A C++ framework for geometric semantic genetic programming

Article *in* Genetic Programming and Evolvable Machines · March 2014

DOI: 10.1007/s10710-014-9218-0

3 authors:

Mauro Castelli
New University of Lisbon
71 PUBLICATIONS **593** CITATIONS
SEE PROFILE

Sara Silva
University of Lisbon
84 PUBLICATIONS **1,305** CITATIONS
SEE PROFILE

Leonardo Vanneschi
New University of Lisbon
174 PUBLICATIONS **2,040** CITATIONS
SEE PROFILE

Genet Program Evolvable Mach
DOI 10.1007/s10710-014-9218-0

LETTER

A C++ framework for geometric semantic genetic programming

Mauro Castelli · Sara Silva · Leonardo Vanneschi

Received: 25 September 2013 / Revised: 24 February 2014

Abstract Geometric semantic operators are new and promising genetic operators for genetic programming. They have the property of inducing a unimodal error surface for any supervised learning problem, i.e., any problem consisting in finding the match between a set of input data and known target values (like regression and classification). Thanks to an efficient implementation of these operators, it was possible to apply them to a set of real-life problems, obtaining very encouraging results. We have now made this implementation publicly available as open source software, and here we describe how to use it. We also reveal details of the implementation and perform an investigation of its efficiency in terms of running time and memory occupation, both theoretically and experimentally. The source code and documentation are available for download at http://gsgp.sourceforge.net.

Keywords Genetic programming · Semantics · Geometric operators · C++

M. Castelli (✉) · L. Vanneschi
ISEGI, Universidade Nova de Lisboa, 1070-312 Lisbon, Portugal
e-mail: mcastelli@isegi.unl.pt

L. Vanneschi
e-mail: lvanneschi@isegi.unl.pt

M. Castelli · S. Silva · L. Vanneschi
INESC-ID, IST, Universidade de Lisboa, 1000-029 Lisbon, Portugal

S. Silva
LabMAg, FCUL, Universidade de Lisboa, 1749-016 Lisbon, Portugal
e-mail: sara@fc.ul.pt

S. Silva
CISUC, Universidade de Coimbra, 3030-290 Coimbra, Portugal

Published online: 02 April 2014

1 Introduction

Genetic programming (GP) [11] is one of the youngest paradigms inside the computational intelligence research area called Evolutionary Computation. The standard GP algorithm evolves a population of computer programs using genetic operators, usually crossover and mutation [7]. These operators, in their original definition, produce offspring by manipulating the *syntax* of the parents. In the last few years, researchers have dedicated several efforts to the definition of GP methods based on the *semantics* of the solutions, where by semantics we generally intend the behaviour of a program once it is executed, or more particularly the set of its output values on input training data [8, 10, 14]. In particular, new genetic operators, called geometric semantic operators, have been recently proposed in [10]. These operators have the interesting property of inducing a unimodal error surface on any supervised learning problem, i.e., any problem consisting in finding a function matching a set of input data into a set of known output values (like for instance regression and classification). As a consequence, when using geometric semantic operators, GP is expected to have a good evolvability on all these problems [8], independently of how complex they are (for instance in terms of number of features or instances). Nevertheless, as stated by Moraglio et al. [10], these operators have a serious limitation: by construction, they always produce offspring that are much larger than their parents, and this makes the size of the individuals in the population grow "very rapidly" with generations. More specifically, crossover approximately doubles the size of a parent, whereas for mutation the increase is approximately the size of two random trees (see Sect. 2 for details). As a consequence, after a few generations the population is composed by such big programs that the computational cost of evaluating their fitness is unmanageable. This limitation makes these operators impossible to use in practice, in particular on complex real-life applications.

We propose an implementation of geometric semantic genetic operators that makes them usable in practice, and does so very efficiently. A preliminary discussion of this implementation has already been presented in [13]. With a prototype of this system, we have been able to exploit the great potentialities of the geometric semantic operators on complex real-life problems. For instance, we have already applied them to a set of real-life applications including problems in pharmacokinetics [13, 14] and pharmacogenetics [4], energy load forecasting [14], construction industry [6], and forest sciences, both on regression [12] and classification [5] tasks. While the work described in [13] presents details about an implementation prototype, here we show how to use a publicly available implementation of geometric semantic GP. Furthermore, we reveal details of the implementation that were not explicit in [13], and we perform a deeper investigation of its efficiency, in terms of running time and memory occupation, both theoretically and experimentally.

By making the implementation publicly available, we are offering the GP community the opportunity to address other problems, in different domains, using this new tool, that from now on we will simply address as geometric semantic genetic programming (GSGP). By making it open source, we are also inviting other researchers to contribute to its improvement and extension. GSGP is licensed under

the terms of the GNU Affero General Public License version 3 (AGPLv3). The source code and documentation are available for download from the SourceForge repository at http://gsgp.sourceforge.net.

2 Geometric semantic operators

The objective of these operators is to define modifications on the syntax of GP individuals that have a precise correspondence on their semantics. In this context, we can identify the semantics of an individual as a point in a multi-dimensional space, that we call semantic space. Given that in other approaches, like for instance real-valued genetic algorithms (GAs), individuals are actually vectors of numeric values, and thus they can be identified as points in a multi-dimensional space, the idea introduced in [10] is to define transformations of the syntax of GP individuals that correspond to well known operators of GAs. In this way, GP could "inherit" the known properties of those GA operators. Furthermore, contrarily to what typically happens in real-valued GAs or other heuristics, in the GP semantic space the target point is also known (it corresponds to the vector of expected output values in supervised learning) and the fitness of an individual is simply given by the distance between the point it represents in the semantic space and the target point. The real-valued GA operators that we want to "map" on GP semantic space are *geometric crossover* and *ball mutation*. In real-valued GAs, geometric crossover produces an offspring that stands in the segment that joins the parents. It was proven in [8] that in cases where the fitness is a direct function of the distance to the target (like the case we are interested in here) this offspring cannot have a worse fitness than the worst of its parents. Ball mutation consists in a random perturbation of the coordinates of an individual. It was shown in [10, 13] that it induces a unimodal error surface for all the problems whose objective is finding a function that maps input data into known target values (like regression and classification). The definitions of the operators that correspond to geometric crossover and ball mutation in the GP semantic space are, as given in [10], respectively:

Definition 1 (*Geometric Semantic Crossover*) Given two parent functions $T_1, T_2 : \mathbb{R}^n \rightarrow \mathbb{R}$, the geometric semantic crossover returns the real function $T_{XO} = (T_1 \cdot T_R) + ((1 - T_R) \cdot T_2)$, where T_R is a random real function whose output values range in the interval [0,1].

Definition 2 (*Geometric Semantic Mutation*) Given a parent function $T : \mathbb{R}^n \rightarrow \mathbb{R}$, the geometric semantic mutation with mutation step *ms* returns the real function $T_M = T + ms \cdot (T_{R1} - T_{R2})$, where T_{R1} and T_{R2} are random real functions.

Geometric semantic crossover creates an offspring whose semantics is a linear combination of the semantics of the parents with coefficients in [0,1] whose sum is 1, and thus corresponds to geometric crossover in the semantic space. To constrain T_R to producing values in [0,1] we use the sigmoid function: $T_R = (1 + e^{-T_{rand}})^{-1}$ where T_{rand} is a random tree with no constraints on the output values. Geometric semantic mutation perturbs each coordinate of the semantics of an individual of a

"small" (positive or negative) amount (given by the difference between two random expressions) and the magnitude of this perturbation can be changed by setting the *ms* (mutation step) parameter. It clearly corresponds to ball mutation in the semantic space. Even though not asserted in the original definition given in [10], our preliminary experiments on a set of real-life applications have shown that GSGP is able to produce better results (in particular on unseen data) when also the outputs of the T_{R1} and T_{R2} random expressions are limited to the [0,1] range. For this reason, both the implementation presented in this paper and the preliminary version used to produce the results presented in [13] constrain the values of T_{R1} and T_{R2} into [0,1] exactly with the same method used to constrain T_R in the implementation of geometric semantic crossover.

3 The proposed implementation

A first explanation of geometric semantic operators implemented in the C++ library that we propose here is given in [13, 14], including a detailed pseudo-code [13]. The basic idea is that, given the parents and the random trees to be used (as well as the value of the *ms* parameter for geometric semantic mutation), there is only one possible way to apply the geometric semantic operators. Therefore, there is no reason to store the whole tree structure of the offspring. For instance, for crossover, given the parents T_1 and T_2, and the random tree T_R, we can simply store a tuple $(crossover, \&T_1, \&T_2, \&T_R)$ where, for each individual $\tau, \&\tau$ is a memory reference (or *pointer*) to τ. Analogously, there is no reason for calculating the fitness of the offspring based on its tree structure. Fitness is simply a distance between the semantic vector and the target one. So, all we need to calculate fitness is the semantic vector, which can be easily obtained from the semantic vectors of T_1, T_2 and T_R by applying the definition of geometric semantic crossover. A completely analogous reasoning can be done for geometric semantic mutation. Being able to calculate fitness simply by calculating a distance between vectors of numbers, instead of having to evaluate tree structures, is the main reason for the efficiency of the proposed implementation. Concluding, we need to store in memory the tree structures of the individuals in the initial population and of a set of random individuals. Then, by maintaining a structure of memory references and semantic vectors, we can perform all the steps of the evolution and have all the information needed to reconstruct the final solution offline, at termination.

We remark that the idea of using memory references is not new at all in GP. Even the original book of Koza contains the idea of reusing ADFs by means of references [7]. Also, among the several other existing examples, one may think of the several studies aimed at reusing code in standard GP(like for instance [1, 3, 9, 16]) or in cartesian GP (like [15]). Thus, it may be tempting to use memory references in STGP the same way they are used in GSGP (and it is not impossible that some of the existing implementations actually already do it). However, the standard genetic operators would not be able to benefit from the same advantages it brings to GSGP in terms of fast fitness evaluation. Unlike GSGP, where the operators always produce offspring with a prefixed combination of the parents, STGP normally

produces offspring whose fitness is not easily calculated from the fitness of the parents.

The cost of the proposed GSGP algorithm in terms of time and space for evolving a population of n individuals for g generations is $O(ng)$. We point out that, in terms of memory occupation, the computational complexity becomes $O(ng + t)$ (where t is the size of the trees in the initial population plus any random trees generated) if random trees are not reused during a GP run (i.e., in case every time a random tree is needed, a new one is created and stored). If the random trees are reused (as in the implementation used in [13]), the term t is a constant and thus the expression of the complexity is again $O(ng)$. With this computational complexity, GSGP ends up being even more efficient than standard GP, given that in standard GP the cost of evaluating the individuals in the population strongly depends on the size of those individuals, while in GSGP that cost is constant. In order to provide an idea about the differences in terms of execution time between the proposed GSGP implementation and a standard GP implementation (STGP), we report in Table 1 the median execution time (of 30 runs) of the two approaches on three different real-life problems described in [2].

Here, by "standard" GP, we mean a version of GP that is completely consistent with the one defined by Koza in [7]: genetic operators used in STGP are subtree swapping crossover and subtree mutation. Crossover selects the subtrees to be swapped with uniform probability from the parents. Mutation selects a subtree with uniform probability and replaces it with a random tree generated using the grow initialization method with a maximum tree depth equal to 6. In standard GP, every individual created either by crossover or mutation is inserted in the new population, which is stored in memory. In other words, all the syntactic structures of all the individuals that have been generated during the evolution are stored in memory.

In can be observed that GSGP requires a significantly lower amount of time in order to execute a run when compared to STGP. Moreover, while the execution time of STGP is strongly dependent on the size of the individuals (which is different in every run, and this is the reason why STGP exhibits such high standard deviation values), it is absolutely not the case for GSGP (where, in fact, standard deviation values are much smaller).

Example Let us consider the simple initial population P shown in Table 2a and the simple pool of random trees that are added to P as needed, shown in Table 2b. For simplicity, we will generate all the individuals in the new population (that we call P' from now on) using only crossover, but a completely analogous reasoning can be done in case mutation is also used. Besides the representation of the individuals in infix notation, these tables also contain an identifier (Id) for each individual ($T_1, \ldots, T_5$ and $R_1, \ldots, R_5$). These identifiers will be used to represent the different individuals, and the individuals created for the new population will be represented by the identifiers $T_6, \ldots, T_{10}$. The individuals of the new population P' are simply represented by the set of entries exhibited in Table 2c. This table contains, for each new individual, references to the trees that have been used to build it, and the name of the operator used to generate it (either "crossover" or "mutation"). For example, the individual T_6 is generated by crossover of T_1 and

Table 1 Execution time (in seconds) of STGP and GSGP on three real-life applications described in [2]

	STGP			GSGP		
Problem	BIOAV	PPB	TOX	BIOAV	PPB	TOX
Median	1091.6	240.0	691.9	63.6	70.3	129.9
Standard deviation	655.4	381.7	400.7	3.6	0.7	3.8

Median and standard deviation obtained considering 30 independent runs

Table 2 Illustration of the example described in Sect. 3

(a) The initial population

Id	Individual
T_1	$x_1 + x_2x_3$
T_2	$x_3 - x_2x_4$
T_3	$x_3 + x_4 - 2x_1$
T_4	x_1x_3
T_5	$x_1 - x_3$

(b) The random trees used by crossover

Id	Individual
R_1	$x_1 + x_2 - 2x_4$
R_2	$x_2 - x_1$
R_3	$x_1 + x_4 - 3x_3$
R_4	$x_2 - x_3 - x_4$
R_5	$2x_1$

(c) The representation in memory of the new population

Id	Operator	Entry
T_6	crossover	$\langle \&T_1, \&T_4, \&R_1 \rangle$
T_7	crossover	$\langle \&T_4, \&T_5, \&R_2 \rangle$
T_8	crossover	$\langle \&T_3, \&T_5, \&R_3 \rangle$
T_9	crossover	$\langle \&T_1, \&T_5, \&R_4 \rangle$
T_{10}	crossover	$\langle \&T_3, \&T_4, \&R_5 \rangle$

T_4 and using random tree R_1. Let us assume that now we want to reconstruct the genotype of one of the individuals in P', for example T_{10}. The Table 2 contains all the information needed to do this. In particular, from Table 2c we learn that T_{10} is obtained by crossover between T_3 and T_4, using random tree R_5. Thus, from the definition of geometric semantic crossover, we know that it will have the following structure: $(T_3 \cdot R_5) + ((1 - R_5) \cdot T_4)$. The remaining Table 2a, b, that

contains the syntactic structure of T_3, T_4 and R_5, provides the rest of the information we need to completely reconstruct the syntactic structure of T_{10}, which is $((x_3 + x_4 - 2x_1) \cdot (2x_1)) + ((1 - (2x_1))(x_1 \cdot x_3))$ and upon simplification becomes $-x_1 \cdot (4x_1 - 3x_3 - 2x_4 + 2x_1 \cdot x_3)$.

4 Using the library

As in any GP system, using the library requires the definition of the elements that are specific to the problem (the fitness function, the primitive functions used to build the individuals and the terminal symbols), plus a set of parameters. In the documentation of the library, the user can find what functionalities should be modified in order to adapt the library to a specific problem. A configuration file (called *configuration.ini*, included in the download package) allows the user to specify the appropriate parameters, that are described in the documentation. To compile the file GP.cc containing the GP algorithm that uses the proposed library, just execute the following command:

```
g++ -Wall -O0 -g GP.cc -o GP
```

Running the program requires the execution of the command:

```
./GP -train_file train.txt -test_file test.txt
```

In this case, *train.txt* and *test.txt* are example files (included in the download package) containing training and test instances. The training and test files must contain, on the first line, the number of independent variables, and on the second line the number of instances. The columns contain the values of the independent variables (features) while rows contain the instances. The last column contains the dependent (target) variable. Executing this command with the provided training and test files and the provided configuration file will run 1,000 generations using 200 individuals. Two text files will be created, containing the fitness values obtained in each generation by the best individual on the training set, and the same individual on the test set. The user can write a few lines of code to store many other types of information related to a run.

The library is very easy to modify and the available documentation can provide further help. In order to use the library to tackle a user-defined problem, the user should follow these initial steps:

- Create training and test sets with the format specified above;
- If the root mean square error is not the desired fitness function, modify the two C++ functions used to calculate training and test fitness;
- If other mathematical operators other than the four already defined (+,−,* and protected division) should be considered, (1) declare an object that represents

the new operator and (2) define the semantics of the new operator in the eval function.
- Change the values of the parameters in the configuration file in order to adapt them to the considered problem.

The C++ source code can be compiled under Linux, Windows and Cygwin. The project is self-contained and only depends on standard libraries. Documentation is provided, including a user's manual and a description of the functions and data structures used.

5 Future developments

This first version of GSGP is obviously aimed at researchers, in the sense that it allows the use of any training and test files and the modification of any running parameter and documented code. In other words, it allows an easy exploration of the powerful capabilities of this novel system. However, it does not yet allow the retrieval of specific solutions to be used outside of this library. This will be included in the next release of GSGP.

Acknowledgments This work was partially supported by national funds through FCT under contract PEst-OE/EEI/LA0021/2013 and by projects EnviGP (PTDC/EIA-CCO/103363/2008), MassGP (PTDC/EEI-CTP/2975/2012) and InteleGen (PTDC/DTP-FTO/1747/2012), Portugal.

References

1. P.J. Angelin, J.B. Pollack, The evolutionary induction of subroutines, in *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, Lawrence Erlbaum, pp. 236–241 (1992)
2. F. Archetti, S. Lanzeni, E. Messina, L. Vanneschi, Genetic programming for computational pharmacokinetics in drug discovery and development. Genet. Progr. Evol. Mach. **8**, 413–432 (2007)
3. O. Brock, in *Evolving reusable subroutines for genetic programming*, ed. by J.R. Koza. Artificial Life at Stanford 1994, pp. 11–19. Stanford Bookstore, Stanford, California, 94305–3079 USA, June 1994
4. M. Castelli, D. Castaldi, I. Giordani, S. Silva, L. Vanneschi, F. Archetti, D. Maccagnola, An efficient implementation of geometric semantic genetic programming for anticoagulation level prediction in pharmacogenetics, in *16th Portuguese Conference on Artificial Intelligence (EPIA 2013)*, Springer. (2013)
5. M. Castelli, S. Silva, L. Vanneschi, A. Cabral, M.J. Vasconcelos, L. Catarino, J.M.B. Carreiras. Land cover/land use multiclass classification using GP with geometric semantic operators, in *Proceedings of the 16th European conference on Applications of Evolutionary Computation*, (Springer, Berlin, Heidelberg, 2013), EvoApplications'13, pp. 334–343
6. M. Castelli, L. Vanneschi, S. Silva, Prediction of high performance concrete strength using genetic programming with geometric semantic genetic operators. Expert Syst. Appl. **40**(17), 6856–6862 (2013)
7. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (MIT Press, Cambridge, 1992)
8. K. Krawiec, P. Lichocki, Approximating geometric crossover in semantic space, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation* (ACM, Montreal, 2009), pp. 987–994
9. E.G. Lopez, R. Poli, C.C. Coello, in *Reusing Code in Genetic Programming*, ed. by M. Keijzer et al., Genetic Programming, volume 3003 of Lecture Notes in Computer Science, (Springer, Berlin 2004), pp. 359–368

10. A. Moraglio, K. Krawiec, C.G. Johnson, in: *Geometric Semantic Genetic Programming*. Parallel Problem Solving from Nature, PPSN XII (part 1), Lecture Notes in Computer Science, vol 7491 (Springer, 2012), pp. 21–31
11. R. Poli, W.B. Langdon, N.F. McPhee, A field guide to genetic programming (2008), http://lulu.com, http://www.gp-field-guide.org.uk, (With contributions by J. R. Koza)
12. S. Silva, V. Ingalalli, S. Vinga, JMB. Carreiras, JB. Melo, M. Castelli, L. Vanneschi, I. Gonçalves, J. Caldas, Prediction of forest aboveground biomass: an exercise on avoiding overfitting, in *Proceedings of the 16th European conference on Applications of Evolutionary Computation*, (Springer, Berlin, Heidelberg, EvoApplications'13, 2013), pp. 407–417
13. L. Vanneschi, M. Castelli, L. Manzoni, S. Silva, in *A New Implementation of Geometric Semantic GP and Its Application to Problems in Pharmacokinetics*, ed. by K. Krawiec, A. Moraglio, T. Hu, A.S. Uyar, B. Hu. Proceedings of the 16th European Conference on Genetic Programming, EuroGP, vol 7831 (Springer, Vienna, Austria, LNCS, 2013), pp. 205–216
14. L. Vanneschi, S. Silva, M. Castelli, L. Manzoni, in *Geometric Semantic Genetic Programming for Real Life Applications*. Genetic Programming Theory and Practice, Springer, Ann Arbor. To appear. (2013)
15. J. Walker, J. Miller, The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. Evol. Comput. IEEE Trans. **12**(4), 397–417 (2008)
16. T. Yu, C. Clack, Recursion, Lambda-Abstractions and Genetic Programming. in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, University of Wisconsin, Madison, Wisconsin, USA, Morgan Kaufmann, pp. 422–431 (1999)