SoftwareX 10 (2019) 100313

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

GSGP-C++ 2.0: A geometric semantic genetic programming framework

Mauro Castelli [a,*], Luca Manzoni [b]

[a] NOVA Information Management School (NOVA IMS), Universidade Nova de Lisboa, Campus de Campolide, 1070-312, Lisboa, Portugal
[b] Dipartimento di Matematica e Geoscienze, University of Trieste, 34127 Trieste, Italy

ARTICLE INFO

Article history:
Received 6 May 2019
Received in revised form 2 July 2019
Accepted 29 August 2019

Keywords:
Genetic programming
Semantics
Machine learning

ABSTRACT

Geometric semantic operators (GSOs) for Genetic Programming have been widely investigated in recent years, producing competitive results with respect to standard syntax based operator as well as other well-known machine learning techniques. The usage of GSOs has been facilitated by a C++ framework that implements these operators in a very efficient manner. This work presents a description of the system and focuses on a recently implemented feature that allows the user to store the information related to the best individual and to evaluate new data in a time that is linear with respect to the number of generations used to find the optimal individual. The paper presents the main features of the system and provides a step by step guide for interested users or developers.

© 2019 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Code metadata

Current code version	2.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX_2019_170
Code Ocean compute capsule	https://doi.org/10.24433/CO.5881521.v1
Legal Code License	GNU General Public License (GPL)
Code versioning system used	none
Software code languages, tools, and services used	C++
Compilation requirements, operating environments & dependencies	The source code can be compiled under Linux, Windows and Cygwin.
If available Link to developer documentation/manual	http://gsgp.sourceforge.net/?page_id=51
Support email for questions	mcastelli@novaims.unl.pt

1. Motivation and significance

Recent years have seen a growing interest in the definition of semantic methods in Genetic Programming (GP) [1]. While the definition of semantics is not unique, it commonly refers to the vector of outputs produced by the evaluation of a GP individual on a set of training cases. Initial attempts to include the concept of semantics in GP focused on indirect methods (i.e., where standard syntax-based genetic operators are used and semantic criteria are considered to accept or reject newly created individuals). While these methods provide some beneficial effects on GP performance, they require a non-negligible computational effort due to the evaluation of a vast number of useless individuals (the ones that are not accepted based on the semantic criterion). This problem has been overcome with the definition of direct methods, that are able to include the concept of semantics in GP by defining particular genetic operators that, differently from the standard ones, have a direct effect on the semantics of the individuals [2]. While these operators have important properties [2] that make them particularly useful in regression and classification problems, they present important drawbacks: first of all, individuals generated by geometric semantic operators (GSOs) are characterized by a bigger size (i.e., number of nodes) with respect to the size of their parents. Individuals that grow exponentially with the number of generations (when crossover is applied) are an important limitation that must be taken into account when semantic operators are used to address complex problems, where hundreds of generations are necessary to generate a good quality

* Corresponding author.
E-mail addresses: mcastelli@novaims.unl.pt (M. Castelli), lmanzoni@units.it (L. Manzoni).

https://doi.org/10.1016/j.softx.2019.100313
2352-7110/© 2019 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

solution. In fact, the evaluation of such individuals makes the GP process unbearably slow. Thanks to the software presented in this paper, GSOs gained popularity in recent years, and it was possible to use them to address complex real-world problems which imply processing a vast amount of data [3]. The software, called GSGP-C++ (Geometric Semantic Genetic Programming in C++) is nowadays a reference in the GP community.

With the new version of the software, it is now possible to save the best model returned at the end of the evolution using it in a production scenario. This paper describes the new version of the software that makes it possible to save the information needed to reconstruct the optimal individual and to apply this individual to test data, i.e., data which has not been previously used for evolving the GP population. While reconstructing the full individual is a time-consuming task, it is possible to store all the components of the best solution in such a way that evaluating new instances will require a time that is linear with respect to the number of generations used to evolve the final solution.

2. Description of the software

The GSGP-C++ framework implements the GSOs outlined in [2] for symbolic regression that are defined as follows:

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].

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 whose output values range in the interval [0, 1].

The main idea that made it possible to implement these operators is that, given the parents and the random trees to be used, 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 π, $\&\pi$ is a memory reference (or pointer) to π. 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. 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. The cost, in terms of time and space, for evolving a population of n individuals for g generations is $O(ng)$ (the reader is referred to [4] for the details).

2.1. Reconstructing the best solution

In order to use the model returned by a GP run, the system stores the following information:

- the file called "individuals.txt" stores the individuals used in the GP run as mathematical expressions. The first part of the file contains the individuals that form the initial population, while the remaining part of the file stores the random individuals. These parameters can be specified in the configuration file of the system (the reader is referred to the documentation available at http://gsgp.sourceforge.net/.) All the individuals are stored in Infix notation, with the operators written in-between their operands (i.e., the usual way we write expressions). Brackets are used to make the precedence between operators explicit. An example, with the first lines of the file "individuals.txt", is reported in Fig. 1.
- the file "trace.txt" contains the information needed to reconstruct the optimal individual. In detail, each line of the file stores the data related to each crossover, reproduction and mutation event as a $6-tuple$ that contains the following information:
  - if a crossover event happened, the $6 - tuple$ contains the index of the first parent, the index of the second parent, the index of the random tree, a 0 indicating a crossover event, the index of the offspring and, finally, a dummy value (the dummy value allows the system to use the same data structure for mutation, crossover and reproduction events).
  - if a mutation event happened, the $6-tuple$ contains the index of the first random tree, the index of the second random tree, the index of the parent (i.e., the individual that is mutated), a 1 indicating a mutation event, the index of the offspring and, finally, the mutation step.
  - if a reproduction event happened, the $6-tuple$ contains the index of the parent, three dummy values, the index of the offspring and, finally, a dummy value.

  This information is used to evaluate the optimal model on unseen data following the procedure depicted in Fig. 2. For instance, the $6 - tuple$

  260 −& 278 −& 186 −& 1 −& 83 −& 0.695687

  indicates that individual 83 of the new population is obtained by performing the mutation of individual 186 of the current population. This semantic mutation must use the random trees 260 and 278 with a mutation step of 0.695687.

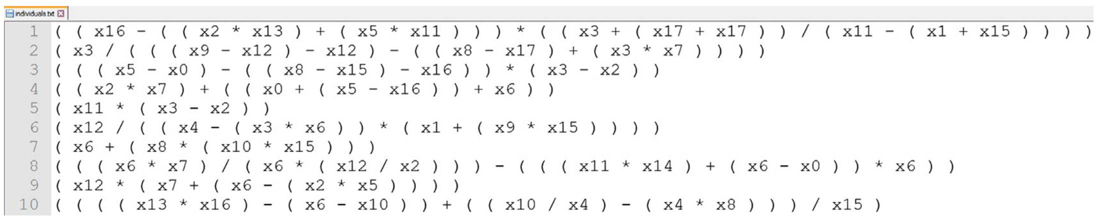
To optimize the performance of the system, the file "trace.txt" contains only the mutation, crossover and reproduction events that have been performed to build the optimal individual. This is obtained by the marking procedure depicted at the beginning of Fig. 2. The procedure performs an exploration of a directed acyclic graph starting from the node representing the best individual (i.e., the one at the last generation). This implementation requires a time that is linear with respect to the number of nodes of the graph which is, at most, $n \cdot g$, where n is the population size and g the number of generations.

Considering that the evaluation procedure has a complexity of $O(ng)$, the global procedure (i.e., marking all the nodes reachable from the best individual and evaluating the resulting graph) retains the same complexity.

3. Using the system

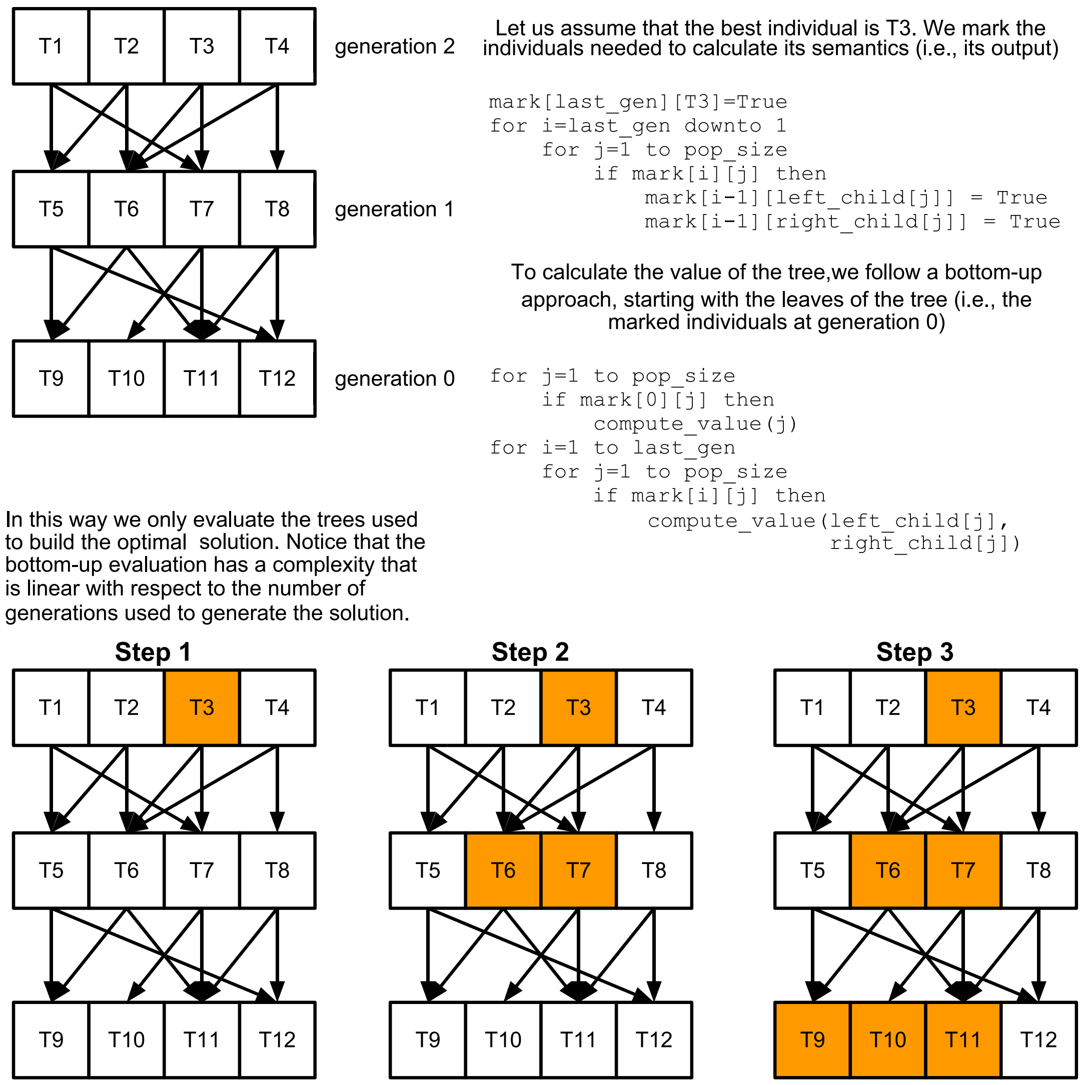
This section briefly explains how to use the system.

The first step for using the library requires the definition of the fitness function, the functional and terminal symbols, and a set of parameters. The default fitness function is the root mean squared error between target and predicted values. Additionally, the software already provides an implementation of the mathematical operator and uses all the dependent variables as terminal symbols. It is possible to specify a range of constant values to be used as terminal symbols. 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 downloadable package) allows



```
( ( x16 - ( ( x2 * x13 ) + ( x5 * x11 ) ) ) * ( ( x3 + ( x17 + x17 ) ) / ( x11 - ( x1 + x15 ) ) ) )
( x3 / ( ( ( x9 - x12 ) - x12 ) - ( ( x8 - x17 ) + ( x3 * x7 ) ) ) )
( ( ( x5 - x0 ) - ( ( x8 - x15 ) - x16 ) ) * ( x3 - x2 ) )
( ( x2 * x7 ) + ( ( x0 + ( x5 - x16 ) ) + x6 ) )
( x11 * ( x3 - x2 ) )
( x12 / ( ( x4 - ( x3 * x6 ) ) * ( x1 + ( x9 * x15 ) ) ) )
( x6 + ( x8 * ( x10 * x15 ) ) )
( ( ( x6 * x7 ) / ( x6 * ( x12 / x2 ) ) ) - ( ( ( x11 * x14 ) + ( x6 - x0 ) ) * x6 ) )
( x12 * ( x7 + ( x6 - ( x2 * x5 ) ) ) )
( ( ( ( x13 * x16 ) - ( x6 - x10 ) ) + ( ( x10 / x4 ) - ( x4 * x8 ) ) ) / x15 )
```

Fig. 1. Solutions stored in Infix notation in the file "individuals.txt".



Let us assume that the best individual is T3. We mark the individuals needed to calculate its semantics (i.e., its output)

```
mark[last_gen][T3]=True
for i=last_gen downto 1
    for j=1 to pop_size
        if mark[i][j] then
            mark[i-1][left_child[j]] = True
            mark[i-1][right_child[j]] = True
```

To calculate the value of the tree,we follow a bottom-up approach, starting with the leaves of the tree (i.e., the marked individuals at generation 0)

```
for j=1 to pop_size
    if mark[0][j] then
        compute_value(j)
for i=1 to last_gen
    for j=1 to pop_size
        if mark[i][j] then
            compute_value(left_child[j],
                          right_child[j])
```

In this way we only evaluate the trees used to build the optimal solution. Notice that the bottom-up evaluation has a complexity that is linear with respect to the number of generations used to generate the solution.

Fig. 2. Evaluation of the optimal individual. The top of the figure depicts the marking procedure, while the bottom part of the plot shows the evaluation procedure. Each block represents an individual and each row is a population. Arrows represent the application of GSOs. White blocks are the ones that are not considered by the marking procedure, hence they are not involved in the process of building the optimal solution. On the other hand, orange blocks are the ones marked by the marking procedure and are the ones that must be evaluated to extract the semantics of the optimal individual.

the user to specify the appropriate parameters described in the documentation.

The file "configuration.ini" has important parameters to be specified for reconstructing the best model:

1. `random_tree`: this parameter specifies the cardinality of the pool of random trees used by the system.
2. `expression_file`: a value of 1 for this parameter indicates to the system to use the individuals contained in the file "individuals.txt". A value different from 1 implies that the system will create the individual by using one of the initialization methods that can be specified by using the parameter "`init_type`".
3. `USE_TEST_SET`: a value of 1 indicates that the system is used to apply the optimal solution to a set of unseen instances. A value different from 1 tells the system to perform the standard evolutionary process.

A typical use of the system to tackle a user-defined problem consists of the following steps:

- use the system in learning mode (i.e., by executing the standard evolutionary process). In this phase, set the parameters `USE_TEST_SET` to 0 and `expression_file` to 1 (if you want to run the system by using the individuals contained in the file "individuals.txt") or to 0 (if you want to randomly initialize the GP population);
- apply the best model obtained in the previous step to unseen data. In this phase, set both the parameters `USE_TEST_SET` and `expression_file` to 1. The output of the model is stored in the file "`evaluation_on_unseen_data.txt`"

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 to perform the standard learning process requires the execution of the command:

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

while to apply a solution to a set of unseen instances the user must run the following command:

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

In this case, *train.txt* and *test.txt* are the files containing training and test instances.

When the system is executed with the value of `USE_TEST_SET` equal to 1, the file containing the test instances does not need to contain the target values. This is carefully explained in the documentation of the system. Moreover, differently from the training and test files specified in learning mode, the test file must not contain on its second line the number of instances it stores.

The C++ source code can be compiled under Linux or Windows with 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.

4. Impact

The system described in this paper will allow GP practitioners to use the final model in a production environment, something that represents a fundamental contribution in the field of GP and, in particular, semantics-based methods. The system is nowadays the fastest-running GP system, allowing a speedup that is at least 20 times the one that characterizes a syntax-based GP system. We believe that this new implementation will make even popular the use of the semantics-based method and genetic programming: in particular, the system is scalable, allowing to solve problems characterized by a vast amount of data.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by national funds through FCT (Fundação para a Ciência e a Tecnologia), Portugal under project DSAIPA/DS/0022/2018 (GADgET). Mauro Castelli acknowledges the financial support from the Slovenian Research Agency, Slovenia (research core funding No. P5-0410).

References

[1] Vanneschi L, Castelli M, Silva S. A survey of semantic methods in genetic programming. Genetic Program Evolvable Mach 2014;15(2):195–214.
[2] Moraglio A, Krawiec K, Johnson CG. Geometric semantic genetic programming. In: Coello CAC, Cutello V, Deb K, Forrest S, Nicosia G, Pavone M, editors. Parallel problem solving from nature - PPSN XII: 12th international conference, Taormina, Italy, September 1–5, 2012, proceedings, part I. Berlin, Heidelberg: Springer Berlin Heidelberg; 2012, p. 21–31.
[3] Vanneschi L, Silva S, Castelli M, Manzoni L. Geometric semantic genetic programming for real life applications. In: Genetic programming theory and practice Xi. Springer; 2014, p. 191–209.
[4] Castelli M, Silva S, Vanneschi L. A C++ framework for geometric semantic genetic programming. Genetic Program Evolvable Mach 2015;16(1):73–81.