Cartesian Genetic Programming as an Optimizer of Programs Evolved with Geometric Semantic Genetic Programming

Ondrej Koncal and Lukas Sekanina(✉)

Faculty of Information Technology, IT4Innovations Centre of Excellence,
Brno University of Technology, Božetěchova 2, 612 66 Brno, Czech Republic
koncalo@gmail.com, sekanina@fit.vutbr.cz

Abstract. In Geometric Semantic Genetic Programming (GSGP), genetic operators directly work at the level of semantics rather than syntax. It provides many advantages, including much higher quality of resulting individuals (in terms of error) in comparison with a common genetic programming. However, GSGP produces extremely huge solutions that could be difficult to apply in systems with limited resources such as embedded systems. We propose Subtree Cartesian Genetic Programming (SCGP) – a method capable of reducing the number of nodes in the trees generated by GSGP. SCGP executes a common Cartesian Genetic Programming (CGP) on all elementary subtrees created by GSGP and on various compositions of these optimized subtrees in order to create one compact representation of the original program. SCGP does not guarantee the (exact) semantic equivalence between the CGP individuals and the GSGP subtrees, but the user can define conditions when a particular CGP individual is acceptable. We evaluated SCGP on four common symbolic regression benchmark problems and the obtained node reduction is from 92.4% to 99.9%.

Keywords: Cartesian Genetic Programming ·
Geometric Semantic Genetic Programming · Symbolic regression

1 Introduction

Geometric Semantic Genetic Programming (GSGP) is a recent branch of genetic programming (GP) in which specific genetic operators, the so-called *geometric semantic genetic operators*, directly work at the level of semantics rather than syntax [1]. In this context, the *semantics* is defined as the vector of outputs of a program on the different training data. GSGP is successful because geometric semantic operators induce a unimodal fitness landscape which is known to be relatively easy for search-based optimization algorithms. On many various symbolic regression and classification problems it has been shown that GSGP provides statistically better results than a common genetic programming and other

L. Sekanina et al. (Eds.): EuroGP 2019, LNCS 11451, pp. 98–113, 2019.
https://doi.org/10.1007/978-3-030-16670-0_7

machine learning methods in terms of the error score [2,3]. However, since the genetic operators used in GSGP produce offspring that are larger than their parents, the evolved programs unprecedentedly grow in their size. Some approaches addressing this issue have been developed (see Sect. 2.1), but the problem is still considered as unsolved.

This paper presents a new method capable of reducing the size of a program evolved with GSGP. This work is motivated by the fact that there could be a high-quality program created by GSGP for a given application, but the program is unfeasible for implementation on a platform with limited resources (e.g., in an embedded system with a small memory). In our approach, we consider this program (i.e. the result of GSGP) as a golden (reference) solution in terms of functionality and try to minimize its size. The optimized program should then be implemented in the target embedded system. In our preliminary experiments, we employed GP to reduce the number of nodes in several programs evolved with GSGP and keep the error at the same level. Because we optimized the entire programs without any decomposition and the programs were too complex, no reduction in the number of nodes has been achieved at all.

We introduce *Subtree Cartesian Genetic Programming* (SCGP) as an efficient optimizer of the size of programs evolved with GSGP. The *reference solution* (i.e. the program evolved with GSGP) is converted to the representation used in Cartesian Genetic Programming (CGP). In order to avoid scalability problems of the aforementioned preliminary approach, SCGP executes, in the first step, a series of CGP runs with the aim to minimize the number of nodes in all subtrees belonging to the CGP representation of the reference solution. These optimized subtrees are then paired and again optimized by CGP. Several iterations of the pairing strategy then lead to a single optimized program. The optimization process is thus decomposed into a number of CGP runs solving low-complexity optimization problems. CGP is used because of its well-known capabilities to optimize the phenotype size (as exemplary demonstrated, for example, for digital circuits in [4]). The proposed method is evaluated using four symbolic regression problems (%F, LD50, PPB and P3D, see Sect. 4.1) for which a significant reduction is reported in the number of nodes in the evolved trees.

2 Relevant Work

2.1 Geometric Semantic Genetic Programming

GP operators traditionally work in the syntactic space and manipulate the syntax of parents. The parents can also be modified based on their semantics which is defined as a vector of outputs of a program on the different training data [1].

GSGP creates new candidate solutions using *geometric semantic operators* working at the level of semantics. *Geometric semantic crossover* (GSC) and *geometric semantic mutation* (GSM) usually work as follows

$$\text{GSC: } T_{XO} = (T_1 \cdot T_R) + ((1 - T_R) \cdot T_2) \tag{1}$$

$$\text{GSM: } T_M = T + ms \cdot (T_{R1} - T_{R2}) \tag{2}$$

where $T, T_1, T_2 : \mathbb{R}^n \rightarrow \mathbb{R}$ are parents, $T_R, T_{R1}, T_{R2} : \mathbb{R}^n \rightarrow \mathbb{R}$ are random real functions with output values in interval $\langle 0, 1 \rangle$ and ms is a *mutation step*. Generating the output values in interval $\langle 0, 1 \rangle$ is ensured by the sigmoid function

$$T_R = (1 + e^{-T_{rand}})^{-1}, \tag{3}$$

where T_{rand} is a random tree with no constraints on the output values.

By applying these operators, one can effectively create a unimodal error surface for problems such as symbolic regression. The search process conducted in such a search space is then more efficient than in the case of a common GP. However, geometric semantic operators, by construction, always produce offspring that are larger than their parents, causing a fast growth in the size of the individuals. The growth is linear for GSM and exponential for GSC [1].

In order to reduce the size of candidate solutions, various approaches have been developed. One branch of the methods is based on simplifying the offspring during the evolution, for example, using a computer algebra system [1] or developing specific genetic operators such as subtree GSC and subtree GSM [5]. Recently, an on-the-fly simplification of trees was proposed capitalizing the fact that the individuals are always linear combinations of trees and repeated structures can be aggregated [3]. The original huge individuals created by GSGP were significantly reduced to several thousands of nodes.

Another approach relies on an efficient GSGP implementation, in which pointers to existing structures (trees) are recorded rather than all the new trees [6]. The method only stores the initial population and a set of randomly generated trees. A new record is created after performing GSC or GSM in form of $(crossover, \&T_1, \&T_2, \&T_R)$ or $(mutation, \&T, \&T_{R1}, \&T_{R2}, ms)$. The semantics of the individuals is also stored and used to compute the semantics of the offspring, again by means of the pointers to stored semantics records. This method enables to reduce the evaluation time as pre-calculated partial results are always available in the memory.

2.2 Cartesian Genetic Programming

In CGP [7], a candidate individual is modeled as a two-dimensional grid of nodes, where the type of nodes depends on a particular application. Each individual utilizes n_i primary inputs and n_o primary outputs. A unique address is assigned to all primary inputs and to the outputs of all nodes to define an addressing system enabling connections to be specified. As no feedback connections are allowed in the basic version of CGP, only directed acyclic graphs can be created. Each candidate individual is represented using $r \times c \times (n_a + 1) + n_o$ integers, where $r \times c$ is the grid size and n_a is the maximum arity of node functions. In this representation, the $n_a + 1$ integers specify one programmable node in such a way that n_a integers specify source addresses for its inputs and one integer determines its function. Finally, the l-back parameter defines how many columns of nodes in front of i-th column can be used as the data source for the i-th column.

New candidate individuals are created by mutation of selected genes (integers) of the chromosome. It is important to ensure that all randomly created gene values are within a valid interval (i.e. a valid candidate individual is always produced). Crossover is not normally used in CGP. CGP employs a simple search algorithm denoted $(1 + \lambda)$ which operates with a set of $1 + \lambda$ candidate individuals [7]. The initial population is created either randomly or heuristically, for example, an existing program can be employed. A new population is constructed by applying the mutation operator on the parent individual to generate λ offspring individuals. These offspring are then evaluated and the best performing individual is taken as a new parent. These steps are repeated until the time available for the evolution is exhausted or a suitable solution is discovered.

CGP was used to evolve new implementations of digital circuits and to optimize existing circuits, for example, in terms of the number of gates [4]. This is a very similar task to our problem – the program size reduction in GSGP.

3 Subtree CGP

Subtree CGP is a method that analyzes a log file created by a single run of GSGP in the framework [6] and minimizes the number of nodes in the best evolved individual. The result of GSGP will be called the *Golden Tree* (GT) in the paper. It has to be emphasized that GT is, in fact, distributed in the log file because the framework [6] only stores: (i) the trees used in the initial population, (ii) randomly created trees needed for geometric semantic operators and (iii) particular records about geometric semantic operators. These entities are linked using pointers. In order to extract GT, the log file has to be parsed and analyzed from the initial to the last population.

SCGP operates in two steps: (i) SCGP converts GT to the CGP representation. (ii) SCGP repeatedly executes a CGP-based optimizer on all elementary subtrees of GT and on various compositions of these optimized subtrees in order to create one compact representation of GT. SCGP does not guarantee the (exact) semantic equivalence between the CGP individuals and subtrees of GT, but the user can define conditions when a particular result of CGP is acceptable. In order to simplify this initial study, we will only consider GSGP utilizing GSM.

3.1 Obtaining the CGP Representation

As our benchmark problems are symbolic regression tasks with d independent variables and one output variable y, the basic SCGP parameters are $n_i = d$, $n_o = 1$, $r = 1$, $n_a = 2$, $l - \text{back} = c$ and $c = u$, where u is the total number of nodes in GT. Each CGP node can operate either as a constant (in interval $\langle -10.0; 10.0 \rangle$) or function (taken from a function set $\Gamma = \{+, -, *, /, (1 + e^{-x})^{-1}, e^x\}$, where division is protected, i.e. $x/0 = 1$). Despite the fact that this setup is application-specific and is typically given in the experimental part, it is provided here to simplify the following description of the method.

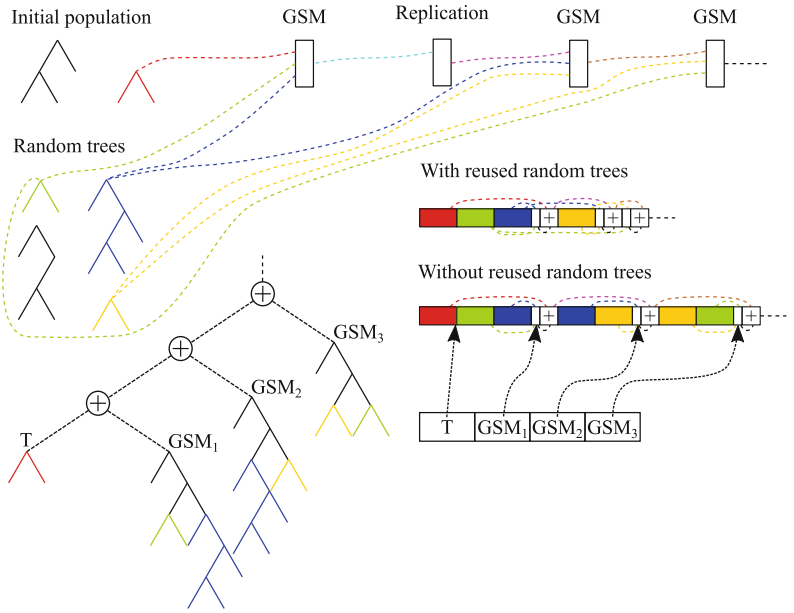


Fig. 1. GSGP to CGP conversion: original records created by GSGP (top); corresponding Golden Tree (bottom-left); CGP chromosome with and without reused subtrees (bottom-right).

Because GSGP operates with common syntax trees representing arithmetic expressions, their conversion to the CGP chromosome is straightforward; a tree is expressed as a string in the postfix notation and during the reading of this string from left to right, appropriate nodes (represented using three integers – two pointers to the inputs of the node and one node function) are created in the CGP chromosome. Note that the primary inputs representing the independent variables are internally handled as special nodes.

In order to build GT, SCGP parses the log file of a single GSGP run (generation by generation), but in such a way that only the nodes contributing to GT are identified and converted to the CGP representation. In particular,

- all subtrees representing the initial population of GSGP,
- all subtrees representing the randomly generated functions and
- arithmetic operators representing (GSM or simple replication) that are connecting these subtrees

are converted to the CGP representation using the procedure given above[1]. Figure 1 illustrates how the subtrees are converted to a CGP chromosome. The number of nodes in the CGP representation is equal to the number of nodes in GT if it were represented as a single tree. Please note that *ms* constants are represented by small white empty boxes in the CGP chromosome in Fig. 1.

[1] As all these trees are used in a single resulting solution, we will call them subtrees.

Some subtrees (e.g. random trees) can, however, be used multiple times in GT. In order to shorten the CGP chromosome, these multiple instances of a given subtree can be detected and only one instance of each subtree can be included into the CGP chromosome as seen in Fig. 1. For the benchmark problems used in this paper, this technique reduces the number of nodes 1.4–2.2×. It should be noted that this technique is NOT used in the current version of SCGP.

3.2 CGP-based Optimization of Subtrees

The proposed SCGP assumes that (i) only GSM was used in GSGP (i.e. no GSC) and (ii) the GSGP to CGP conversion does not apply any size reduction techniques, i.e. all (even multiple instances of) subtrees are preserved. These assumptions are important as they ensure that the CGP-based optimization can independently be performed on all the subtrees. Since the subtrees are connected via (associative and commutative) addition operators their optimization can be performed in an arbitrary order.

During the conversion process an array of indexes p is created, pointing to the subtrees that will further be optimized. These subtrees include a random tree from the initial population (T) and all subtrees whose root is the $\cdot$ operator from the $ms \cdot (T_{R1} - T_{R2})$ expressions created by GSM. Let s be the size of p. By means of p, all s subtrees can be identified in the CGP chromosome, their corresponding chromosomes extracted and used as initial seeds (denoted $\alpha[i]$) for the CGP-based subtree optimization.

A single CGP run is executed for each subtree in p. The objective of CGP is to minimize the number of nodes and keep the error at desired level. In order to accelerate the fitness (error) evaluation, responses of a particular subtree (that was identified in GT) are pre-calculated for the whole data set. In other words, the semantics $ST[i]$ is created for subtrees $\alpha[i]$, $i = 1, \ldots, s$ and data set D. Please note that these particular CGP runs operate in different search spaces because the chromosome sizes can, in principle, be different. The remaining CGP parameters (such as the population size, the number of generations, mutation rate etc.) are identical for all the CGP runs.

Thanks to the addition operators connecting all the subtrees, one can define an auxiliary vector $AUX[i]$ for each subtree $p[i]$

$$AUX[i](j) = y(j) - \sum_{\forall k: k \neq i} ST[k](j), \tag{4}$$

whose role is to define a contribution of the i-th subtree to the overall fitness, where $y(j)$ is a correct result for j-th fitness case. This vector is used as a golden solution (in terms of the error) during the subtree optimization conducted by CGP.

The fitness function reflecting the error of a candidate CGP individual g in the optimization of i-th subtree is defined as

$$f^{err}[i](g) = \frac{\sum_{j \in D} |AUX[i](j) - g(j)|}{|D|}, \tag{5}$$

where D is the training data set. The final fitness score is constructed hierarchically

$$f^{cgp}[i](g) = \begin{cases} \#\text{ of active nodes in g} & \text{if } f^{err}[i](g) \leq f^{err}[i](\alpha[i]) \\ \infty & \text{otherwise,} \end{cases} \quad (6)$$

where $\alpha[i]$ is the seed used in the initial population of CGP. The goal is to minimize the number of active nodes and, at the same time, keep the error at least identical with respect to the original subtree.

An individual satisfying the quality condition (i.e. its fitness f^{cgp} is no worse than the fitness of the seed individual) is called an *acceptable solution*. This quality condition can be re-defined according to user's requirements.

The best performing solution obtained in the CGP run is then used in the following steps of SCGP. As this solution can be semantically different w.r.t the seed, it is necessary to recalculate its $ST[i]$.

All remaining subtrees are optimized using the aforementioned procedure; however, updated versions of all $ST[i]$ vectors are always employed. All active nodes in these CGP chromosomes are marked in order to avoid useless evaluations of the inactive nodes in the next steps.

3.3 Subtree Pairing

After optimizing all elementary subtrees and with the aim of minimizing the total number of nodes of this set of subtrees (formally denoted `Collection` in the following pseudo-code), selected subtrees are paired and then minimized. The pairing procedure can be executed on arbitrarily chosen subtrees because all subtrees are connected using the associative and commutative addition operators (Fig. 1). As the following pseudo-code illustrates the pairing is performed in iterations until only one final tree (optimized GT) is obtained:

```
while (the number of subtrees > 1) do
    New_Collection = empty_set
    while (the number of subtrees in Collection > 1) do
      select two subtrees A and B according to Pairing Strategy
      create subtree C by joining A and B
      compute ST[C] = ST[A] + ST[B]
      run CGP (Sect. 3.2) to optimize the number of nodes in C
      remove A and B from Collection
      update ST[C]
      insert C to New_Collection
    end while
    Collection = Union (Collection, New_Collection)
end while
run CGP (Sect. 3.2) to optimize the final tree
update ST[0]
```

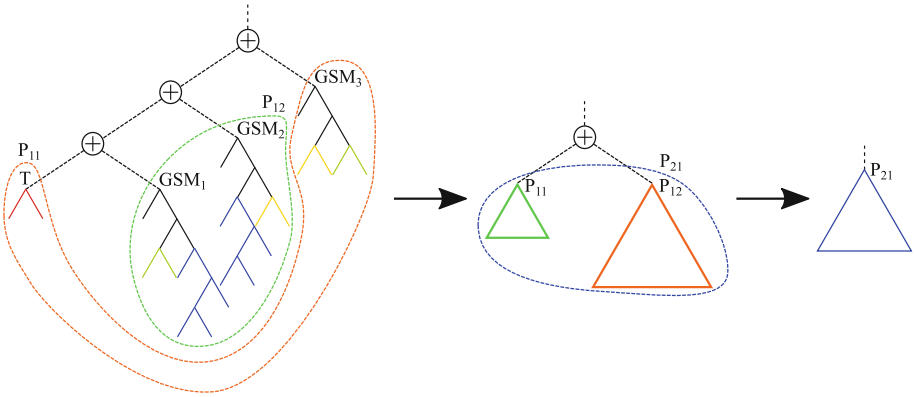


Fig. 2. Subtree pairing example

Figure 2 shows the pairing mechanism on a structure containing four subtrees. We propose three pairing strategies whose impact on the SCGP performance will be analyzed in Sect. 4.3:

- MAXDIFF – subtrees showing a maximum difference in their sizes are paired;
- MINDIFF – subtrees showing a minimum difference in their sizes are paired;
- RAND – randomly selected subtrees are paired;

The final (sub)tree is the result of SCGP. In total, SCGP executes $2s - 1$ elementary CGP runs, where s is the number of subtrees in GT.

4 Results

This section provides a basic experimental evaluation of the proposed method. It starts with a description of the data sets used in our experiments. It presents the results of the conventional GSGP (i.e. GTs for our data sets) that were obtained with the framework from [6]. SCGP capabilities to reduce the GT size are analyzed with respect to the number of generations allowed in a single CGP run and the pairing strategy used.

4.1 Data Sets

The proposed method is evaluated using four data sets that are commonly used in connection with the GSGP research. Three data sets are from the area of pharmacokinetics: (i) predicting the value of human oral bioavailability of a candidate new drug as a function of its molecular descriptors (*bioav*, denoted %F in the literature), (ii) predicting the value of the plasma protein binding level of a candidate new drug as a function of its molecular descriptors (LD50) and (iii) predicting the value of the toxicity of a candidate new drug as a function

of its molecular descriptors (PPB; denoted %PPB in the literature) [8,9]. Finally, the P3D is a data set of physicochemical properties of protein tertiary structure from UCI Machine Learning Repository [10]. In each data set, 70% records are used for training and 30% for test.

Table 1 gives the number of independent variables and fitness cases for each data set. It also provides basic parameters of GTs evolved with GSGP including *training fitness* (f_{train}) and *test fitness* (f_{test}) as it will be discussed in Sect. 4.2.

Table 1. Parameters of data sets and parameters of Golden Trees evolved with GSGP.

	Independent variables (rows)	Fitness cases	f_{train} (best)	f_{test} (best)	Size (nodes)	Reduced size (nodes)
bioav	241	359	19.455	26.938	22,285	9,863
PPB	628	131	4.312	30.768	22,907	10,291
LD50	626	234	1336.960	1495.621	21,568	9,934
P3D	9	45,730	3.981	3.999	5,764	3,992

4.2 Obtaining Reference Solutions with GSGP

In order to evaluate the proposed method, we first establish reference solutions (i.e. Golden Trees) for each benchmark problem. The GSGP implementation from [6] is used with the parameters summarized in Table 2, function set $\{+, -, *, /\}$ and the fitness function defined as the mean absolute error between the output of a candidate individual and the golden output (y). GSGP only employs GSM in which the ms step is randomly generated from interval $\langle 0, 1\rangle$. This parameter setting is almost identical with [2]; the main difference lies in allowing 10× more evaluations for GSGP to ensure that resulting GTs are of high quality and non-trivial in order to later illustrate the performance of SCGP.

Figure 3 shows box plots constructed from 40 independent runs for each data set. In particular, (i) the program size, (ii) the reduced program size and (iii) the training fitness are reported. Properties of GTs are summarized in Table 1.

As we used a state of the art GSGP implementation, a common setup of GSGP and the obtained results are consistent with [9] in terms of quality, we can conclude that relevant reference solutions (GTs) were generated with GSGP.

Table 2. Parameters of GSGP

Population size	2000	Mutation prob.	0.9
Generations	1000 (300 for *P3D*)	Max. tree depth	8
# of random trees	500	Single-node trees	Not used
Tree initialization	Ramped Half-and-Half	Tournament size	4

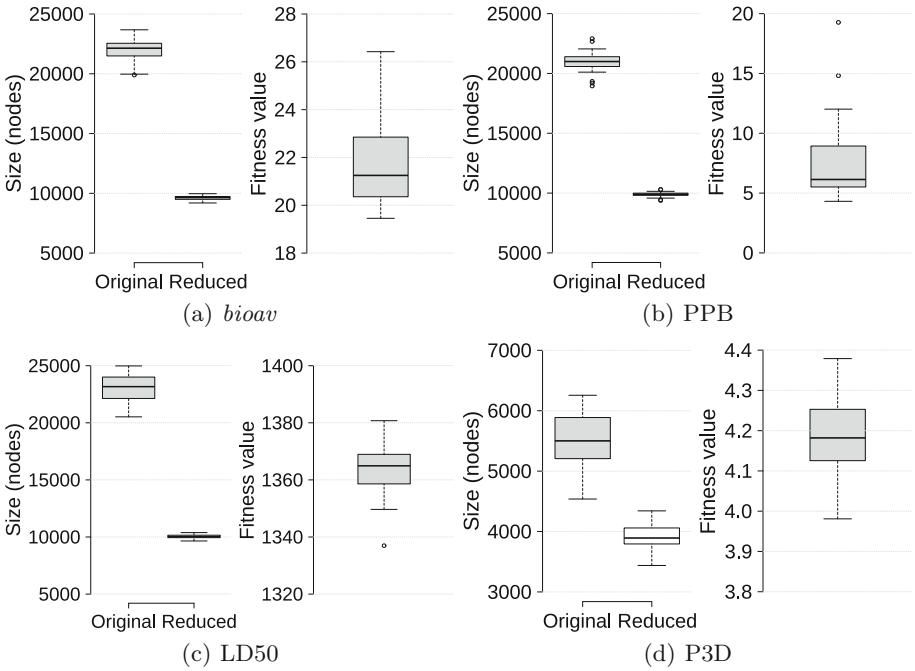


Fig. 3. Program size and training fitness values obtained from 40 independent GSGP runs.

4.3 Experiments with SCGP

From GTs reported in Table 1, all subtrees are extracted and used as the initial solutions α_i for the size optimization conducted by SCGP. Each subtree is optimized separately in a single CGP run in which the population size is 8 and the mutation probability is 0.08 (per gene). The impact of these parameters on the SCGP performance was not the subject of a detailed analysis because these values are typical for CGP. As the problem instances that are optimized by CGP are relatively simple, only a low number of generations seems to be sufficient in comparison with typical numbers from the CGP literature [7]. Our search for the most suitable number of generations was conducted with 10, 50 and 100 generations. The fitness computation exploits the pre-computed values $ST[i]$ as explained in Sect. 3.2. The impact of three pairing strategies (MAXDIFF, MINDIFF and RAND) on the SCGP performance is also investigated.

The Number of Generations. Figure 4 shows the impact of the number of generations (used by CGP when it optimizes one subtree) on the execution time and the size of resulting programs when applied on the *bioav* data set. The box plots are derived from 20 independent SCGP runs in which the RAND pairing strategy is employed. Because the population size is always 8, the execution time

Table 3. Parameters of the best solutions obtained with SCGP when CGP produces 10, 50 and 100 generations for optimizing one subtree.

	Generations	10	50	100
bioav	nodes; f_{train}; f_{test}	653; 19.46; 26.51	322; 19.45; 26.39	242; 19.40; 26.65
	Acceptable solutions	1	2	5
PPB	nodes; f_{train}; f_{test}	7643; 4.29; 30.04	2178; 4.29; 28.86	1723; 4.26; 26.80
	Acceptable solutions	3	3	1
LD50	nodes; f_{train}; f_{test}	46; 1334.8; 1489.1	19; 1333.0; 1492.1	17; 1327.2; 1473.8
	Acceptable solutions	11	13	9
P3D	nodes; f_{train}; f_{test}	939; 3.98; 3.99	281; 3.97; 3.98	202; 3.98; 3.98
	Acceptable solutions	20	18	18

is growing proportionally to the number of generations. If more generations are produced, one can obtain more compact programs; however, the dependency is not linear. A similar pattern was observed for the remaining data sets (not shown because of limited space). Table 3 summarizes the number of nodes, training fitness and test fitness for the most compact solutions that were evolved. If an individual is marked as an acceptable solution then we require that its fitness values (training and test errors) are not worse than the fitness values (training and test errors) of the reference solution. The number of acceptable solutions is not proportional to the number of generations, but the resulting program is always smaller if more generations are produced. As we are primarily interested in reducing the number of nodes, we will consider 100 generations as a reasonable setting in the final experiments.

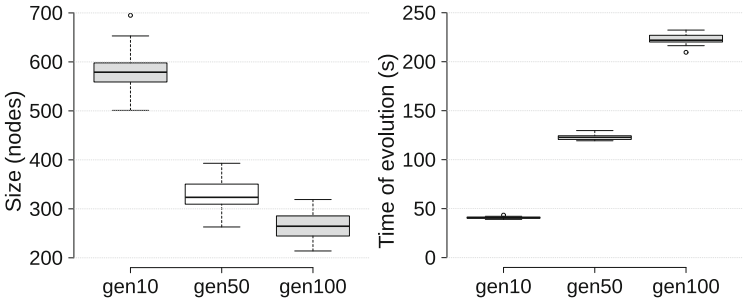


Fig. 4. The program size and the time of evolution obtained with SCGP on *bioav* when CGP produces 10, 50 and 100 generations for optimizing one subtree.

Subtree Pairing Strategies. Figure 5 shows the impact of three pairing strategies on the execution time and the size of programs resulting from SCGP when applied on particular data sets. The boxplots are derived from 20 independent runs of SCGP in which 50 generations are produced per CGP run. While MAXDIFF is the best performing approach, RAND gives slightly worse results than

MAXDIFF and MINDIFF is clearly the worst one (as there are, in principle, more limited options to reduce the program size in comparison with the other approaches). Regarding the execution time, MAXDIFF is the most expensive approach except one case (P3D). Table 4 summarizes the number of nodes, the training fitness and test fitness for the smallest, but still acceptable programs. The number of acceptable solutions is very low in some cases, but recall that only 50 generations per CGP run are used in this set of experiments.

Table 4. Parameters of the best solutions obtained with SCGP utilizing different pairing strategies

	Pairing strategy	MAXDIFF	MINDIFF	RAND
bioav	nodes; f_{train}; f_{test}	288; 19.45; 25.38	383; 19.45; 26.69	322; 19.45; 26.39
	Acceptable solutions	3	1	2
PPB	nodes; f_{train}; f_{test}	1816; 4.29; 30.67	3755; 4.22; 28.94	2178; 4.29; 28.86
	Acceptable solutions	1	1	3
LD50	nodes; f_{train}; f_{test}	27; 1334.0; 1406.9	21; 1308.5; 1382.9	19; 1333.0; 1492.1
	Acceptable solutions	11	8	13
P3D	nodes; f_{train}; f_{test}	198; 3.98; 3.98	457; 3.98; 3.99	281; 3.97; 3.98
	Acceptable solutions	19	17	18

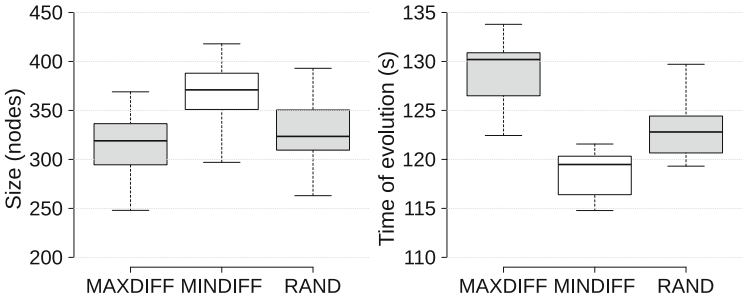
4.4 Final Results

For the final set of experiments we use the SCGP setup given in Sect. 4.3, but with the best performing number of generations (100) and pairing strategy (MAXDIFF) identified in the previous experiments. Figure 6 shows the program size and the execution time in form of box plots derived from 20 independent SCGP runs. The best obtained solutions are compared in Table 5 against GTs for all data sets. In the case of PPB, the setup used for SCGP did not provide any acceptable solution (a very compact solution with 1270 nodes was discovered, but its f_{test} is slightly higher than the golden tree exhibits). Hence, we took the best solution $\text{PPB}_{Tab.3}$ from Table 3 in order to report the best performing solutions for all data sets in one table.

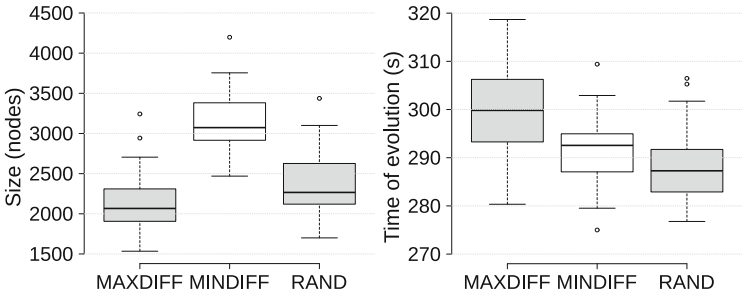
4.5 Discussion

We have shown that if GSGP is followed by SCGP a significant reduction in the number of nodes can be obtained while the (error) fitness is not worsened.

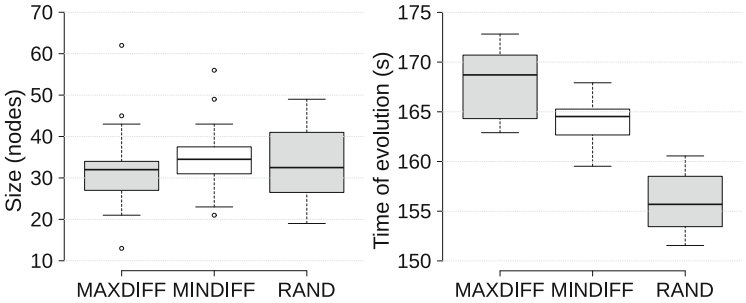
In paper [3], where PPB was used as one of benchmark problems, the median size of the best individual obtained by a common GSGP (utilizing GSM and GSC) is 2.29e+64 nodes and the median size of the best individual evolved with their optimized GSGP-Red method is 12,185 nodes. Both numbers are much bigger with respect to our results. For the remaining benchmarks, no relevant results are available in the literature.



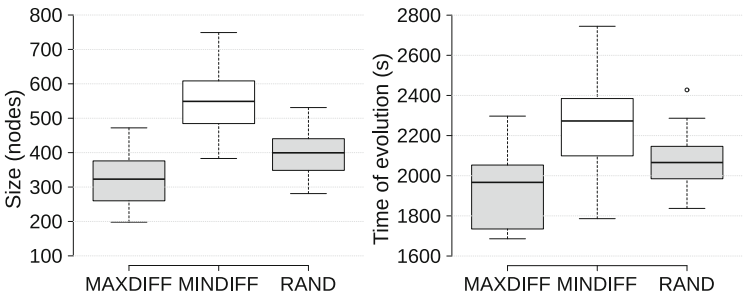
(a) *bioav*



(b) PPB



(c) LD50



(d) P3D

Fig. 5. The program size and the time of evolution obtained with SCGP for three subtree pairing strategies.

In order to further investigate how the pairing mechanism works, we report Table 6 which gives the number of subtrees after pairing. The initial number of subtrees is given by GT. One can observe that the number of subtrees is reduced to half in each iteration of SCGP. The number of nodes is significantly reduced in the first iterations of pairing; however, at some point it remains unchanged as seen in Fig. 7 for *bioav*, PPB and P3D. It means that a future improved version of our method could detect the iteration of pairing in which the number of nodes remains unchanged and skip the remaining pairing steps of the algorithm to save the computation time.

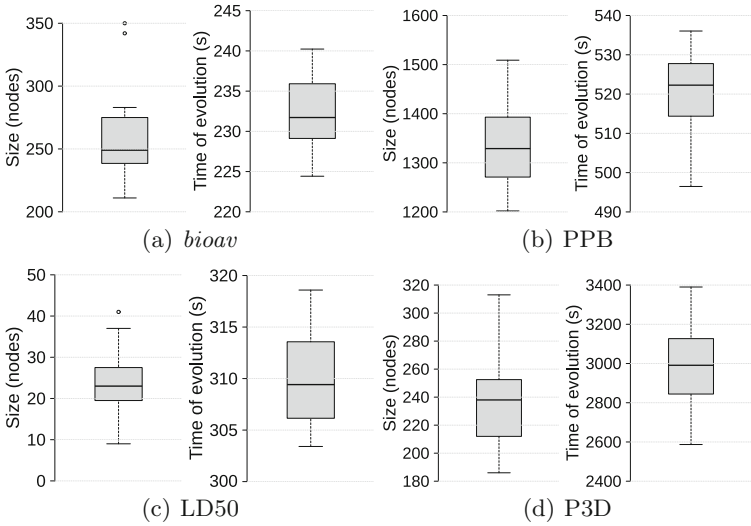


Fig. 6. Program sizes and time of evolution obtained with the final setup of SCGP.

Table 5. The number of nodes and the training and test (error) fitness for the best individuals obtained with GSGP and SCGP. As there is no acceptable solution created by SCGP for PPB, the best acceptable solution obtained in previous experiment ($PPB_{Tab.3}$) is reported.

	GSGP				SCGP			
	nodes	nodes (reduced)	f_{train}	f_{test}	nodes	f_{train}	f_{test}	accept.
bioav	22,285	9,863	19.455	26.938	224	19.39	26.69	4
PPB	22,907	10,291	4.312	30.768	1270	4.27	32.30	–
$PPB_{Tab.3}$					1723	4.26	26.80	1
LD50	21,568	9,934	1336.960	1495.621	13	1333.8	1417.6	12
P3D	5,764	3,992	3.981	3.999	186	3.98	3.98	16

Table 6. The number of subtrees after 1–10 iterations of pairing.

	Initial	1	2	3	4	5	6	7	8	9	10
bioav	911	456	228	114	57	29	15	8	4	2	1
PPB	892	446	223	112	56	28	14	7	4	2	1
LD50	921	461	231	116	58	29	15	8	4	2	1
P3D	259	130	65	33	17	9	5	3	2	1	–

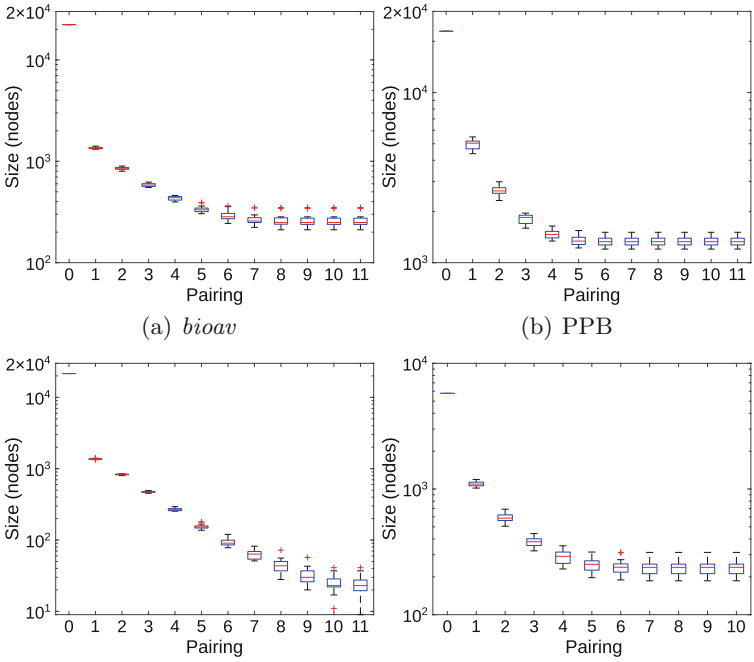


Fig. 7. The number of nodes in subtrees after pairing. Box plots constructed from 20 independent SCGP runs.

5 Conclusions

We proposed a CGP-based method capable of reducing the number of nodes in programs generated by GSGP. The obtained node reduction is 98.9% for *bioav*, 92.4% for PPB, 99.9% for LD50 and 96.7% for P3D. One SCGP run required hundreds of seconds for data sets containing hundreds of records (*bioav*, PPB, LD50) and thousands of seconds for P3D in which the data set contains ten thousands of records. The method can directly be used in GSGP utilizing a local search [11].

Our future work will be devoted to improving the key steps of SCGP (pairing strategies, fitness function for subtrees and termination conditions) and involving GSC into the process.

Acknowledgments. This work was supported by the Ministry of Education, Youth and Sports, under the INTER-COST project LTC 18053. The authors would like to thank Dr. Mauro Castelli for his support regarding the C++ framework for GSGP.

References

1. Moraglio, A., Krawiec, K., Johnson, C.G.: Geometric semantic genetic programming. In: Coello, C.A.C., Cutello, V., Deb, K., Forrest, S., Nicosia, G., Pavone, M. (eds.) PPSN 2012. LNCS, vol. 7491, pp. 21–31. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32937-1_3
2. Vanneschi, L., Silva, S., Castelli, M., Manzoni, L.: Geometric semantic genetic programming for real life applications. In: Riolo, R., Moore, J.H., Kotanchek, M. (eds.) Genetic Programming Theory and Practice XI. GEC, pp. 191–209. Springer, New York (2014). https://doi.org/10.1007/978-1-4939-0375-7_11
3. Martins, J.F.B.S., Oliveira, L.O.V.B., Miranda, L.F., Casadei, F., Pappa, G.L.: Solving the exponential growth of symbolic regression trees in geometric semantic genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2018, 15–19 July 2018, Kyoto, Japan, pp. 1151–1158. ACM (2018)
4. Miller, J.F., Job, D., Vassilev, V.K.: Principles in the evolutionary design of digital circuits - part I. Genet. Program. Evolvable Mach. **1**(1), 8–35 (2000)
5. Nguyen, Q.U., Pham, T.A., Nguyen, X.H., McDermott, J.: Subtree semantic geometric crossover for genetic programming. Genet. Program. Evolvable Mach. **17**(1), 25–53 (2016)
6. Castelli, M., Silva, S., Vanneschi, L.: A C++ framework for geometric semantic genetic programming. Genet. Program. Evolvable Mach. **16**(1), 73–81 (2015). https://doi.org/10.1007/s10710-014-9218-0
7. Miller, J.F.: Cartesian Genetic Programming. Springer, Berlin (2011). https://doi.org/10.1007/978-3-642-17310-3
8. Archetti, F., Lanzeni, S., Messina, E., Vanneschi, L.: Genetic programming for computational pharmacokinetics in drug discovery and development. Genet. Program. Evolvable Mach. **8**(4), 413–432 (2007). https://doi.org/10.1007/s10710-007-9040-z
9. Vanneschi, L.: An introduction to geometric semantic genetic programming. In: Schütze, O., Trujillo, L., Legrand, P., Maldonado, Y. (eds.) NEO 2015. SCI, vol. 663, pp. 3–42. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-44003-3_1
10. Dua, D., Karra Taniskidou, E.: UCI machine learning repository (2017). http://archive.ics.uci.edu/ml
11. Castelli, M., Trujillo, L., Vanneschi, L., Silva, S., Z.-Flores, E., Legrand, P.: Geometric semantic genetic programming with local search. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, 11–15 July 2015, Madrid, Spain, pp. 999–1006. ACM (2015)