Sequential Symbolic Regression with Genetic Programming

Luiz Otávio V.B. Oliveira, Fernando E.B. Otero, Gisele L. Pappa and Julio Albinati

Abstract This chapter describes the Sequential Symbolic Regression (SSR) method, a new strategy for function approximation in symbolic regression. The SSR method is inspired by the sequential covering strategy from machine learning, but instead of sequentially reducing the size of the problem being solved, it sequentially transforms the original problem into potentially simpler problems. This transformation is performed according to the semantic distances between the desired and obtained outputs and a geometric semantic operator. The rationale behind SSR is that, after generating a suboptimal function f via symbolic regression, the output errors can be approximated by another function in a subsequent iteration. The method was tested in eight polynomial functions, and compared with canonical genetic programming (GP) and geometric semantic genetic programming (SGP). Results showed that SSR significantly outperforms SGP and presents no statistical difference to GP. More importantly, they show the potential of the proposed strategy: an effective way of applying geometric semantic operators to combine different (partial) solutions, avoiding the exponential growth problem arising from the use of these operators.

Key words: symbolic regression, semantic genetic programming, geometric semantic crossover, problem transformation

Luiz Otávio V. B. Oliveira (✉)
DCC, Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
e-mail: luizvbo@dcc.ufmg.br

Fernando E. B. Otero
School of Computing, University of Kent, Chatham Maritime, UK
e-mail: F.E.B.Otero@kent.ac.uk

Gisele L. Pappa
DCC, Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
e-mail: glpappa@dcc.ufmg.br

Julio Albinati
DCC, Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
e-mail: jalbinati@dcc.ufmg.br

1 Introduction

Many researchers have been interested in exploring the regularities and modularities of the search space in order to improve the performance of Genetic Programming (GP) (Koza, 1992a, 1994) when dealing with complex problems. A popular approach is to allow GP to define modules, by either evolving specific code to be used as a module or identifying potentially useful code in existing individuals, in the hope that a module will capture regularities in the search space and ultimately decompose the original problem into small (more tractable) subproblems. While previous approaches have shown some degree of success, they rely on the idea that useful modules will emerge during the GP search and they are very much focused on the structure (syntax) of the individuals. There are potential drawbacks associated with these assumptions: there is no guarantee that modules are solving different parts of the problem, the quality of modules is determined indirectly by evaluating the individuals that use the modules and there is still a pressure on the GP to find the complete solution to the problem at once—i.e., both modules and the code that uses the modules are evolved at the same time.

Traditionally, GP search operators perform modifications to individuals' representation (syntax), with the aim that these modification will lead to changes on their behaviour (semantics). In other words, traditional GP search operators are *blind* operators regarding the semantics of an individual. In the same sense, syntactical approaches for modularisation are also *blind* regarding the definition of modules, since there is no guarantee that their behaviours are different—i.e., that they are solving different parts of the problem. Moraglio et al (2012) recently proposed geometric semantic search operators in the context of the Geometric Semantic Genetic Programming (GSGP), which can be used to search directly the semantic space of the problem. An interesting characteristic of GSGP is that the fitness landscape seen by the search operators is unimodal for problems consisting in finding the correct mapping for input-output pairs—the fitness is the distance of the output vector of a solution to the optimum. Therefore, these operators present a new opportunity to explore the modularity of the GP search.

The problem-solving procedure employed by GP algorithms can be seen as a supervised learning procedure: given $\{(c_1, o(c_1)), \ldots, (c_n, o(c_n))\}$ input-output pairs representing the training cases C, where each pair $(c_i, o(c_i))$ denotes an input value and its correspondent output value, respectively; the problem can be defined as finding a function $f : C \rightarrow O$ that maps each case c_i in C to its correspondent output $o(c_i)$ in O. Many supervised learning algorithms employ a strategy to decompose the original into subproblems, find solutions to these subproblems and use them to generate the solution for the original problem. For example, top-down decision tree induction employ a divide-and-conquer strategy, where at each decision (internal) node the training cases are divided based on a test outcome. Each subset of the training cases, representing a reduced problem, is push-down the tree and the procedure is repeated until a leaf node is generated. A similar strategy is used by many rule induction algorithms, where a sequential covering strategy is used to transform the problem of finding a list of classification rules into a sequence of smaller problems

of finding a good rule. After a rule is created, the training cases classified by the rule are removed, reducing the number of training cases for the next iteration of the procedure.

Given that GP is essentially a supervised learning method and geometric semantic operators enable the direct manipulation of the output vectors, *could we apply a heuristic to decompose the problem into smaller subproblems and use GP to solve them?* Otero and Johnson (2013) presented a strategy based on the sequential covering to decompose a boolean problem into smaller subproblems. Each subproblem is then solved by a traditional GP and the individual solutions are combined using a geometric semantic crossover. It uses a property of the geometric semantic crossover for the boolean domain: individuals are combined using a boolean mask, which acts as a selector to inform when a particular individual solution should be used. While this strategy is successful for boolean domains, there is not a straightforward way to adapt it to the real domain, since the operation of the geometric semantic crossover is different.

In this chapter we present a method to sequentially solve symbolic regression problems using a combination of geometric semantic operators and a heuristic inspired by the traditional sequential covering strategy. The proposed method, Sequential Symbolic Regression (SSR), works by sequentially transforming the original problem, according to the partial solutions generated, into potentially simpler ones. The rationale behind SSR is that, after generating a suboptimal function f via symbolic regression, the output errors can be approximated by another function, in a subsequent iteration. In order to transform the original output based on the output of function f, each iteration of SSR applies a transformation based on a geometric semantic crossover operator (Moraglio et al, 2012). This procedure allows the GP to focus, at each iteration, on different aspects (subproblems) of the original problem.

The remaining of the chapter is organised as follows. Section 2 reviews previous works exploring regularities and modularity in GP. Section 3 revises the properties of geometric semantic operators. The proposed strategy for sequential symbolic regression is presented in Section 4, followed by computational experiments in Section 5. Finally, Section 6 concludes the chapter and presents future research directions.

2 Modulatisation in Genetic Programming

Since the introduction of genetic programming (Koza, 1992a), researchers have been interested in exploring the regularities and modularity of the problem space. One of the main motivation is to identify these regularities to decompose the problem at hand into more tractable sub-problems: finding solutions to sub-problems should be easier than finding a solution to the original problem, and these sub-solutions can be used to create the solution to the whole problem. This process is illustrated in Figure 1. This is analogous to how human programmers usually tackle problems: instead of creating a single procedure to implement an entire program,

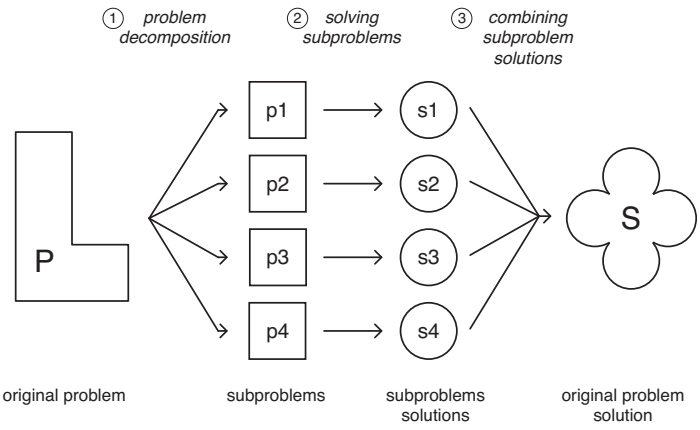


Fig. 1 The hierarchical problem-solving process: the original problem *P* is decomposed in a set of subproblems (step 1); the goal is then to solve each of the subproblems (step 2); finally, the solution *S* to the original problem *P* is created by using the solutions to the subproblems (step 3). Figure adapted from (Koza, 1992a).

they usually break down the implementation into several different procedures and the combination of these procedures compose the complete implementation.

The Automatic Defined Functions (ADFs) proposed by Koza (1992a,b, 1994) was one of the first ideas to address the automated problem decomposition. ADFs impose a syntactical structure to individuals: an individual genotype is divided into a *result-producing branch* and several *function-defining branches*. The motivation is that function definitions potentially exploit the regularities of the problem space and these definitions can be used from the result-producing branch. On the one hand, Koza argues that by allowing the definition and use of functions, the problem is decomposed into subproblems. On the other hand, the modular structure (syntax) of individuals is manually defined, therefore, the decomposition process is not autonomous—the number of ADFs and their parameters are controlled by user-defined values. Additionally, even if functions actually represent solutions to subproblems, they are being evolved at the same time as the complete solution. There is a pressure to solve all parts of the problem at once—the definition of the functions and the correct use of those functions.

A popular idea to explore problem space regularities focused on defining modules based on the genetic material of individuals. Several involved the random selection of subtrees to create modules: Koza (1992a) proposed the use of a subtree *encapsulation* operator, which consists of randomly selecting a subtree from an individual to create a terminal primitive that encapsulated the subtree; Angeline and Pollack (1992, 1994) proposed the Genetic Library Builder (GLiB) system, which employs mutation operators that randomly select subtrees to create modules (*compress* operator) that can be later expanded (*expand* operator); similar *compress* and

expand operators to create and expand modules were more recently proposed by Walker and Miller (2008) in the context of Embedded Cartesian Genetic Programming (ECGP), with the extension of the use of module-altering operators (*module point* mutation, *add-input*, *add-output*, *remove-input* and *remove-output* operators); Spector et al (2011a,b, 2012) proposed the use of the concept of 'tags' to label fragments of code that can be later reused by referencing the same label—while this is similar to the use of a *compress* operator, it provides the flexibility of partial name matches (a label will match the closest matching tag).

Other authors followed the idea of identifying useful building blocks (subtrees) to define modules: Rosca and Ballard (1994) proposed the use of heuristics to create new modules, selecting fit blocks (blocks with high fitness value) and frequent blocks (blocks that appear frequently in the population); similarly, Roberts et al (2001) accumulate the frequency information of multiple runs of a GP to create a *subtree database* and subsequent runs can use the most frequent subtrees encapsulated as terminal primitives.

There are also works that explore the idea of a library of modules created prior to the run of a GP. Keijzer et al (2004) introduced the use of Run Transferable Libraries (RTL). The RTL is created by running the GP on lower-order problem instances, considered as a training phase, and then used to solve more complex instances of the same problem. Similarly, Christensen and Oppacher (2007) generated small trees for the Santa Fe Trail problem to create a library of modules in a training phase, where the small trees are not necessarily complete solutions, and then is used this library to find the complete solution to the problem. Another approach that uses the idea of training a GP on smaller problem instances in order to generate modules was presented by Jackson and Gibbons (2007), where the authors proposed the use of *layered learning*. The first layer is used to solve a lower-order version of the original problem and the final solution at this layer is converted to a parameterised module. The second layer uses this module to search the solution of a higher-order version of the same problem. While the creation of a library of modules in a training phase or in different layers can provide a decomposition of the problem, it represents a single decomposition step and it is not automated—the user has to manually choose to use either a training phase or to generate small trees prior to the search for the complete solution.

Considering the initial goal of problem decomposition, the aforementioned approaches rely on the assumption that the modules created could represent solutions to subproblems. The main drawback of this assumption is that modules are defined based on their syntax—i.e., the creation/selection of the modules does not involve any evidence that the modules are solving different parts of the problem.[1] A common characteristic of these approaches is that they provide a mechanism to create/identify modules during the run of the GP and expect that good modules will emerge as a result of the search, but at the same time, they do not employ any control over whether the use of modules decomposes the problem into subproblems.

[1] The selection of building blocks based on fitness proposed by Rosca and Ballard (1994) is an exception to the syntax-oriented selection, although there is no guarantee that different modules are solving different parts of the problem.

Perhaps the emphasis in syntactical approaches to modularity is a result of the tendency of using syntactical search operators in GP—both crossover and mutation operators are blind search operators regarding their effect on the individual behaviour, only focusing on syntactical changes. Additionally, the pressure of solving all parts of the problem at once might reduce diversity and, in some cases, also prevent the convergence to the optimal solution (McKay, 2000).

Lee (1999) proposed an extension to GP to deal with forecasting of real world chaotic time series, which resembles the sequential strategy of the algorithm proposed in this chapter. Lee's assumption is that a time series is composed by deterministic and stochastic parts—subtracting the solution found by a run of the GP for the deterministic part from the original time series, the stochastic part is obtained as a residual time series. Repeating this process recursively to the sequence of residual time series, a set of (sub-)solutions can be created. These are then combined using numerical coefficients calculated by the least square method with respect to a predetermined region of the time series—the explicit definition of regions of the time series (regions of the search space) can be seen as a *manual* decomposition of the problem. As we will discuss in the following sections, our proposed algorithm does not rely on the definition of regions of the search space and the (sub-)solutions evolved are combined using the principle of a geometric semantic crossover to produce the solution to the original problem.

3 Geometric Semantic Operators

Standard genetic programming operators were originally conceived to operate in the *syntatic-level* of the solutions being evolved. Consider, for example, a subtree crossover. It will randomly select subtrees from two previously generated solutions and swap them, regardless of what the outputs of the selected subtrees are. When tree outputs are neglected, we ignore the fact that, at the end of the evolutionary process, what matters is the quality of the best solution found, which is indirectly defined by the output generated.

The semantics of an individual can be informally defined as the meaning of syntactically correct programs or functions (Uy et al, 2011)—in a GP context, the set of outputs produced by a program or function given a set of inputs. Many approaches have been previously used to represent and extract semantics from genetic programming (Vanneschi et al, 2014). This section is interested in one of these approaches: geometric semantic operators.

In order to design operators that directly impact the semantics of a solution, Moraglio et al (2012) defined the concept of *semantic distance* and *geometric semantic operators* for the real functions domain (e.g., symbolic regression), which are replicated in Definition 1 and Definitions 2 and 3, respectively.

Definition 1 *Let S be the set of solutions and $s_1, s_2 \in S$. A function $SD : S \times S \to \mathbb{R}$ is said to be a semantic distance function if $SD(s_1, s_2) = D(O(s_1), O(s_2))$, where $O(s)$ returns the output vector of s and D is a distance function.*

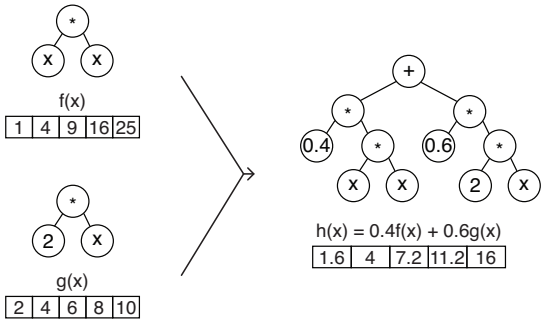


Fig. 2 Example of geometric semantic crossover operator between $f(x) = x^2$ and $g(x) = 2x$ using $r = 0.4$.

Definition 2 *Let S be the set of solutions, $XO : S \times S \to S$ be a crossover operator and SD be a semantic distance function. XO is said to be geometric with relation to SD if, for all $s_1, s_2, s_3 \in S$ such that $s_3 = XO(s_1, s_2)$, $SD(s_1, s_2) = SD(s_1, s_3) + SD(s_3, s_2)$.*

Definition 3 *Let S be the set of solutions, $MT : S \to S$ be a mutation operator and SD be a semantic distance function. MT is said to be ε-geometric with relation to SD if, for all $s_1, s_2 \in S$ such that $s_2 = MT(s_1)$, $\boldsymbol{E}[SD(s_1, s_2)] \leq \varepsilon$, where $\boldsymbol{E}$ denotes the expected value.*

Definitions 2 and 3 show that semantic geometric operators generate solutions in a much more controlled fashion. Particularly, the semantics of a solution generated through a geometric semantic crossover is guaranteed to be somewhere between the semantics of its parents. This fact implies in an interesting property: an offspring will never be worse than the worst of its parents. Similarly, an ε-geometric semantic mutation will generate solutions that are, on average, not worse than the original solution by more than ε.

Moraglio et al (2012) also proposed specific semantic geometric operators for regression problems. The crossover operator proposed is essentially a convex combination of functions. Let S be the set of solutions, $s_1, s_2 \in S$, $XO(s_1, s_2) = r.s_1 + (1 - r).s_2$, where r is a random real number in the interval $[0, 1]$. The mutation operator was defined as $MT(s) = s + ms.(TR_1 - TR_2)$, where $s \in S$, ms is a real number and TR_1, TR_2 are randomly generated trees. The authors show that these operators are geometric with relation to the semantic distance function $SD(s_1, s_2) = \sum_{x_i \in T} [O(s_1)(x_i) - O(s_2)(x_i)]^2$, where T is a set of training examples.

Figures 2 and 3 show examples of geometric semantic operators for the real functions domain. Observe that in Figure 2, each element of the output vector of the offspring is a convex combination of elements from the parents' output vectors using coefficients 0.4 and 0.6. In Figure 3, we notice how the impact of the geometric semantic mutation operator can be controlled by setting appropriate values for *ms*.

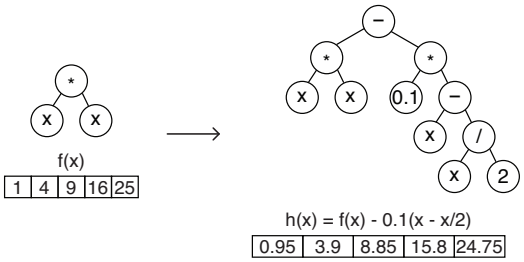


Fig. 3 Example of geometric semantic mutation operator of $f(x) = x^2$ using $ms = 0.1$, $TR_1(x) = x$ and $TR_2(x) = x/2$.

4 Sequential Symbolic Regression

This section introduces Sequential Symbolic Regression (SSR), a method that sequentially executes a standard GP for symbolic regression and indirectly considers the semantic of the solutions being created. SSR is inspired by a sequential covering strategy, similar to the one employed by Otero and Johnson (2013), where at each iteration a solution to a transformed (and potentially simpler) problem is evolved.

The main difference between SSR and a traditional sequential covering method is in the transformation step that occurs at each iteration. In a traditional sequential covering strategy, the problem is reduced at each iteration—i.e., the training cases covered by the iteration solution are removed, effectively reducing the problem to the subsequent iterations. Since SSR deals with problems in the real-valued domain, the concept of *covered* training cases is not directly applicable.[2] Instead of removing training cases, at each iteration of SSR, the output values of the original problem are modified based on the use of a geometric semantic crossover and the iteration solution output—the transformation of the problem is based on the semantic of the solution created on the iteration. We hypothesise that the use of the iterative (sequential) solution construction procedure allows the GP to focus on different aspects (subproblems) of the original problem, creating individual solutions that are combined by a geometric semantic crossover.

A typical symbolic regression problem can be defined as follows. Given a set of input-output pairs $C = \{(c_1, o(c_1)), ..., (c_n, o(c_n))\}$ representing the training cases, where each pair $(c_i, o(c_i))$ denotes an input value and its correspondent output value, respectively; a symbolic regression problem can be defined as finding a function $f : C \rightarrow O$ that minimizes an error metric, such as the mean squared error (MSE), the mean absolute error (MAE) or the root mean squared error (RMSE).

[2] It is unlikely that a solution will reach (near) zero error only for a subset of the points (training cases), unless it is the optimal solution, which in this case it will reach a (near) zero error for all points.

The metrics described above use the summation of the squared or absolute residuals—the difference between the current output and the function output—to compute the error function. Hence, when the absolute value of residuals is minimized, so is the measured error. A residual $e(c_i)$ corresponds to the error in the fitting of the function to the i-th observation, and is defined as

$$e(c_i) = o(c_i) - \hat{o}(c_i) = o(c_i) - f(c_i) \ . \tag{1}$$

The optimal solution to a regression problem is a function f^*, such that $e(c_i) = o(c_i) - f^*(c_i) = 0$ for $i = 1,2,...,n$, and often a function f found by a regression method is an approximation of f^*, not reaching a zero error or the minimum error defined according to the problem.

The rationale behind the sequential procedure of SSR is that, after generating a suboptimal function f, the residual can be approximated by another function, in a subsequent iteration. In order to transform the original output based on the output of function f, each iteration of SSR applies a transformation based on a geometric semantic crossover operator (Moraglio et al, 2012). The geometric semantic crossover operator for the real-value domain combines the output of two known functions f and f' to generate a new function f^*, with an a priori unknown output. The principle used in SSR is that the output of function f and the output of the function f^* are known, and therefore, they can be used to define the transformation required to determine the desired output of function f' based on the residual of function f. The definition of the geometric semantic crossover is given by

$$f^*(c_i) = r \cdot f(c_i) + (1 - r) \cdot f'(c_i) \ , \tag{2}$$

where r is a random real constant in the range $[0,1)$. Substituting the definition of function f^* to the residual equation, we obtain

$$e(c_i) = o(c_i) - [r \cdot f(c_i) + (1 - r) \cdot f'(c_i)] \ . \tag{3}$$

Using Equation (3) and given that f is the function created by an iteration of SSR, the output $o'(c_i)$ for function $f'(c_i)$ that reduces the residual error e to zero is computed as

$$o'(c_i) = \frac{o(c_i) - r \cdot f(c_i)}{1 - r} \ . \tag{4}$$

The transformed output vector o' defines a new regression problem, where the goal is to find a function f' that minimizes the new residuals $e'(c_i) = o'(c_i) - f'(c_i)$, which is the definition of problem for the next iteration.

Another way to see the strategy employed in SSR is to look at the use of the transformation step: a solution is built starting from the desired output, the output of the original problem; if the function (individual) f created at an iteration of SSR does not minimises the error e to zero, a geometric semantic crossover is used to transform the original problem. Given that we know the desired output—the output of the individual generated by the crossover operation—and one of the individuals

Algorithm 1 Sequential Symbolic Regression procedure

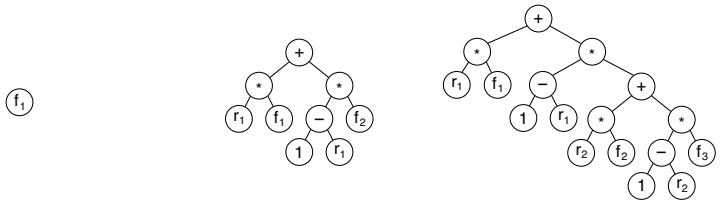
```
input: training points (C), stopping criteria, GP parameters
input ← (c_1, c_2, ..., c_n), for c_k ∈ C
output ← (o(c_1), o(c_2), ..., o(c_n)), for o(c_k) ∈ C
/* Solution iteratively constructed */
S ← ∅
while stopping criteria not reached do
    f ← RunGP(input, output)
    if (MSE(f, output) ≤ 0.01) then
        S ← AddFunction(f)
    else
        r ← random()
        S ← AddFunction(f, r)
        output← AdjustOutputs(f, r, output)
        return
    end if
end while
return S
```

of the crossover, we can determine the required output of a second individual that complements the crossover.

Therefore, instead of combining individuals at random as in the Semantic GP, SSR optimises the effect of the geometric semantic crossover operator by searching for the individual that represents the best match (minimises the error) given the desired output vector. At the same time, it indirectly mitigates the problem of exponential growth of individuals observed in SGP (Moraglio et al, 2012; Vanneschi et al, 2013), since the solution is created sequentially, without requiring that all individual solutions are kept in memory, and there is only one solution being created using the geometric semantic operator, requiring a single simplification step at the end of SSR if the size of the complete solution needs to be reduced.

4.1 SSR procedure

Algorithm 1 presents the high-level pseudocode of the SSR procedure. It starts with an empty solution tree S, which is iteratively incremented by carrying out sequential regressions using a traditional GP algorithm. At the k-th iteration, a new function f_k is generated by the GP (*RunGP* procedure). If function f_k corresponds to the optimal solution—i.e., the output of f_k is such that $MSE(f_k, output) \leq 0.01$—$f_k$ is added to the solution tree S and the sequential procedure stops. Otherwise, f_k is added to the solution tree S using a geometric semantic crossover with a random constant r_k in the range $[0, 1)$. Note that at this point the crossover operation is incomplete—i.e., only one of the parent individuals is known. Then, the constant r_k and the function f_k are used to modify the desired *output* using transformation represented by Equation (4). The iterative transformation step is given by



(a) First iteration. (b) Second iteration. (c) Third iteration.

Fig. 4 Illustration of the solution tree S and the corresponding expression at different iterations: (a) $S = f_1$; (b) $S = r_1 \cdot f_1 + (1 - r_1) \cdot f_2$; (c) $S = r_1 \cdot f_1 + (1 - r_1) \cdot [r_2 \cdot f_2 + (1 - r_2) \cdot f_3]$.

$$o_{k+1}(c_i) = \frac{o_k(c_i) - r_k \cdot f_k(c_i)}{1 - r_k} \ , \tag{5}$$

for $k = 1, 2, ..., n$, where n is the maximum number of iterations. The sequential SSR process continues until a minimum error or a maximum number of iterations is reached. Figure 4 illustrate the sequential solution construction, showing the solution tree S at different iterations of the procedure.

Next, we present an illustrative example of how SSR works. Let us consider we want to find a function whose values match those in a set of training input cases $C = \{(1,1),(3,4),(5,9)\}$, i.e., $input = (1,3,5)$ and $output_1 = (1,4,9)$. Let us assume the first GP regression generates a function f_1 that produces the output vector $(1,3.5,8)$, and the absolute residual vector $(0,0.5,1)$. A constant $r_1 = 0.4$ is generated randomly and stored in f_1. From there, the new target output vector is calculated (Equation 4), and is equal to $(1,4.33,9.67)$. The process continues until $MSE \leq 0.01$, as shown in Table 1. The column $output_k$ represents the target output points the regression needs to generate (when $k = 1$, they represent the original problem output), followed by the generated output ($f_k(c_i)$) and the residual generated by f_k ($|e'_k(c_i)|$) and the overall MSE.

Table 1 Example of SSR execution. The First column presents the current iteration, followed by the values of r_k, the desired outputs $output_k$ (3 columns), the evolved outputs f_k (3 columns), the absolute residuals of f_k regarding $output_k$ (3 columns) and MSE (last column).

		$output_k$			$f_k(c_i)$			$\lvert e'_k(c_i)\rvert = \lvert o_k(c_i) - f_k(c_i)\rvert$			
k	r_k	$o_k(c_1)$	$o_k(c_2)$	$o_k(c_3)$	c_1	c_2	c_3	$c_1, o_k(c_1)$	$c_2, o_k(c_2)$	$c_3, o_k(c_3)$	MSE
1	0.4	1.00	4.00	9.00	1.00	3.50	8.00	0.00	0.50	1.00	0.417
2	0.5	1.00	4.33	9.67	1.00	4.00	9.00	0.00	0.33	0.67	0.067
3	0.3	1.00	4.67	10.33	2.00	4.50	11.00	1.00	0.17	0.67	0.044
4	0.2	0.57	4.74	10.05	0.50	5.00	10.50	0.07	0.26	0.45	0.004

5 Experiments

This section presents experimental results performed to test SSR. All tests are compared with the semantic GP (SGP) proposed in Moraglio et al (2012) and a canonical GP (Koza, 1994) in a set of polynomial regression problems. Given that one of the main characteristics of the method is to use the geometric semantic crossover to combine solutions sequentially discovered to solve the problem, we use the same testbed as Moraglio et al (2012), composed by 8 univariate polynomials functions of degrees from 3 to 10, with real-valued coefficients uniformly drawn from $[-1,1]$.

In order to make the comparisons fair, all algorithms were given an execution budget of 100,000 evaluations, and the parameters used in each algorithm are detailed in Table 2. Note that, as SSR evolves a GP for k iterations, the sizes of populations vary across different algorithms, always respecting the evaluation budget. Because of that, different tournament sizes were used in order to balance selective pressure considering different population sizes. Notice that results of two versions of SSR and SGP are reported. In the case of SSR, the variation tests the trade-off between the number of generations of the canonical GP and the number of iterations of SSR.

For SGP, we used the same parameters reported in Moraglio et al (2012), but varied the method used for population initialization. The first algorithm configuration (SGP1) initializes with polynomials of degree 10 (same procedure used in Moraglio et al (2012)), while the initial population of SGP2 is randomly generated. One may argue that the assumption that we know the structure of the function we are looking for makes the use of symbolic regression unnecessary, which is true. However, the way geometric semantic crossover works depends heavily on the individuals in the initial population. If the genetic material we start with is not enough to produce the target function, mutation will probably not be able to insert enough modifications to the population to change this situation.

The experiments were performed in two phases. First, we run the methods in a training set with 20 points. Then, we used the function discovered in the first phase in a second set of 20 points. The points were uniformly drawn from the $[0,1]$ interval. All methods were executed 30 times. Table 3 shows the mean square error

Table 2 Parameter values for the methods used in the experiments.

Parameter	GP	SSR1	SSR2	SGP1	SGP2
Crossover probability	0.9	0.9	0.9	1	1
Mutation Probability	0.1	0.1	0.1	1	1
Tournament Size	7	3	3	5	5
Population Size	1000	100	100	20	20
Number of generations	100	50	100	5000	5000
Number of iterations	-	20	10	-	-
Initialization	-	-	-	YES	NO

Table 3 Average MSE (*average* [*standard error*]) for each algorithm in the training set, calculated over 30 runs.

Problem	GP	SSR1	SSR2	SGP1	SGP2
polynomial3	0.000 [0.000]	0.000 [0.000]	0.000 [0.000]	0.000 [0.000]	0.009 [0.002]
polynomial4	0.000 [0.001]	0.000 [0.000]	0.000 [0.000]	0.000 [0.000]	0.009 [0.002]
polynomial5	0.001 [0.003]	0.000 [0.000]	0.000 [0.001]	0.000 [0.000]	0.013 [0.004]
polynomial6	0.001 [0.001]	0.000 [0.001]	0.000 [0.000]	0.000 [0.000]	0.010 [0.003]
polynomial7	0.002 [0.001]	0.001 [0.002]	0.000 [0.000]	0.000 [0.000]	0.008 [0.002]
polynomial8	0.002 [0.002]	0.000 [0.000]	0.000 [0.000]	0.000 [0.000]	0.009 [0.002]
polynomial9	0.005 [0.004]	0.001 [0.001]	0.001 [0.003]	0.000 [0.000]	0.010 [0.002]
polynomial10	0.002 [0.003]	0.001 [0.001]	0.001 [0.002]	0.000 [0.000]	0.010 [0.002]

Table 4 Pairwise Nemenyi test for MSE in the training set. The symbol ▲ indicates the method in the column is statistically better than the one in the row.

	SSR1	SSR2	SGP1	SGP2
SSR2	-	-	-	-
SGP1	-	-	-	-
SGP2	▲	▲	▲	-
GP	-	-	-	-

(MSE) and standard deviation obtained by the three methods using different configurations.

Results are compared using a two-step approach. First, we apply Friedman's test with the null hypothesis $H_0: \ \theta_1 = \theta_2 = ...\theta_5$, where θ_i represents the MSE of one of the algorithms tested. If H_0 is rejected we apply Nemenyi test (Demšar, 2006) as a post-hoc procedure and make pairwise comparisons between the MSEs. Table 4 shows the results of the comparisons. The symbol ▲ indicates that the method in the column is statistically better than the method indicated in the row.

The results show that there is no evidence for statistical difference among the two versions of SSR. However there is statistical difference among the SGP versions, with SGP1 performing statistically better than SGP2. Concerning SSR, there is no evidence of statistical difference regarding the GP or SGP1 and the results are statistically better than those obtained by SGP2. In summary, the results of the proposed approach are as good as the results of the GP and SGP1 and better than the results of SGP2.

Figure 5 shows the results of MSE for different iterations of SSR for the 8 functions tested using 50 and 100 generations over 20 iterations. The behaviour of the method is the expected one: as iterations go on, the error is reduced. As observed, in most cases the error converges as we approach 10 iterations. Hence, we can say that a different stopping criteria—such as convergence—could significantly reduce the

Table 5 Average MSE (*average* [*standard error*]) for each algorithm in the test set, calculated over 30 runs.

Problem	GP	SSR1	SSR2	SGP1	SGP2
polynomial3	0.001 [0.001]	0.001 [0.004]	0.000 [0.001]	4.9e8 [2.6e9]	891.7 [2989.9]
polynomial4	0.001 [0.002]	0.000 [0.000]	0.000 [0.001]	84.33 [231.6]	5.360 [13.186]
polynomial5	0.007 [0.020]	0.001 [0.001]	0.001 [0.002]	8.158 [15.36]	7.158 [17.318]
polynomial6	0.008 [0.011]	0.003 [0.007]	0.002 [0.003]	1.2e5 [6.6e5]	9.350 [16.763]
polynomial7	0.009 [0.034]	0.001 [0.002]	0.001 [0.001]	41.27 [83.21]	6.005 [11.144]
polynomial8	0.004 [0.003]	0.001 [0.001]	0.001 [0.001]	117.0 [350.3]	13.497 [49.12]
polynomial9	0.014 [0.020]	0.006 [0.008]	0.003 [0.004]	43.66 [223.2]	2.811 [2.682]
polynomial10	0.032 [0.027]	0.013 [0.012]	0.011 [0.015]	58.64 [230.4]	3.574 [4.479]

Table 6 Pairwise Nemenyi test for MSE in the test set. The symbol ▲ indicates the method in the column is statistically better than the one in the row.

	SSR1	SSR2	SGP1	SGP2
SSR2	-	-	-	-
SGP1	▲	▲	-	-
SGP2	▲	▲	-	-
GP	-	-	-	-

number of evaluations required to obtain the reported results (note that we did not halt the algorithm and always allowed it to run for the maximum evaluation budget). A different parameter setting, where the GP run for less generations at each iteration of the sequential procedure, combined with an effective stopping criteria might reduce significantly the fitness budget, making the use of SSR preferable over a single GP—these parameters can be tuned according to the problem at hand.

Table 5 presents the results of generalisation of the functions evolved in the training set and Table 6 the results for the Nemenyi test. The results show again that GP and SSR present no evidence of statistical difference. However, in this case, the results obtained by SSR are better than both versions of SGP. Looking at the values of MSE, we observe that SGP does not generalize well and has a tendency for overfitting. Therefore, these results show that SSR was successful in reducing the error of the symbolic regression problems and, at the same time, produced solutions with good generalisation power.

Regarding the comparisons with SGP, recall that the semantic operator has completely different roles in the algorithms. For SGP, experiments showed data overfitting (poor generalisation) can be a problem. Overfitting may be caused by the restrictions imposed by the geometric crossover, which combined with a semantic mutation designed to produce little semantic impact, makes SGP success heavily dependent on the initial population. This fact, combined with a small population size,

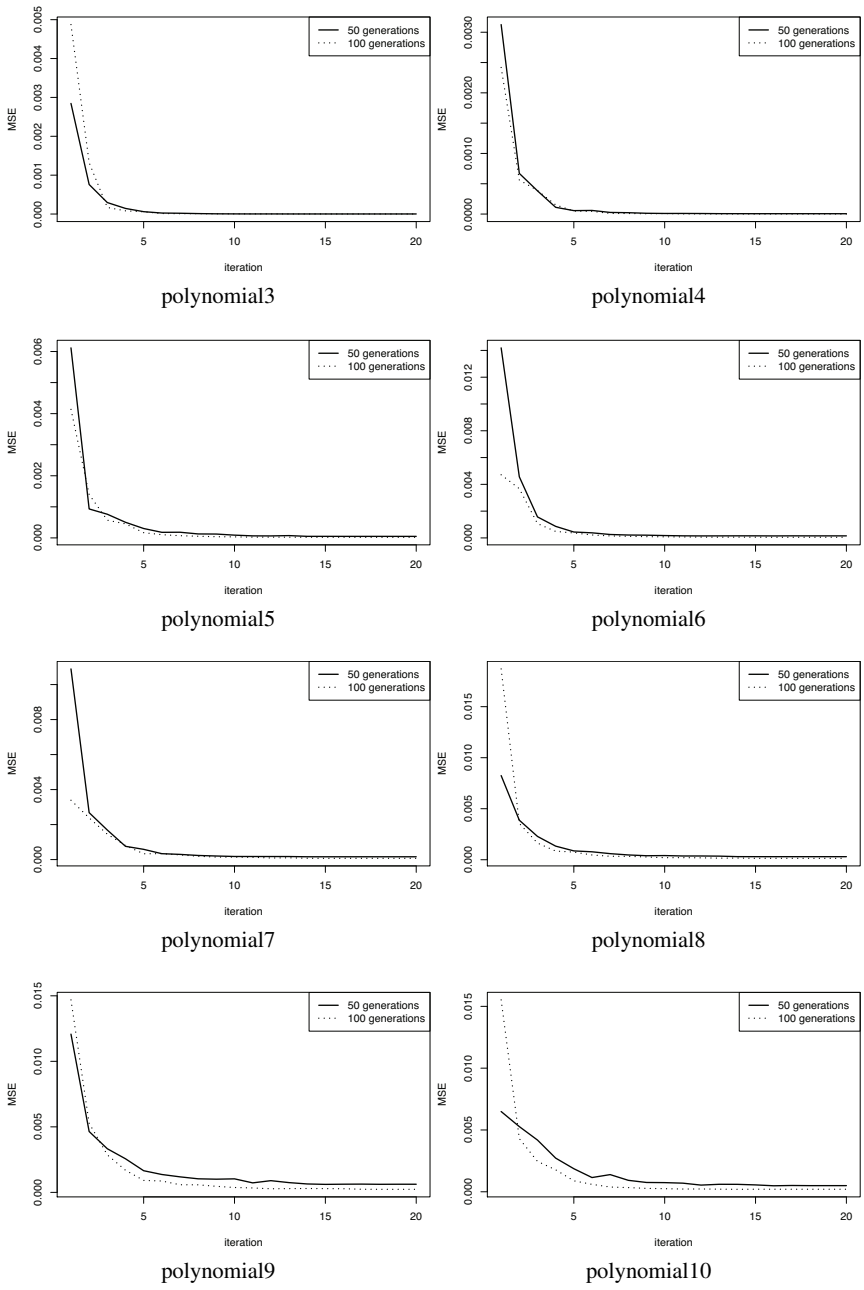


Fig. 5 Evolution of the error during iterations for both configurations of SSR for each problem, computed using the median of 30 runs.

Table 7 Number of nodes (*average* [*standard error*]) of the resulting function for each algorithm, calculated over 30 runs.

Problem	GP	SSR1	SSR2	SGP1	SGP2
polynomial3	50.5 [21.0]	1677.9 [242.5]	637.1 [165.3]	2.3e9 [1.2e9]	2.0e9 [1.3e9]
polynomial4	59.7 [28.0]	1720.7 [228.5]	635.2 [164.7]	2.0e9 [1.2e9]	2.3e9 [1.2e9]
polynomial5	68.1 [24.2]	1745.9 [195.9]	729.1 [133.8]	2.2e9 [1.4e9]	2.1e9 [1.2e9]
polynomial6	60.8 [26.9]	1664.9 [257.5]	691.6 [134.0]	1.9e9 [1.1e9]	2.0e9 [1.4e9]
polynomial7	63.2 [21.2]	1752.0 [170.8]	767.0 [140.4]	2.2e9 [1.1e9]	2.0e9 [1.2e9]
polynomial8	57.8 [28.8]	1644.1 [220.0]	712.1 [164.5]	2.1e9 [1.3e9]	2.2e9 [1.4e9]
polynomial9	49.4 [22.1]	1736.7 [197.5]	771.9 [154.9]	1.9e9 [1.3e9]	2.4e9 [1.1e9]
polynomial10	62.6 [24.2]	1786.6 [170.1]	784.3 [142.0]	2.0e9 [1.3e9]	2.0e9 [1.3e9]

can make it difficult for SGP to find a good solution. Even if such solution is found, it will usually be much more complex than those produced by SSR, also potentially leading to overfitting—something that has been observed when analysing the size of the evolved solutions.

Table 7 presents the average number of nodes and standard deviation of the final solutions found by each algorithm. The size of SSR1 and SSR2 solutions reflect approximately the number of GP executions within the algorithm, i.e. it is 20 and 10 times the number of nodes of the solutions generated by the canonical GP, respectively. The size of the functions generated by both SGP versions, on the other hand, are at least 10^6 times greater than the other methods, since the size of SGP individuals grows exponentially in the number of generations. Note that while SSR performs as many semantic crossovers as iterations, for SGP this number depends in the number of individuals, crossover probability and number of generations. The difference in size of the solutions found by SSR1 and SSR2 can be explained by the number of iterations of the sequential procedure: while SSR1 has a total of 20, SSR2 has a total of 10 (Table 2). This illustrates the impact of the number of crossover operations—iterations of the sequential procedure in the case of SSR—on the size of the solutions. At the same time, we don't see a big impact on the performance of the SSR algorithm, since the error is minimised after 10 iterations in most cases—as illustrated in Figure 5.

6 Conclusions and Future Work

This chapter proposed Sequential the Symbolic Regression (SSR), a new strategy to perform symbolic regression by iteratively learning solutions from a transformed set of problems. The definition of the problem changes according to the semantic distance (or error rate) generated from the desired and obtained outputs, and different (sub-)problem solutions are put together using a geometric semantic crossover

operator. The use of the semantic operator guarantees the solutions generated are never worse than the weakest of their parents.

Experiments were run in a set of eight polynomial functions and results compared with a canonical GP and a geometric semantic GP (SGP). When compared with SGP, which has a problem of exponential growth of its individuals, SSR has the advantage of generating smaller solutions that are less prone to overfitting. Regarding the canonical GP, the method has the potential of improving solutions even when the algorithm has already converged, by transforming the original problem into a new one. The results showed SSR presents MSE values statistically better than those generated by the solutions evolved by SGP, specially when a test set of points is used to evaluate the generalisation of the method. When compared with GP, there is no evidence of statistical difference among the results. However, we believe the results can still be improved to use a minimal computational budget (fitness evaluations).

As future work, a more complete study of the impact of the parameters in SSR needs to be performed, specially investigating what is the impact of running the GP for longer or SSR for more iterations. The method also needs to be validated in more complex symbolic regression problems, such as those suggested as GP benchmarks (White et al, 2013). Finally, other methods for combining different solutions are worth further investigation.

References

Angeline PJ, Pollack JB (1992) Evolutionary induction of subroutines. In: Proceedings of the 14th Annual Cognitive Science Conference, pp 236–241

Angeline PJ, Pollack JB (1994) Coevolving high-level representations. In: Langton C (ed) Artificial Life III, Addison-Wesley, Reading, MA, pp 55–71

Christensen S, Oppacher F (2007) Solving the artificial ant on the santa fe trail problem in 20,696 fitness evaluations. In: GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, London, vol 2, pp 1574–1579

Demšar J (2006) Statistical comparisons of classifiers over multiple data sets. The Journal of Machine Learning Research 7:1–30

Jackson D, Gibbons AP (2007) Layered learning in Boolean GP problems. In: Proceedings of the 10th European Conference on Genetic Programming, Springer, Valencia, Spain, Lecture Notes in Computer Science, vol 4445, pp 148–159

Keijzer M, Ryan C, Cattolico M (2004) Run transferable libraries – learning functional bias in problem domains. In: Genetic and Evolutionary Computation – GECCO-2004, Part II, Springer-Verlag, Seattle, WA, USA, Lecture Notes in Computer Science, vol 3103, pp 531–542

Koza JR (1992a) Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA

Koza JR (1992b) Hierarchical automatic function definition in genetic programming. In: Whitley LD (ed) Foundations of Genetic Algorithms 2, Morgan Kaufmann, Vail, Colorado, USA, pp 297–318

Koza JR (1994) Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge Massachusetts

Lee GY (1999) Genetic Recursive Regression for Modeling and Forecasting Real-World Chaotic Time Series. In: Advances in Genetic Programming III, MIT Press, chap 17, pp 401–423

McKay B (2000) Partial functions in fitness-shared genetic programming. In: Proceedings of the 2000 Congress on Evolutionary Computation CEC00, IEEE Press, La Jolla Marriott Hotel La Jolla, California, USA, vol 1, pp 349–356

Moraglio A, Krawiec K, Johnson C (2012) Geometric Semantic Genetic Programming. In: Parallel Problem Solving from Nature - PPSN XII, Springer Berlin Heidelberg, Lecture Notes in Computer Science, vol 7491, pp 21–31

Otero FEB, Johnson CG (2013) Automated problem decomposition for the boolean domain with genetic programming. In: Proceedings of the 16th European Conference on Genetic Programming, EuroGP 2013, Vienna, Austria, pp 169–180

Roberts SC, Howard D, Koza JR (2001) Evolving modules in genetic programming by subtree encapsulation. In: Genetic Programming, Proceedings of EuroGP'2001, Springer-Verlag, Lake Como, Italy, LNCS, vol 2038, pp 160–175

Rosca JP, Ballard DH (1994) Learning by adapting representations in genetic programming. In: Proceedings of the 1994 IEEE World Congress on Computational Intelligence, IEEE Press, Orlando, Florida, USA, vol 1, pp 407–412

Spector L, Harrington K, Martin B, Helmuth T (2011a) What's in an evolved name? the evolution of modularity via tag-based reference. In: Genetic Programming Theory and Practice IX, Genetic and Evolutionary Computation, Springer, Ann Arbor, USA, chap 1, pp 1–16

Spector L, Martin B, Harrington K, Helmuth T (2011b) Tag-based modules in genetic programming. In: GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation, ACM, Dublin, Ireland, pp 1419–1426

Spector L, Harrington K, Helmuth T (2012) Tag-based modularity in tree-based genetic programming. In: GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference, ACM, Philadelphia, Pennsylvania, USA, pp 815–822

Uy NQ, Hoai NX, ONeill M, McKay RI, Galván-López E (2011) Semantically-based crossover in genetic programming: application to real-valued symbolic regression. Genetic Programming and Evolvable Machines 12(2):91–119

Vanneschi L, Castelli M, Manzoni L, Silva S (2013) 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, Vienna, Austria, vol 7831, pp 205–216

Vanneschi L, Castelli M, Silva S (2014) A survey of semantic methods in genetic programming. Genetic Programming and Evolvable Machines 15(2):1–20

Walker JA, Miller JF (2008) The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. IEEE Trans Evolutionary Computation 12(4):397–417

White D, McDermott J, Castelli M, Manzoni L, Goldman B, Kronberger G, Jakowski W, OReilly UM, Luke S (2013) Better gp benchmarks: community survey results and proposals. Genetic Programming and Evolvable Machines 14(1):3–29

Index