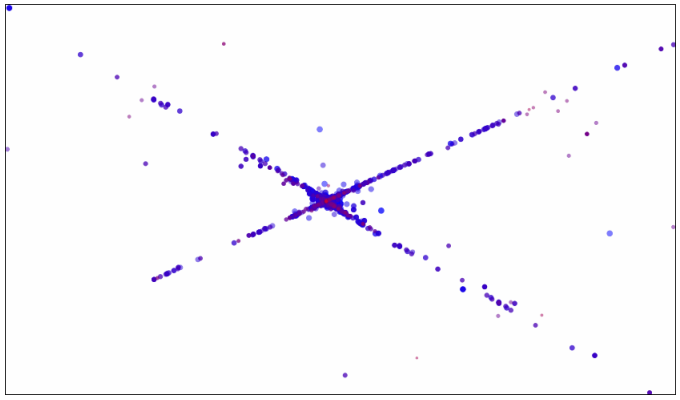


Figure 3: Distribution of program semantics in the static library ($h = 4$) for the symbolic regression domain. The coordinates of points have been determined by transforming the original 20-dimensional space (20 fitness cases, x evenly distributed in $[-5, 5]$) using principal component analysis and discarding all but the two first components. Dots reflect program size: red for the smallest programs (a single node), blue for the largest programs in the library ($2^h - 1 = 15$ nodes).

use the constants *true* and *false*.

RDO, AGX and LGX use the same libraries of programs. We consider static libraries and population-based (dynamic) libraries (Section V-D). A static library contains all program trees up to height h. That limit is marked by a subscript, e.g. RDO_3 refers to a static library for $h = 3$, while RDO_p refers to the population-based (dynamic) library.

Program semantics is a vector of length 20 for all regression benchmarks, as this is the number of fitness cases available in the training set (Table II). Because there is only one terminal symbol x, the number of programs grows moderately with tree height. This allows us to consider two libraries for the regression problems, a small one ($h = 3$) and a large one ($h = 4$), which hold 289 and 111458 semantically unique programs, respectively. In the following, the terms *small* and *large* refer to static libraries only.

For the Boolean problems, the training set comprises all input combinations, so the length of semantics can be even two orders of magnitude greater than for regression problems (see the last column of Table II). Also, depending on the benchmark, there are between five and eleven input terminals. This makes the large ($h = 4$) static library technically infeasible, so we use only the small library for the Boolean domain. Table IV presents the sizes of static libraries used in the Boolean tasks.

Larger libraries are desirable, as they offer richer semantic diversity. The distribution of program semantics converges with growing program size to a strongly non-uniform distribution (see demonstration for the Boolean domain in [8], [30]). However, with the small programs in the libraries considered here, we are far from that convergence, which we verified experimentally. Figure 3 visualizes the two first principal components of semantics of the programs in the large library (symbolic regression) and marks program size with color. The small programs clearly group around the origin of the coordinate system, while the larger ones spread further from it. Thus, the large library for the regression domain is semantically more diverse. However, the price paid for greater semantic diversity is the library size and, consequently, more costly library search.

Other parameters of the evolutionary algorithm can be found in Table III. Among them is the upper limit on tree height (17), observed by all methods. For RDO, AGX and LGX this implies that LIBRARYSEARCH receives an extra argument which specifies what is the maximum height of programs that can be inserted at the selected locus in the parent program (e.g., if the locus is at depth 7, that argument is 10). LIBRARYSEARCH ignores the programs in the library that would violate this constraint.

The Java source code of our implementation of Semantic Backpropagation, RDO, AGX, LGX and GPX is available at www.cs.put.poznan.pl/tpawlak/link/?IEEESemanticBackprop.

B. Performance of the operators

Figure 4 presents the mean (minimized) fitness of the best-of-generation individuals for the symbolic regression and Boolean benchmarks. The curves plot the averages of 30 runs per method with 0.95-confidence intervals.

For all symbolic regression benchmarks, RDO_4 is the unquestionable winner in terms of best-of-run fitness and speed of convergence. The second place belongs to AGX_4 or RDO_p, depending on the task, however RDO_p converges noticeably quicker than AGX_4. AGX_4 performs better than its main competitor, LGX_4. GPX, the only non-semantic operator, fares worse than most of the other methods.

In almost all cases, a method equipped with the large library is better than the same method equipped with the small library (except for LGX in Nguyen-6 problem). This is not surprising, since the large library provides a more diversified choice of matches for desired semantics. The operators that use the dynamic library typically fare in between.

For Boolean benchmarks, RDO again achieves the best fitness, however this time with the dynamic library (RDO_p). For each benchmark, the second place is occupied by RDO_3. The performance of AGX is strongly problem-dependent. It converges quickly, leaving behind LGX and GPX, however later it is usually caught up or overtaken by LGX (except for the multiplexer problems where LGX takes precedence of AGX from the beginning). GPX is usually in the second half of method ranking at the end of runs.

C. Generalization

The ability to generalize beyond the training data is a desirable property of a GP system and can be investigated using the tools borrowed from traditional machine learning [31]. To assess the generalization performance, we applied the best-of-run individuals to the test sets defined in Table II. The medians of errors committed by those individuals are presented

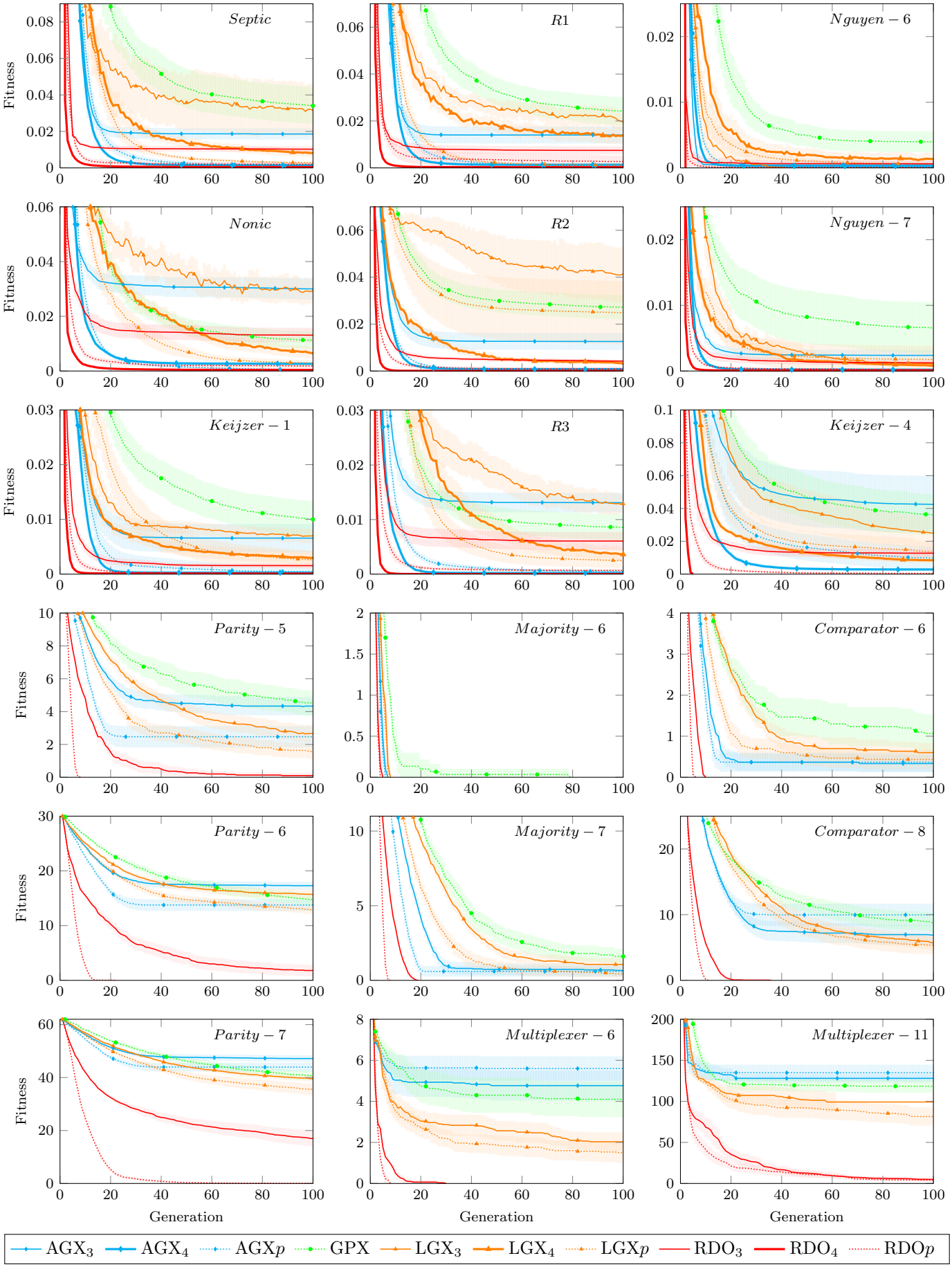
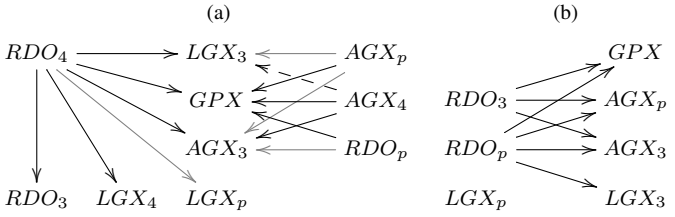


Figure 4: Average fitness and 0.95 confidence interval of the best-of-generation individuals.

Table V: Median error committed by the best-of-run individuals on the test set (cf. Table II). Bold values refer to the lowest error achieved on the problem (a row).

Problem	AGX_3	AGX_4	AGX_p	GPX	LGX_3	LGX_4	LGX_p	RDO_3	RDO_4	RDO_p
Septic	.0117	.0018	.0055	.0364	.0105	.0048	.0039	.0095	**.0009**	.0017
Nonic	.0235	.0023	.0031	.0101	.0245	.0061	.0031	.0097	**.0009**	.0053
R1	.0110	.0011	.0023	.0270	.0151	.0075	.0007	.0071	**.0003**	.0017
R2	.0091	.0007	.0012	.0393	.0259	.0021	.0100	.0030	**.0000**	.0012
R3	.0108	.0002	.0016	.0068	.0107	.0024	.0017	.0055	**.0000**	.0025
Nguyen-6	**.0000**	.0003	.0002	.0018	**.0000**	.0002	.0003	**.0000**	**.0000**	**.0000**
Nguyen-7	.0016	.0002	.0002	.0026	.0006	.0006	.0005	.0011	**.0001**	.0005
Keijzer-1	.0059	.0004	.0007	.0181	.0080	.0024	.0023	.0022	**.0002**	.0008
Keijzer-4	.0371	.0127	.0662	.0803	.0450	.0400	.0431	.0239	**.0000**	.0207

Figure 5: The results of post-hoc analysis of Friedman's test visualized as outranking graphs. $A \to B$ indicates that A outranks B in a significant manner ($p < 0.05$ in Table VI). For symbolic regression (a), gray, dashed, and black arrows represent outranking that holds for, respectively, training sets only, test sets only, and all sets. For the Boolean tasks (b), there are no test sets and the arrows pertain to the training sets.



in Table V. This test was conducted only for the symbolic regression benchmarks, since in the Boolean benchmarks the training sets enumerate all possible program inputs.

RDO_4 clearly generalizes best on all benchmarks. AGX_3, LGX_3, RDO_3, RDO_p match its performance only on the relatively easy Nguyen-6 problem. The results of all library-backed operators are markedly related to library size: the operators equipped with the large library generalize better. The ranking of methods on the test set is usually consistent with the end-of-run results for the training sets (Fig. 4). This suggests that the methods based on semantic backpropagation, in particular RDO, are quite resistant to overfitting. This holds also for the large library, which, by offering greater semantic diversity, allows a search operator to fit the training data more tightly.

D. Statistical significance

To draw qualitative conclusions about methods' comparative performance, we carried out Friedman's test for multiple achievements of multiple subjects [32]. Compared to ANOVA, this procedure does not require the distributions of variables in question to be normal. Because the Boolean tasks are fundamentally different in having no test set, we consider the domains and the training and test sets separately. In all cases the test concludes that there is at least one pair of operators with statistically significant difference of performance ($\alpha = 0.05$; the particular p-values are presented in the headers of Tables VIa-c).

To determine for which pairs of operators the difference is significant, we conducted a post-hoc analysis using symmetry test [33], shown in Table VI. In Fig. 5 we visualize the statistically significant part of these results as outranking graphs.

Table VI: Post-hoc analysis of Friedman's test using symmetry test conducted on average and median errors of the best-of-run individuals applied to training and test set, respectively. Every cell gives the probability of erroneously judging the method in a row as being better than the method in a column. Significant values marked in bold ($\alpha = 0.05$).

(a) Symbolic regression, training set (Friedman's p-value = 3.45×10^{-7}).

	AGX_3	AGX_4	AGX_p	GPX	LGX_3	LGX_4	LGX_p	RDO_3	RDO_4	RDO_p
AGX_3				1.000	1.000					
AGX_4	**0.012**		1.000	**0.001**	0.051	0.637	0.690	0.365		
AGX_p	**0.003**			**0.000**	**0.014**	0.364	0.417	0.163		
GPX					0.994					
LGX_3										
LGX_4	0.831			0.470	0.971		1.000	1.000		
LGX_p	0.788			0.416	0.957			1.000		
RDO_3	0.965			0.741	0.998					
RDO_4	**0.000**	0.885	0.982	**0.000**	**0.000**	**0.019**	**0.024**	**0.005**		0.850
RDO_p	**0.016**	1.000	1.000	**0.002**	0.065	0.690	0.741	0.416		

(b) Symbolic regression, test set (Friedman's p-value = 6.94×10^{-8}).

	AGX_3	AGX_4	AGX_p	GPX	LGX_3	LGX_4	LGX_p	RDO_3	RDO_4	RDO_p
AGX_3				0.924	1.000					
AGX_4	**0.033**		0.955	**0.000**	**0.015**	0.407	0.825	0.515		0.999
AGX_p	0.600			**0.022**	0.433					
GPX										
LGX_3				0.976						
LGX_4	0.991		0.993	0.308	0.963					
LGX_p	0.825		1.000	0.068	0.682	1.000				
RDO_3	0.976		0.998	0.223	0.924	1.000	1.000			
RDO_4	**0.000**	0.976	0.285	**0.000**	**0.000**	**0.020**	0.129	**0.034**		0.655
RDO_p	0.243		1.000	**0.003**	0.142	0.881	0.996	0.936		

(c) Boolean tasks, training set (Friedman's p-value = 3.37×10^{-4}).

	AGX_3	AGX_p	GPX	LGX_3	LGX_p	RDO_3	RDO_p
AGX_3		1.000					
AGX_p							
GPX	1.000	0.999					
LGX_3	0.993	0.999	0.968				
LGX_p	0.291	0.430	0.181	0.738			
RDO_3	**0.006**	**0.013**	**0.002**	0.056	0.805		
RDO_p	**0.001**	**0.002**	**0.000**	**0.013**	0.507	0.999	

For the symbolic regression domain, the graph shows that RDO_4 outranks five out of nine other setups on both training and test sets, and one setup (LGX_p) on the training set only. No other method is equally superior. The other semantic backpropagation-backed method, AGX_4, is also quite successful, outranking two other setups on both sets, and a single other setup on the test set only.

The behaviors of both RDO and AGX indicate that using the large static library ($h = 4$) significantly improves performance when compared to the small static library ($h = 3$).

In the Boolean domain the superior setup is RDO_p (recall that the large static library is absent here). It outranks four out of six other setups. RDO_3 outranks three methods.

E. Bloat analysis

Contrary to GPX that generates replacing subtrees at random, RDO and AGX choose them from a relatively small library in a semantically biased manner. We expect this to impact the distribution of sizes of replacing subtrees, and

Table VII: Average program size in the last generation (maxima in bold).

Problem	AGX_3	AGX_4	AGX_p	GPX	LGX_3	LGX_4	LGX_p	RDO_3	RDO_4	RDO_p
Septic	60	302	1051	124	26	169	459	81	338	**1922**
Nonic	45	309	989	100	29	216	432	85	344	**1125**
R1	56	313	944	104	29	112	332	78	303	**1519**
R2	70	307	926	72	22	69	291	88	270	**1238**
R3	62	266	934	75	31	154	328	87	300	**1200**
Nguyen-6	16	220	**905**	73	13	95	276	25	16	482
Nguyen-7	67	262	862	74	33	223	327	73	279	**981**
Keijzer-1	77	307	**905**	145	34	97	452	101	315	800
Keijzer-4	84	287	**878**	160	86	302	630	97	22	878
Parity-5	133	—	**2060**	427	182	—	1506	101	—	484
Parity-6	148	—	**2601**	474	170	—	1448	163	—	1601
Parity-7	171	—	2665	484	190	—	1402	241	—	**3584**
Majority-6	23	—	**196**	97	29	—	145	21	—	123
Majority-7	126	—	**1541**	502	168	—	1171	76	—	360
Comparator-6	62	—	**1189**	308	100	—	907	35	—	185
Comparator-8	118	—	**2390**	306	124	—	1105	62	—	538
Multiplexer-6	26	—	**3457**	215	85	—	1144	32	—	215
Multiplexer-11	15	—	**3484**	198	56	—	1131	128	—	3063

indirectly of programs in the population (even given that the corresponding evolutionary runs of particular methods start here from identical initial populations).

Table VII presents the average program size (number of tree nodes) in the last generation. The most bloating are the methods equipped with the dynamic library, especially RDO_p for regression and AGX_p for Boolean benchmarks. Clearly, this is due to the size of library programs. The average program size in a dynamic library is not only typically greater than for the static library, but also will increase as soon as population starts to suffer from bloat. However, the method performing best on symbolic regression (RDO_4) does not generate the largest trees, although they are still greater than the ones produced by GPX. On the other hand RDO_p, the best performing method on the Boolean problems, does not bloat very much in this domain, generating substantially bigger trees than GPX's only in three benchmarks. Note that the programs yielded by RDO_3 are even smaller than those produced by GPX, while their fitness is noticeably better.

F. Analysis of RDO

The results presented above demonstrate that RDO and AGX outperform the standard GP search and the semantic-aware LGX on a representative suite of benchmarks. One can suppose that they owe this superiority to semantic backpropagation, which correctly identifies the desired semantics, and to library search that finds subprograms that closely match the desired semantics. However, the above experiment does not prove that directly. The dynamics of evolutionary search is very complex and other causes could potentially be behind this result.

To verify the above supposition, we carry out an experiment that examines the behavior of the simpler of the two operators, RDO, on a population of random programs. For symbolic regression and Boolean domain independently, we generate a sample of $10,000$ random programs according to the setup in Table III. Next, we apply RDO to each program, independently for each of the target semantics of the nine benchmarks, which results in $90,000$ applications of RDO. For every application of RDO, we note two pieces of information. Firstly, we record whether the semantic distance between the desired semantics D (Algorithm 2) and the program p' found by LIBRARYSEARCH is greater than, equal to, or smaller than the distance between D and the subtree being replaced, i.e., n. Secondly, we check whether the fitness of the offspring o resulting from RDO is worse than, equal to, or better than the fitness of the parent p. Technically we used equality threshold 10^{-10}, to handle floating point errors. Table VIII presents the statistics gathered in this way from the runs for all benchmarks, separately for the static and dynamic libraries and for the symbolic regression domain and the Boolean domain.

Do the data in Table VIII confirm that LIBRARYSEARCH finds programs that are better than the subtrees being replaced? The three qualitative outcomes of LIBRARYSEARCH correspond to table rows and are summarized in the last column of each table. Of these three possibilities, the first one is the least desired (the program found by LIBRARYSEARCH is a worse match for D than the 'old' subtree n), and the third is the one we are most interested in. The figures clearly show that the fraction of LIBRARYSEARCH failures is very low, even for the small library. Most often, LIBRARYSEARCH manages to find a program that is a better match for the desired semantics. The fraction of neutral cases (i.e., when the two compared semantic distances are equal) is substantial, but much lower. Expectedly, it is higher in the Boolean domain where the discrete nature of the Hamming distance makes ties more likely.

This result is promising, however it only characterizes the *internal* operation of RDO. Do the programs found by LIBRARYSEARCH cause the offspring to be more fit than the parent?

This question can be answered positively by examining the columns of Tables VIIIa–e. Most applications of RDO produce an offspring that outperforms the parent. The distributions are not as extreme as for the rows, because lowering the subtree's semantic distance to D does not guarantee lowering the entire program's semantic distance to the target t. The numbers in the lower left cells indicate that distortions of fitness landscape, often resulting from inversions of multiple instructions and thus very complex (cf. Table I and Figs. 2a–2c), can cause the offspring to be sometimes worse than the parent, even if the subprogram found by LIBRARYSEARCH matches the desired semantics better than the old subprogram. Nevertheless, this is infrequent compared to improvements.

For the Boolean problems, large numbers clearly group on the diagonals, suggesting strong causality between the qualitative outcome of library search and the performance of offspring. For symbolic regression this is not so prominent, therefore we apply the χ^2 interdependence test to each table in Table VIII separately and report the results in Table VIIIf. χ^2 yields positive outcome ($p \ll 0.001$) for all tables. The two considered random variables are significantly interdependent for every library, which allows us to conclude that RDO's performance is due to the outcomes of LIBRARYSEARCH.

In every call, LIBRARYSEARCH does not only find a subprogram in a library, but also determines the constant semantics that matches best the desired semantics D. If such a constant has smaller semantic distance from D, LIBRARY-

Table VIII: Joint statistics of the change of semantic distance to desired semantics (rows) and the change of fitness (columns) for RDO applied to $10,000$ random individuals for each benchmark (nine in each domain). Rows marked 'Worse' count how many times $d(s(p'), D) > d(s(n), D)$ in Alg. 2; 'Same' and 'Better' for '$=$' and '$<$', respectively (assuming 10^{-10} tolerance). Columns correspond to $d(s(o), t) > d(s(p), t)$ in Alg. 2, and '$=$' and '$<$', respectively. Table (f) presents Cramer's V [34], the measure of association ranging in $[0, 1]$, for Tables (a) – (e) (χ^2 test was conclusive for $p \ll 0.001$ for every table). Empty rows in (a) – (e) were excluded from tests.

(a) RDO_3, symbolic regression.

LibrarySearch \ Offspring's fitness vs. parent's	Worse	Same	Better	Sum
Worse	50	6	16	72
Same	2043	5431	1689	9163
Better	8469	5443	66853	80765
Sum	10562	10880	68558	90000

(b) RDO_4, symbolic regression.

LibrarySearch \ Offspring's fitness vs. parent's	Worse	Same	Better	Sum
Worse	0	0	0	0
Same	1996	4889	1515	8400
Better	8679	5656	67265	81600
Sum	10675	10545	68780	90000

(c) RDO_p, symbolic regression.

LibrarySearch \ Offspring's fitness vs. parent's	Worse	Same	Better	Sum
Worse	0	0	0	0
Same	4326	3506	481	8313
Better	8363	5396	67928	81687
Sum	12689	8902	68409	90000

(d) RDO_3, Boolean domain.

LibrarySearch \ Offspring's fitness vs. parent's	Worse	Same	Better	Sum
Worse	288	0	2	290
Same	2	24158	25	24185
Better	133	440	64952	65525
Sum	423	24598	64979	90000

(e) RDO_p, Boolean domain.

LibrarySearch \ Offspring's fitness vs. parent's	Worse	Same	Better	Sum
Worse	0	0	0	0
Same	0	14122	0	14122
Better	31	318	75529	75878
Sum	31	14440	75529	90000

(f) Interdependence tests on Tables (a) – (e).

Library	Domain: Regression	Domain: Boolean
Static small ($h = 3$)	0.37	0.90
Static large ($h = 4$)	0.49	—
Dynamic	0.54	0.86

Table IX: Empirical probability that LIBRARYSEARCH returns a constant for each library and domain.

Library	Symbolic regression	Boolean domain
Static small ($h = 3$)	0.209	0.311
Static large ($h = 4$)	0.051	—
Dynamic	0.039	0.000

SEARCH returns it rather than a program from the library. This justifies posing the following question: does RDO perform so well because the libraries contain the needed programs and LIBRARYSEARCH manages to find them, or because LIBRARYSEARCH resorts to constants?

To verify this, when collecting the data reported in Table VIII, we counted also how many times LIBRARYSEARCH returned constants. These statistics, presented in Table IX, show that LIBRARYSEARCH resorts to constants relatively infrequently for the large library and for the dynamic libraries. For the small library, where the choice of programs is much smaller, this is more common. Nevertheless, comparing these numbers with the data in Table VIII allows us to conclude that neither the desired behavior of LIBRARYSEARCH, nor the improvements elaborated by RDO, can be explained by constants alone.

All above statistics have been gathered from samples of random individuals, so they characterize the behavior of RDO in the first generations of evolutionary runs. It can be expected that with run progress, getting closer to search targets becomes harder, and the frequency of improvements decreases (which is however typical for all search operators). Indeed, this is clearly visible in many fitness graphs in Fig. 4. Nevertheless, the capability of making quick progress in the early phase of the search compensates for this deficiency.

The conclusions formulated above can be to a certain extent extrapolated to AGX, because it calls RDO to modify the offspring (Alg. 3). However, a separate analysis, which we skip here for brevity, would be necessary to answer analogous questions for AGX.

VII. DISCUSSION

The experiments demonstrate that RDO operates as expected (Sec. VI-F) and outperforms all other operators in both considered domains, while AGX proves useful only in the regression domain (Sec. VI-B). What are the causes of these differences?

RDO's performance results from the explicit use of the most informative part of problem specification: the target. By using target as a goal of semantic backpropagation, RDO aims at solving the problem *directly*, i.e., without the 'intention' of gradually approaching the target. With a little luck, it can choose the right suffix already from the programs available in the initial population, and find a perfectly matching program in the library, thus solving the problem in the first generation. For an analogous thing to happen for AGX, the surrogate target determined by MIDPOINT has to coincide with the true target, which is unlikely, particularly in continuous semantic spaces. Thus, AGX is by design unable to converge faster than RDO.

How is it then possible that AGX fares quite well in the symbolic regression domain? The geometric properties of the semantic space mentioned in Sec. V-C play the key role here. For the continuous semantic space and Euclidean metric d, the fitness landscape forms a Euclidean cone hovering over the semantic space (an example of which was shown in Fig. 2a). The fitness of a point in the semantic space can be obtained by projecting it on the cone and measuring the distance to that projection (the 'height', depicted by color in Fig. 2a). For a pair of parents' semantics $s(p_1)$ and $s(p_2)$, the MIDPOINT procedure appoints as AGX's surrogate target the midpoint m on the segment between $s(p_1)$ and $s(p_2)$ (Algorithm 3). By cone's definition, the image of m projected on the cone cannot be higher than the height of both projections of $s(p_1)$ and $s(p_2)$. Thus, provided that AGX manages to produce a program that matches m (which is not guaranteed, due to imperfect nature of program inversion and library search), that program cannot be worse than the worse of the parents. In

particular, when $s(p_1)$ and $s(p_2)$ happen to be located on the opposite sides of the original target t (and thus their images on the opposites slopes of the cone), the offspring can be more fit (its image lower on the cone) than the fitter of the parents.

This geometric property causes AGX converge faster than GPX and LGX for symbolic regression benchmarks. In the Boolean domain however AGX fares much worse (Fig. 4). The possible cause for this is the structure of the semantic space and fitness landscape which are fundamentally different here from that of the continuous symbolic regression domain. Though segments are well-defined in the Hamming space, the midpoint of a segment is non-unique in general: for two Boolean semantics that differ on n bits, there are roughly[9] $\binom{n}{n/2}$ such midpoints. Contrary to the Euclidean space, such a midpoint can can be less fit than both parents.[10] In the Boolean space AGX can appoint a surrogate target m that pulls the search away from the original target t.

Given this, does AGX have any other virtues that make it potentially useful given RDO's much better performance? We claim it does, because, as signaled in Section V, application areas of RDO and AGX are to a certain extent complementary. The advantage of RDO is that it does not require the semantic distance d to be a norm: it is sufficient for d to be a metric. AGX, to the contrary, requires the semantic space to be a normed vector space, because it needs to construct a midpoint between the parents' semantics. However, AGX offers in exchange the advantage of being ignorant about the search target t; it does not need to explicitly know what is the desired program output. Thanks to that, it can conduct search with fitness values as the only information (apart from the programming language) about the problem being solved. This makes it a convenient method of choice when the target cannot be explicitly revealed to the search algorithm, for instance because of confidentiality issues, or when the target is simply not known, which is the case in control problems (like an artificial ant or pole balancing). We suppose that AGX can perform well also in the latter case, despite the fact that such tasks are in general not unimodal (i.e. there may be more than one target). In a broader perspective, it is interesting to note that the concept of desired semantics introduced here can be used as a generalization of target for multimodal tasks. The algorithms introduced here equal target with a *vector* mostly because of the convention widespread in GP; many of them (in particular SEMANTICBACKPROPAGATION and RDO) would work equally well given task's desired semantics as an input.

Throughout this paper, we considered the ambiguous, one-to-many nature of program inversion as a challenge, because it potentially leads to exponentially many subtargets being derived from the original target (or even infinitely many subtargets for the periodic functions like $\sin$). There is however a flip side to this problem. An ideal solution to the subtask is a program that returns *any* combination of desired outputs from the desired semantics[11]. In general, finding such a program is easier than finding a program that produces a *specific* combination of outputs, which one does when solving the original problem with the single original target. This is an interesting aspect of semantic backpropagation that deserves future investigation.

More generally, semantic backpropagation uses the suffixes of programs in a population to generate a different task that is expected to be easier to solve than the original programming task. To solve the derived subtask, we used here an exhaustive search in a library, mainly because it is conceptually straightforward and reasonably fast, particularly for the small libraries. Other search algorithms, like GP, could be used for that purpose as well. GP could potentially find better programs than those found by LIBRARYSEARCH, given that its search space is much larger than the sizes of the considered libraries.

However, investing substantial computational effort into solving any subtask bears certain risk. A subtask can, but is not guaranteed, to be easier to solve than the original task. In the worst case, it may have no solution in the considered search space. Therefore, rather than appointing a single subtask (or a small sample thereof) and devoting a large amount of search effort to solving it, the algorithms presented here attempt to solve a large number of subtasks (each created by an application of RDO/AGX to parent individual(s)) by means of exhaustive search in a limited library. The tradeoff between the number of considered subtasks and the computational effort devoted to solving them resembles the famous multi-armed bandit paradigm, and is an interesting future research topic.

When applied to a program, RDO temporarily freezes its suffix and manipulates its prefix, so that the modified prefix fulfills the 'expectations' of the suffix (the desired semantics) as close as possible. A hypothetical alternative search operator could swap the roles of prefixes and suffixes, and freeze the prefix while manipulating suffixes. However, this would have at least one practical downside: the *entire* modified program would have to be executed to calculate its fitness. In RDO, programs in the library can be precomputed, because their behavior depends only on program input provided by the fitness cases, which come with the programming task and remain fixed.

The overall computational cost of RDO and AGX is higher than for the conventional GP operators. The main cause is the library search, which can take substantial time, particularly when applied to large libraries. In our previous study [18] we managed to speed up the search using kd-trees [35]. However, such indexing structures can be used only when the object sought for is a single vector of numbers (a point), while our desired semantics represents a *set* of points in general. The conventional indexing techniques need to be extended to meet this requirement and make these search operators more time-efficient. The second, albeit minor factor that influences the computational cost of RDO and AGX is tree size. Similarly to standard GP, programs built by RDO and AGX tend to grow with the time of evolution, which we reported in Section VI-E. We hypothesize that this problem can be addressed using generic bloat prevention techniques, like for instance those

[9] $\binom{n}{n/2}$ for even n; for odd n, there are no equidistant points on the segment, and $n\binom{n-1}{(n-1)/2}$ points where distances from the segment ends differ by one.

[10] Let $s(p_1) = 000$ and $s(p_2) = 011$. The point $m = 001$ is a midpoint on the Hamming segment between these points, but it is further from the target $t = 110$ (distance 3) than either parent (distance 2), and thus has worse fitness.

[11] Example: for desired semantics [{0,1},{2,3}] all programs with the following semantics are perfect matches: (0,2), (0,3), (1,2), (1,3).

that do it implicitly, using spatial population structure [36].

VIII. Conclusions

We demonstrated that inversion of program execution can generate subtasks from the original programming task, and that such subtasks can be solved using exhaustive search in a constrained set of programs (a library). Although program inversion is by nature imperfect due to the many-to-one operation of instructions, the two GP search operators that employ this approach outperform the standard GP and other semantic-aware operators. Conceptually, this working principle can be applied in other domains and programming languages. Program inversion can be thus a feasible avenue for equipping the automatic programming algorithms, including GP algorithms, with the capability of problem decomposition.

Acknowledgments. Work supported by National Science Centre, T. Pawlak grant no. DEC-2012/07/N/ST6/03066 and K. Krawiec grant no. DEC-2011/01/B/ST6/07318.

References

[1] W. B. Langdon and R. Poli, "Why ants are hard" in *GP'98*, pp. 193–201, Morgan Kaufmann, 1998.
[2] J. Rosca and D. H. Ballard, "Causality in genetic programming" in *ICGA'95*, pp. 256–263, Morgan Kaufmann, 1995.
[3] K. Krawiec and B. Wieloch, "Analysis of semantic modularity for genetic programming" *Foundations of Computing and Decision Sciences*, vol. 34, no. 4, pp. 265–285, 2009.
[4] K. Krawiec, "On relationships between semantic diversity, complexity and modularity of programming tasks" in *GECCO'12*, 2012.
[5] K. Krawiec and J. Swan, "Pattern-guided genetic programming" in *GECCO'13*, ACM, 2013.
[6] K. Krawiec and T. Pawlak, "Approximating geometric crossover by semantic backpropagation" in *GECCO'13*, ACM, 2013.
[7] B. Wieloch and K. Krawiec, "Running programs backwards: Instruction inversion for effective search in semantic spaces" in *GECCO'13*, 2013.
[8] W. B. Langdon, "How many good programs are there? How long are they?" in *FOGA'02*, pp. 183–202, Morgan Kaufmann, 2002.
[9] N. F. McPhee, B. Ohs, and T. Hutchison, "Semantic building blocks in genetic programming" in *EuroGP'08*, pp. 134–145, 2008.
[10] L. Beadle and C. Johnson, "Semantically driven crossover in genetic programming" in *WCCI'08*, pp. 111–116, IEEE Press, 2008.
[11] D. Jackson, "Phenotypic diversity in initial genetic programming populations" in *EuroGP'10*, vol. 6021 of *LNCS*, pp. 98–109, Springer, 2010.
[12] E. Galvan-Lopez, B. Cody-Kenny, L. Trujillo, and A. Kattan, "Using semantics in the selection mechanism in genetic programming: a simple method for promoting semantic diversity" in *CEC'13*, 2013.
[13] N. Q. Uy, N. X. Hoai, M. O'Neill, R. I. McKay, and E. Galvan-Lopez, "Semantically-based crossover in genetic programming: application to real-valued symbolic regression" *Genetic Programming and Evolvable Machines*, vol. 12, pp. 91–119, June 2011.
[14] N. Q. Uy, N. X. Hoai, M. O'Neill, R. I. McKay, and D. N. Phong, "On the roles of semantic locality of crossover in genetic programming" *Information Sciences*, vol. 235, pp. 195–213, 20 June 2013.
[15] P. Day and A. K. Nandi, "Binary string fitness characterization and comparative partner selection in genetic programming" *IEEE Transactions on Evolutionary Computation*, vol. 12, pp. 724–735, Dec. 2008.
[16] A. Moraglio, K. Krawiec, and C. G. Johnson, "Geometric semantic genetic programming" in *PPSN'12*, vol. 7491 of *LNCS*, pp. 21–31, 2012.
[17] A. Moraglio and A. Mambrini, "Runtime analysis of mutation-based geometric semantic genetic programming for basis functions regression" in *GECCO'13*, pp. 989–996, ACM, 2013.
[18] K. Krawiec and T. Pawlak, "Locally geometric semantic crossover: a study on the roles of semantics and homology in recombination operators" *Genetic Programming and Evolvable Machines*, vol. 14, no. 1, pp. 31–63, 2013.
[19] T. Toffoli, "Reversible computing" in *ICALP*, pp. 632–644, 1980.
[20] W. B. Langdon, "The distribution of reversible functions is Normal" in *Genetic Programming Theory and Practice*, pp. 173–187, Kluwer, 2003.
[21] M. Oltean, "Evolving reversible circuits for the even-parity problem" in *Applications of Evolutionary Computing*, pp. 225–234, 2005.
[22] M. Oltean and C. Grosan, "Evolving digital circuits using multi expression programming" in *EH-2004*, pp. 87–97, IEEE Press, 2004.
[23] K. Krawiec and P. Lichocki, "Approximating geometric crossover in semantic space" in *GECCO'09*, pp. 987–994, ACM, 2009.
[24] A. Moraglio, "Abstract convex evolutionary search" in *FOGA'11*, pp. 151–162, ACM, 2011.
[25] K. Krawiec, "Medial crossovers for genetic programming" in *EuroGP'12*, vol. 7244 of *LNCS*, pp. 61–72, Springer Verlag, 2012.
[26] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.
[27] R. Poli and W. B. Langdon, "A new schema theory for genetic programming with one-point crossover and point mutation" in *GP'97*, pp. 278–285, Morgan Kaufmann, 1997.
[28] J. McDermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong, and U.-M. O'Reilly, "Genetic programming needs better benchmarks" in *GECCO'12*, pp. 791–798, ACM, 2012.
[29] J. A. Walker and J. F. Miller, "Investigating the performance of module acquisition in cartesian genetic programming" in *GECCO'05*, vol. 2, pp. 1649–1656, ACM, 2005.
[30] W. B. Langdon and R. Poli, *Foundations of Genetic Programming*. Springer-Verlag, 2002.
[31] I. Kushchu, "Genetic programming and evolutionary generalization" *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 5, pp. 431–442, 2002.
[32] G. Kanji, *100 Statistical Tests*. SAGE Publications, 1999.
[33] M. Hollander and D. Wolfe, *Nonparametric Statistical Methods*. Wiley, 1999.
[34] H. Cramér, *Mathematical Methods of Statistics*. Almqvist & Wiksells Akademiska Handböcker, Princeton University Press, 1946.
[35] Bentley, "Multidimensional binary search trees used for associative searching" *CACM: Communications of the ACM*, vol. 18, 1975.
[36] P. A. Whigham and G. Dick, "Implicitly controlling bloat in genetic programming" *IEEE Transactions on Evolutionary Computation*, vol. 14, no. 2, pp. 173–190, 2010.

Tomasz Pawlak received M.Sc. degree in Computer Science from Poznan University of Technology in 2011. He is a Ph.D. student at Poznan University of Technology, and works mainly on topics related to evolutionary computation, genetic programming and health informatics. His recent work focuses on semantics and program behavior in genetic programming, design of genetic operators and use of semantics in other parts of genetic programming algorithm. More details at www.cs.put.poznan.pl/tpawlak.

Bartosz Wieloch received B.Eng, M.Sc. and Ph.D. degrees in computing science from Poznan University of Technology, Poland, in 2004, 2006 and 2013, where he is currently an assistant professor in the Laboratory of Intelligent Decision Support Systems. His main research interests include semantics, problem decomposition and program behavior analysis in genetic programming and program synthesis. His work involves also pattern recognition in image analysis like object recognition or medical diagnosis.

Krzysztof Krawiec (M'06) received Ph.D. and Habilitation degrees from Poznan University of Technology, Poland, in 2000 and 2004, where he is currently an associate professor. His recent work includes: semantics and program behavior in genetic programming; coevolutionary algorithms and test-based problems; evolutionary computation for machine learning, primarily for learning game strategies and for synthesis of pattern recognition systems; and modeling of complex phenomena using genetic programming. Dr. Krawiec is the author of over 100 publications on the above and related topics, including *Evolutionary Synthesis of Pattern Recognition Systems* (2005), an associate editor of Genetic Programming and Evolvable Machines, and the president of the Polish Chapter of IEEE Computational Intelligence Society for the term 2013–2014. More details at www.cs.put.poznan.pl/kkrawiec.