Geometric Crossover in Syntactic Space

João Macedo[1,2](✉), Carlos M. Fonseca[2], and Ernesto Costa[2]

[1] ISR, Department of Electrical and Computer Engineering, University of Coimbra, 3030 290 Coimbra, Portugal
[2] CISUC, Department of Informatics Engineering, University of Coimbra, 3030 290 Coimbra, Portugal
{jmacedo,cmfonsec,ernesto}@dei.uc.pt

Abstract. This paper presents a geometric crossover operator for Tree-Based Genetic Programming that acts on the syntactic space, where each expression tree is represented in prefix notation. The proposed operator is compared to the standard subtree crossover on a symbolic regression problem, on the Santa Fe Ant Trail and on a classification problem. Statistically validated results show that the individuals produced using this method are significantly smaller than those produced by the subtree crossover, and have similar or better performance in the target tasks.

Keywords: Genetic Programming · Geometric operators · Crossover

1 Introduction

Geometric variation operators have been available for Genetic Algorithms (GA) for some time now [1]. They are representation-independent operators based on a distance on the search space interpreted as a metric space. The geometric description of the variation operators uses the notions of line segment (crossover) and ball (mutation). In the case of the crossover operator, the resulting offspring is on a shortest path, i.e., line segment, linking its parents. In the case of mutation, the resulting individual is in the neighbourhood of the original individual, i.e., within a ball centred on the individual and with a given radius, which defines the magnitude of the mutation. More formally, considering a given distance d defined over the search space, and provided that the parent individuals A and B are different, a geometric crossover operator will produce an offspring O such that $d(A, B) = d(A, O) + d(O, B)$.

Uniform crossover is an instance of a geometric operator devised for GAs. As an example, consider the binary strings: $A = 00000$ and $B = 11111$. One possible offspring generated by uniform crossover could be: $O = 10101$. Considering the Hamming distance as the metric we have, $d(A, O) = 3, d(O, B) = 2, d(A, B) = 5$. This example can be easily modified to accommodate other types of search spaces, the only requirement being the definition of an appropriate distance between individuals.

© Springer International Publishing AG, part of Springer Nature 2018
M. Castelli et al. (Eds.): EuroGP 2018, LNCS 10781, pp. 237–252, 2018.
https://doi.org/10.1007/978-3-319-77553-1_15

Moraglio [1] introduced abstract definitions of geometric crossover and mutation operators that are independent of the individuals' representation. One such operator is Geometric Uniform Crossover, *UX*. In this operator, all individuals between the parents have equal probability of being an offspring. Using this and other definitions, Moraglio showed that it is possible to devise geometric variation operators for different representations, e.g., binary strings, real value vectors and permutations.

Constructing a geometric variation operator for expression-trees, a structure commonly used in Genetic Programming (GP), is not straightforward, as it is not clear what a suitable distance would be. Moreover, making a small modification to the genotype of a GP individual may lead to a big change in its behaviour. This is known as the low-locality problem inherent to the representation of GP individuals [2].

Moraglio and Poli [3] have provided a theoretical study on how homologous crossover is geometric. When using homologous crossover, the topologies of both trees are compared, and the common rooted structures are found. Then, genetic material from the common regions is exchanged. One point crossover [4] is a special case of homologous crossover. With this operator, only one node is selected from the common region of each parent and the subtrees rooted by them are exchanged. Despite having been proposed some time ago, these operators never achieved much popularity. The fact that the alignment of the parent trees is a time-consuming task may be one of the reasons for their lack of usage. Another reason may be that, with extremely different trees, the common region may be very small, leading to very big syntactic changes which, together with the low-locality inherent to this tree representation, may have a great impact on the semantics of the offspring, possibly leading to their death.

Krawiec and Lichocki [5] proposed the Approximately Geometric Semantic Crossover (SX). They consider the semantics of a GP individual as the set of input-output mappings created by it. Based on this idea, they propose a binary variation operator that tries to approximate a geometric crossover in semantic space by producing an offspring that has a behaviour as close as possible to the linear combination of the semantics of its parents. This is done by applying multiple times a usual crossover operator to the parents, creating a set of candidate offspring. The semantics of the created offspring is assessed and compared to those of the parents. The resulting offspring is one with the closest semantics to both parents. This approach has several drawbacks. Firstly, it does not guarantee that the individuals created are on a shortest path between their parents. As the authors point out, there is a low probability of producing semantically geometric offspring, depending on the parents, the crossover operator, the terminal and function sets, and the size of the set of candidate offspring. Secondly, the evaluation of the many GP individuals produced usually becomes a time consuming process. Thus, the execution time of the algorithm is expected to increase proportionally to the size of the generated sets of candidate offspring.

Moraglio et al. [6] further developed the idea of geometric variation operators in semantic space, and later proposed Geometric Semantic Genetic Programming (GSGP), where the crossover operation consists of making a weighted average

of both parents. More formally, the crossover operation is defined by:

$$o = \alpha \cdot p_1 + (1 - \alpha) \cdot p_2,$$

where o denotes the offspring, p_1 and p_2 are the parent individuals, and α is a random value sampled uniformly from the interval $[0, 1]$. This operation guarantees that the semantics of the generated individual, i.e., its input-output mapping, is a blend of the semantics of both its parents. GSGP offers the advantage of, under certain conditions, inducing a unimodal fitness function over the semantic space, which makes the search process much easier. Also, by acting on the semantic space, geometric semantic crossover controls how much the behaviour of the individual is changed. Conversely due to the low locality of the tree-based representation, making modifications at the syntactic level often leads to large modifications to the semantics of the individuals, which may deem them unfit and prevent them from surviving into the next generations. However, GSGP is not without its faults, the most significant being the exponential growth of the individuals, which makes it unfeasible to employ this method for a large number of generations. Vanneschi et al. [7] propose a different, more efficient implementation that remedies this problem, but does not solve it.

A different approach to the development of geometric operators for GP would be to adopt a simpler individual representation and, like the traditional variation operators, act on the syntactic space. There are already some works on Evolutionary Algorithms (EAs) that use an alternative representation to encode the individuals. Brameier and Banzhaf [8] describe Linear Genetic Programming, which uses a linear representation of the individuals to evolve computer programs. Another work that represents the individuals linearly is Gene Expression Programming (GEP) [9]. The genotype of GEP individuals is represented by a string that can be decoded into more complex structures, such as expression trees, graphs and neural networks. The notation used for the genotypes is the sequence of nodes visited during a breadth-first traversal of the phenotype. The length of the genotypes is fixed and chosen a priori for all individuals. GEP is able to evolve different individuals using various types of functions, which may have different arities. In order to guarantee their validity, each genotype is divided into head and tail. While the head may contain functions and terminal symbols, the tail must contain only terminal symbols. The sizes of the head and tail are pre-computed to ensure that even in the presence of a head composed only of functions with maximum arity, the tail is long enough to generate a valid individual. A consequence of this is that there will often exist redundant genes at the end of the tail. As an example, consider that the genotype of an individual is Q*+−abcd, where the head contains Q*+−, and the tail contains abcd. Figure 1 depicts the phenotype of this individual. However, the genotype Q*+−abcd**efgh** would lead to the same phenotype, with the genes in bold not being decoded. The individuals are evolved using a set of variation operators that are not known to be geometric. The GEP author claims that the advantage of this method lies in using a simple representation that is easy to manipulate, and yet originates complex structures.

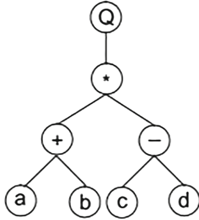


Fig. 1. Phenotype of an individual evolved by GEP. Figure extracted from [9].

In this paper, we propose a geometric crossover operator for GP that acts on the syntactic space. We use a linear, prefix representation of the GP individuals with no redundant genes. This way, we can avoid tree alignment algorithms and, instead, rely on string edit distances between the individuals. We compute the longest common subsequence of the parents, and use the result to make a controlled modification in one of the parents producing an offspring that lies in between those parents in the syntactic, i.e., genotype, space. The proposed operator is compared to the standard subtree crossover on a symbolic regression problem, on the Santa Fe Ant Trail and on a classification problem. Statistically validated results show that the individuals produced using this method are significantly smaller than those produced by subtree crossover, and have equivalent or better performance on the target tasks.

The rest of this paper is organised as follows: Sect. 2 describes the proposed recombination operator, Sect. 3 presents the experimental setup, Sect. 4 presents and discusses the experimental results obtained and Sect. 5 presents the conclusions and provides some insight into the future work.

2 Geometric Crossover on the Syntactic Space

We propose to perform a geometric crossover operation between two GP individuals in the syntactic space. The genotype of each individual is a string that encodes an expression tree in prefix notation. There are no redundant genes. The crossover operation aligns the genomes of the two parents and performs the necessary operations so that one becomes more similar to the other. In our implementation, the alignment is performed by computing the Longest Common Subsequence. The modifications are made by inserting or deleting pairs of symbols of different types, i.e., a terminal and a non-terminal symbol, or by deleting a symbol and inserting another one of the same type. The distance between the two parent individuals is the number of operations that convert one individual into the other.

A flow chart of the crossover operator is depicted on Fig. 2. We start by using a dynamic programming algorithm to compute the Longest Common Subsequence between two parent individuals, A and B, obtaining a matrix C. This matrix contains the information about the common and non-common genetic

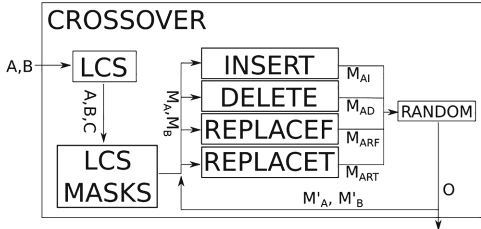


Fig. 2. Geometric crossover operator.

material to A and B. Using that information, Algorithm 1 (LCS_MASKS), constructs two modification masks M_A, M_B. These masks contain the aligned symbols from the longest common subsequence, along with blank spaces denoting the locations where insertions and deletions must be made, and the non-common symbols marked for deletion or insertion. Using these masks, it is possible to make a copy of the parent A more similar to parent B by inserting and/or deleting genetic material. The resulting offspring O is chosen randomly from the set of valid candidate individuals resulting from each of those operations. This can be repeated for a number of steps, in order to generate individuals farther from the first parent (and closer to the second one). In that case, the previously chosen individual takes the place of A and M_A and M_B are replaced by M'_A and M'_B, respectively, which are updated versions that reflect the operations performed.

2.1 String Edit Distance

The string edit distance is a family of metrics that reflect the number of operations needed to transform a string A into a string B. The Longest Common Subsequence (LCS) [10] is the longest possibly non consecutive subsequence that is common to both strings. From it, it is possible to compute the distance between the two strings, i.e., the number of operations required to modify the non-common symbols from the two strings. We refer the interested reader to [11], where a dynamic programming algorithm to compute the LCS is presented. This algorithm outputs a matrix C that holds in each position (i, j) the length of the longest common subsequence contained up to the character i of string A and character j of string B. After the algorithm terminates, a longest common subsequence can be obtained by going through the matrix C from the bottom right cell to the upper left cell. A symbol in position (i, j) is part of a longest common subsequence if $C[i][j] = C[i-1][j-1] + 1$ and $C[i][j] \neq C[i-1][j]$, $C[i][j] \neq C[i][j-1]$.

2.2 Crossover Operator

The proposed geometric crossover operator (GSynGP) works by modifying a copy of the first parent, A, to make it more similar to the second parent, B,

Algorithm 1. Modification masks generated from the Longest Common Subsequence.

```
1: function LCS_MASKS(A, B, C)
2:     M_A, M_B ← []
3:     i ← len(C) − 1
4:     j ← len(C[0]) − 1
5:     while i >= 1 or j >= 1 do
6:         if i > 0 and j > 0 and C[i − 1][j − 1] = C[i][j] then
7:             M_A ← get_symbol(A[i − 1], function_set)
8:             M_B ← get_symbol(B[j − 1], function_set)
9:             i ← i − 1
10:            j ← j − 1
11:        else if i > 0 and C[i − 1][j] = C[i][j] then
12:            M_A ← get_symbol(A[i − 1], function_set)
13:            M_B ←' '
14:            i ← i − 1
15:        else if j > 0 and C[i][j − 1] = C[i][j] then
16:            M_A ← ' '
17:            M_B ← get_symbol(B[j − 1], function_set)
18:            j ← j − 1
19:        else if i > 0 and j > 0 and C[i − 1][j − 1] = C[i][j] − 1 then
20:            M_A ← A[i − 1]
21:            M_B ← B[j − 1]
22:            i ← i − 1
23:            j ← j − 1
24:    return reverse(M_A), reverse(M_B)
```

by inserting and/or removing nodes while preserving the syntactic validity of the intermediate individuals. It starts by computing the LCS to determine the similarity between the two parent individuals (A, B). That information is contained in matrix C. Then, using matrix C, Algorithm 1 (LCS_MASKS) computes two modification masks, M_A and M_B, that contain the symbols in the longest common subsequence, as well as those that must be modified and blank spaces denoting locations that must be filled. In other words, Algorithm 1 computes an alignment of the two parent strings. The masks are created by going through matrix C, aligning the common genetic material and inserting blank spaces or markers in the positions where symbols must be added or removed from A. The markers are provided by function *get_symbol*, which returns the symbol passed as a parameter along with a prefix to denote whether it belongs to the terminal ($T_$) or function set ($F_$). As an example, consider two parent individuals A: $/ - */bcaa * aa$ and B: $*b/cb$. A possible LCS between them is $*bc$. The masks produced by Algorithm 1 are:

M_A: $F_/$, F_-, $*$, $F_/$, b, ' ', c, T_a, T_a, F_*, T_a, T_a
M_B: ' ', ' ', $*$, ' ', b, $F_/$, c, ' ', ' ', ' ', ' ', T_b

where, the symbols in M_A that have the prefixes $T_$ and $F_$ are not present in the individual B, and must be deleted. The symbols with those prefixes present in M_B represent genetic material that must be inserted into A. The locations for insertions are marked by blank spaces in M_A, and the locations for deletions are marked by blank spaces in M_B.

Algorithm 2. Geometric Syntactic Crossover Operator

1: **function** CROSSOVER(M_A, M_B)
2:     $candidates \leftarrow []$
3:     **if** 'F_' in M_A and 'T_' in M_A **then**
4:         $candidates \leftarrow candidates \cup delete(M_A, M_B, function_set)$
5:     **if** not' F_' in M_A and 'T_' in M_A **then**
6:         $candidates \leftarrow candidates \cup replaceT(M_A, M_B, function_set)$
7:     **if** 'F_' in M_B and 'T_' in M_B **then**
8:         $candidates \leftarrow candidates \cup insert(M_A, M_B, function_set)$
9:     **if** 'F_' in M_B and 'F_' in M_A **then**
10:         $candidates \leftarrow candidates \cup replaceF(M_A, M_B, function_set)$
11:     **return** $random(candidates)$

The remaining steps of the crossover operator, presented in Algorithm 2, consist simply in checking four conditions and performing the corresponding operations, where appropriate. The four possible operations are: inserting a function and a terminal (Algorithm 3), removing a function and a terminal (Algorithm 4), removing a terminal and inserting another one and removing a function and inserting another one (Algorithm 5). The operation of deleting a terminal and inserting another one is identical to what is presented in Algorithm 5, with the difference that $F_$ should read $T_$ and function_set should read terminal_set. The function *all_combinations*() returns all pairs of symbols to be tested in each case, that is, in Algorithm 3, the combinations of all function and terminal symbols

Algorithm 3. Insertion of a function and a terminal symbol

1: **function** INSERT(M_A, M_B)
2:     $combs \leftarrow all_combinations()$
3:     **while** $len(combs) > 0$ **do**
4:         $(f, t) \leftarrow random(combs)$
5:         $M_A[t] \leftarrow M_B[t]$
6:         $M_A[f] \leftarrow M_B[f]$
7:         **if** $check_indiv(M_A, function_set)$ **then return** M_A
8:         **else**
9:             $M_A[t] \leftarrow$' '
10:             $M_A[f] \leftarrow$' '
11:     **return** M_A

that are present in B and absent in A; in Algorithm 4, all pairs of terminal and function symbols that are present in A and absent in B and; for the operations of deleting and inserting symbols of the same type, all pairs of symbols of the desired type that are present in one parent and absent in the other one.

Algorithm 4. Deletion of a function and a terminal symbol

```
1: function DELETE(M_A, M_B)
2:     combs ← all_combinations()
3:     while len(combs) > 0 do
4:         (f, t) ← random(combs)
5:         v ← [M_A[f], M_A[t]]
6:         M_A[t] ←' '
7:         M_A[f] ←' '
8:         if check_indiv(M_A, function_set) then return M_A
9:         else
10:            M_A[t] ← v[1]
11:            M_A[f] ← v[0]
12:    return M_A
```

An individual is then selected from the set of valid generated individuals. These operations create an individual that is one step away from the first parent. In order to create offspring at different distances from each parent, Algorithm 2 may be iterated over a number of times, with the resulting individual of one iteration taking the place of A in the following iteration. For example, if the operator is applied twice, on the first iteration it will be applied to parents A and B, outputting an offspring O_1. In the second iteration, the offspring O_1 will take the place of A in the crossover, creating the individual O_2. In general, in an iteration where all operations are possible, the offspring created only has a 25% chance of becoming larger than its parent. That growth will only by 2 nodes and, at most, by one depth level.

An individual is valid if it can be converted into a valid expression tree, without any exceeding genes. It is also possible to test this validity without converting the string into the corresponding tree. Consider a counter c, that holds the number of necessary terminal symbols for an individual to be valid. For an empty string, $c = 1$, i.e., a terminal symbol is required in order to create a valid individual. To test an individual, start with $c = 1$ and go through the string. Increment c for each function symbol read, and decrement it for each terminal read. If c reaches 0 and there are still unread symbols, the individual has redundant genes and, thus, is invalid. On the other hand, if the string ends and $c > 0$, the individual can not be converted into a valid tree. In the given example, two valid candidates for offspring are:

1. $/ - */bcaaa$, obtained by removing the last $*$ and one of the marked a from A
2. $/ - */b/caaaa$, obtained by removing the last $*$ from A and inserting the $/$ from B

Algorithm 5. Deletion of a function and insertion of another function symbol

```
1: function REPLACEF(M_A, M_B)
2:     combs ← all_combinations()
3:     while len(combs) > 0 do
4:         (f_b, f_a) ← random(combs)
5:         if M_A[f_b] =' ' then
6:             M_A[f_b] ← M_B[f_b]
7:             v ← M_A[f_a]
8:             M_A[f_a] ← ' '
9:             if check_indiv(M_A, function_set) then return M_A
10:            else
11:                M_A[f_b] ←' '
12:                M_A[f_a] ← v
13:        else if 'F_' in M_A[f_b] then
14:            M_A[f_b] ← M_B[f_b]
15:        else
16:            v ← M_A[f_a]
17:            M_A[f_a] ← M_B[f_b]
18:            if check_indiv(M_A, function_set) then return M_A
19:            else
20:                M_A[f_a] ← v
21:    return M_A
```

Table 1. Parameters of the SGP.

Parameter	Value
Population size	400
Elite size	1
Tournament size	10
Maximum tree depth	10
Generations	1000
Crossover rate	0.7
Mutation rate	0.3
Number of immigrants	120

3 Experimental Setup

In order to assess the usefulness of the proposed approach, we performed 30 independent runs on a Symbolic Regression problem, on the Santa Fe Ant Trail and on a Classification problem, comparing the performance of the proposed approach to that of Standard Genetic Programming (SGP). In the following, we shall refer to the geometric syntactic approach as GSynGP.

3.1 Standard Genetic Programming Algorithm

We implemented a version of SGP using subtree crossover and two types of mutation: point mutation and subtree crossover with a randomly generated individual. Point mutation selects one symbol from the genotype and replaces it by another of the same type. The initial population is created using the method known as ramped-half-and-half [12]. The parents are selected using tournaments and survivor selection is generational, with an elite individual. In order to maintain a diverse population throughout the run, at each generation a set of immigrant individuals are introduced into the population. These immigrants have a 50% chance of being elitist or randomly generated. Elitist immigrants are mutated copies of a good quality individual that is selected from the population by tournament. The random immigrants are generated using the ramped-half-and-half method. The parameters used for this algorithm are presented on Table 1.

3.2 Geometric Syntactic Approach

This algorithm uses the same parameters as those used for the SGP, differing only in the crossover operator employed, which has been described in Algorithm 2. Due to the low-locality problem inherent to expression trees in GP, we are not interested in making modifications that are too disruptive. However, we are still interested in generating individuals that have different distances to each parent. For these reasons, the crossover operator is performed for a random number of steps, which is uniformly sampled from $\{1, 2, 3\}$.

3.3 Symbolic Regression

Dataset. The performance of the two algorithms is assessed on a dataset generated using Eq. 1, as proposed by Keijzer [13]. The dataset contains 50 randomly generated points, with x_1, x_2 sampled uniformly from the interval $[-10, 10]$.

$$y = x_1^4 - x_1^3 + \frac{x_2^2}{2} - x_2 \tag{1}$$

Terminal and Function Sets. The SGP and the GSynGP use the same terminal and function sets. For this problem, the terminal set is composed only of the variables from the dataset (i.e., x_1, x_2) and the function set is composed of the basic arithmetic functions (i.e., $+, -, *, /$). / stands for protected division, where $x/0 = 1$.

Fitness Evaluation. The fitness of each individual was assessed using the Mean Squared Error (MSE), as defined in Eq. 2, thus making this a minimisation problem.

$$MSE = \frac{\sum_{i=1}^{N} (\hat{Y}_i - Y_i)^2}{N} \tag{2}$$

where N is the number of samples, $\hat{Y}_i$ is the ith predicted value and Y_i is the corresponding target value.

3.4 Santa Fe Ant Trail

The Santa Fe Ant Trail is a path planning benchmark problem where an artificial ant must follow a deceptive trail collecting food pellets. We used the traditional version of this problem, as used by Koza in [12]. The map is a 32×32 toroidal grid with 89 food pellets. The ant starts facing east, at the upper left cell. The simulation ends when the ant completes 400 moves.

Terminal and Function Sets. The SGP and the geometric approach use the same terminal and function sets. For this problem, the terminal set is composed of the basic actions of the ant $\{left, right, move\}$, where $left$ and $right$ rotate the ant 90^0 in each direction and $move$ makes it move forward one cell. The function set is composed of the functions $ifFoodAhead$ and $Progn2$. Both functions have an arity of 2. Function $ifFoodAhead$ checks if there is a food pellet directly in front of the ant and, if there is, executes its first argument, otherwise executes the second argument. $Progn2$ is a progression function that executes both arguments in sequence.

Fitness Evaluation. The fitness of each individual is measured as the number of food pellets eaten by the ant, within the 400 steps limit. Thus, this is a maximisation problem.

3.5 Classification

In order to understand how our approach performs in real world problems, we decided to test it in a classification problem with real data. The chosen dataset is the Wisconsin Breast Cancer [14], available at the UCI repository. It is a binary classification problem that aims at determining whether a sample, described by 30 features, represents a benign or malign tumour. This dataset has 569 samples, from which, on the beginning of each run, the algorithm chose 70% to be used to evolve the individuals. As the dataset was unbalanced, prior to this split a balancing of the data was made, by randomly discarding samples of the larger class until both had an equal amount of examples. Thus, the balanced dataset was left with 424 samples, from which 296 were used in the evolutionary process.

Terminal and Function Sets. As before, both the SGP and the GSynGP use the same terminal and function sets. The terminal set is composed only of the variables from the dataset (i.e., $x_1, x_2, ..., x_{30}$) and the function set is composed of the basic arithmetic functions (i.e., $+, -, *, /$).

Fitness Evaluation. The performance of each individual was measured with the F1-Score, a commonly used criterion in classification problems. As the Santa Fe Ant Trail, this is a maximisation problem.

4 Experimental Results

For each problem, we characterise each algorithm according to three features: the fitness of the best individual at the end of each run, and the average depth and average number of nodes of the individuals in the population, also at the end of each run. The results were validated using statistical tests, at a significance level $\alpha = 0.05$. The results of those tests are presented on Tables 2 and 3.

4.1 Symbolic Regression

We start by applying the Kolmogorov-Smirnov test to the data, in order to be able to decide whether we should apply parametric or non-parametric tests. The results of this test show that the data for the average depth and average number of nodes of the individuals of both algorithms follow normal distributions, with $p > 0.05$, whereas the fitness data follows non-normal distributions ($p < 0.05$). As we have two sets of paired samples per feature, we apply the Wilcoxon Rank-Sum test to the data that follow non-normal distributions, i.e., to the fitness data, and the Paired Samples T-Test to the data of the average number of nodes and average depth. Both tests were applied as SGP-GSynGP. For the fitness data, the Wilcoxon test shows that there are no statistically significant differences between the performances of the individuals evolved by each

Table 2. Kolmogorov-Simrnov test applied to the data of the three problems.

		Symbolic regression			Santa Fe Ant Trail			Classification		
		Depth	Size	Fitness	Depth	Size	Fitness	Depth	Size	Fitness
SGP	Z	0.88	0.74	1.64	0.65	0.47	1.51	0.79	0.93	0.75
	p	0.416	0.648	0.006	0.794	0.979	0.014	0.556	0.358	0.628
GSynGP	Z	0.83	0.93	1.37	2.00	2.26	2.41	1.40	1.74	0.75
	p	0.498	0.358	0.035	0.0	0.0	0.0	0.028	0.003	0.628

Table 3. Wilcoxon applied to the fitness of the Symbolic Regression problem, to all data of the Santa Fe Ant Trail and to the average size and depth data of the Classification. Paired Samples T-Test applied to the data of the average depth and size of the Symbolic Regression problem and to the fitness data of the Classification problem.

	Symbolic regression			Santa Fe Ant Trail			Classification		
Wilcox. (Dep. T)	Depth	Size	Fitness	Depth	Size	Fitness	Depth	Size	Fitness
Z (t)	14.11	3.40	−1.22	−4.78	−4.78	−3.15	−4.62	−4.78	−1.16
p	0.0	0.0	0.221	0.0	0.0	0.002	0.0	0.0	0.257
P Ranks ($\overline{SGP}$)	119.34	4149.04	15	30	30	3	28	30	0.96
N Ranks ($\overline{GSynGP}$)	13.12	108.02	15	0	0	18	2	0	0.97
Effect size	0.934	0.931	−0.223	−0.873	−0.873	−0.575	−0.843	−0.873	0.211

algorithm. However, the Paired Samples T-Test revealed that the geometric approach evolved significantly smaller individuals, both in depth and in number of nodes (both $p = 0.0$), with large effect sizes of respectively 0.934 and 0.931. Analysing the average values, SGP evolved individuals with an average depth of 119.34 levels and an average size of 4149.04 nodes. The geometric syntactic approach, on the other hand, evolved much smaller individuals, with an average depth of 13.12 levels (only 3.12 levels higher than the maximum depth of the initial individuals) and an average size of 108.02 nodes.

4.2 Santa Fe Ant Trail

This section describes the analysis of the data collected on the Santa Fe Ant Trail. The Kolmogorov-Smirnov test yielded $p < 0.05$ for the data of all three features of the geometric approach. For that reason, we applied the Wilcoxon test to the data of the three features, in a SGP-GSynGP manner. It yielded $p = 0.0$ for the two features related to the size of the evolved individuals, leading us to conclude that the geometric approach evolved significantly smaller individuals, with a large effect size of -0.873. In fact, the individuals evolved by the SGP had an average depth of 107,60 levels and an average size of 3450.31 nodes, while those evolved by the GSynGP had an average depth of 10.94 and an average size of 61.05 nodes. Regarding the fitness of the best individuals, the Wilcoxon test output a p-value of 0.002, which lets us know that there are statistically significant differences between the performance of the individuals of each algorithm. Moreover, the 18 negative ranks found by this test lead us to conclude that the geometric approach evolved individuals that perform significantly better in this task than those evolved by the SGP, with a large effect size of -0.575. On average, the individuals evolved by the SGP collected 77.87 food pellets, while those evolved by the GSynGP collected 86.77, out of the 89 available in the world.

4.3 Classification

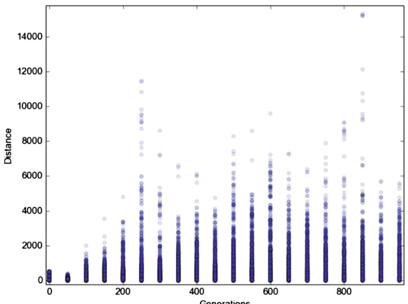
In the classification problem, the results of the Kolmogorov-Smirnov test do not allow us to reject the null hypothesis that the fitness data follow normal distributions. For that reason, we applied the Paired Samples T-Test which revealed no statistically significant differences, with the individuals evolved by the SGP achieving an average F1-Score of 0.96 and those evolved by the GSynGP achieving 0.97. Carrying on to assess the average size of the individuals in the last populations of each algorithm, the results of the Kolmogorov-Smirnov, allow us to assume that the data of the SGP follows a normal distribution, while the data for the GSynGP does not. For that reason, we applied the Wilcoxon test, which yielded significant differences (p=0.00) and, with all ranks being positive, it shows that the individuals evolved by the SGP are significantly larger, with an average number of nodes of 5278.39, while those evolved by the GSynGP have an average size of 283.98 nodes. Finally, the Kolmogorov-Smirnov test yielded the same results for the average depth as it did for the average size. Applying a Wilcoxon test to the

depth data, we conclude that the individuals evolved by the SGP are significantly deeper (p=0.0), with a large effect size of −0.843. The individuals evolved by the SGP have an average depth of 122.54 levels, while those evolved by the GSynGP have an average depth of 26.86 levels.

4.4 Population Diversity

We now focus on studying the diversity of the individuals from the population in each generation. We repeated the experiments without using immigrants, as they are an artificial method of increasing the population diversity. Due to space and time constraints, we focused on a single benchmark problem, the Santa Fe Ant Trail, and reduced the population size to 50 individuals. The tournament size was reduced to 3, as this is expected to increase the population diversity. The other parameters remained unchanged from the previous experiments.

Figure 3 presents the distances between each pair of parent individuals evolved by the SGP (left) and the GSynGP (right), sampled with a 50 generations period. Each distance is represented by a blue circle with high transparency. Thus, darker circles represent many pairs of parents with equal distances. At the beginning of the experiments, both algorithms present similar diversity (note the different scales). However, over time, the SGP seems to achieve a much greater diversity than the GSynGP. This is due to bloat. Bloated populations contain individuals with more diverse genotypes, but that diversity does not necessarily transfer into different behaviours. Moreover, over the entire run there are very large individuals in the population that result in big differences between the parents. However, these individuals are usually unable to survive many generations, leading the population to converge on a set of more similar individuals. The GSynGP behaves differently, gradually converging its population into a good quality area of the search space, with the most different individuals being phased out over the generations. However, this does not necessarily mean that the population has completely lost its diversity, as the parents of the last population still have an average distance of approximately 6.8 operations and a



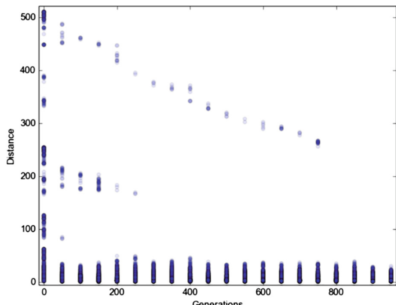


Fig. 3. Distances between each pair of parent individuals of the SGP (left) and GSynGP (right), sampled with a period of 50 generations.

standard deviation of 5.8. The bloat present in the populations of the SGP leads to more diverse populations in the genotype space, with the pairs of parents of the last population having an average distance of 1151 operations and a standard deviation of roughly 1095.3.

5 Conclusions

This work presented a novel geometric crossover operator that acts on the syntactic space of expression trees. The method was implemented and compared to SGP on problems from the domains of symbolic regression, path planning and classification. Our approach was able to consistently evolve smaller individuals than the SGP, both in size and depth. This reduction in size of the individuals does not imply a loss in quality, as our approach outperformed the SGP in the only test problem where there were statistically significant differences. The diversity experiments showed that the geometric operator led the population into a good quality region of the search space, without completely loosing its diversity. SGP seemed to have a much more diverse population, but that diversity was due to bloat. Future work includes adapting this approach to function sets containing symbols with different arities. Moreover, the crossover implementation should be improved to avoid having to test individuals after each insertion and/or deletion pair before a more thorough experimental study is carried out.

Acknowledgement. This article is based upon work from COST Action CA15140: Improving Applicability of Nature-Inspired Optimisation by Joining Theory and Practice (ImAppNIO), supported by COST (European Cooperation in Science and Technology), www.cost.eu. Support by national funds through the Portuguese Foundation for Science and Technology (FCT) and by the European Regional Development Fund (FEDER) through COMPETE 2020 – Operational Program for Competitiveness and Internationalization (POCI) is also acknowledged. J. Macedo acknowledges the Portuguese Foundation for Science and Technology for Ph.D. studentship SFRH/BD/129673/2017.

References

1. Moraglio, A.: Towards a Geometric Unification of Evolutionary Algorithms. Ph.D. thesis, Department of Computer Science, University of Essex (2007)
2. Galván-López, E., McDermott, J., O'Neill, M., Brabazon, A.: Defining locality as a problem difficulty measure in genetic programming. In: Genetic Programming and Evolvable Machines (2011)
3. Moraglio, A., Poli, R.: Geometric landscape of homologous crossover for syntactic trees. In: 2005 IEEE Congress on Evolutionary Computation (2005)
4. Poli, R., Langdon, W.B.: Genetic programming with one-point crossover. In: Chawdhry, P.K., Roy, R., Pant, R.K. (eds.) Soft Computing in Engineering Design and Manufacturing. Springer, London (1998). https://doi.org/10.1007/978-1-4471-0427-8_20

5. Krawiec, K., Lichocki, P.: Approximating geometric crossover in semantic space. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (2009)
6. 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
7. Vanneschi, L., Castelli, M., Manzoni, L., Silva, S.: A new implementation of geometric semantic GP and its application to problems in pharmacokinetics. In: Krawiec, K., Moraglio, A., Hu, T., Etaner-Uyar, A.Ş., Hu, B. (eds.) EuroGP 2013. LNCS, vol. 7831, pp. 205–216. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37207-0_18
8. Brameier, M., Banzhaf, W.: Linear Genetic Programming. Springer, Boston (2007). https://doi.org/10.1007/978-0-387-31030-5
9. Ferreira, C.: Gene expression programming in problem solving. In: Roy, R., Köppen, M., Ovaska, S., Furuhashi, T., Hoffmann, F. (eds.) Soft Computing and Industry: Recent Applications. Springer, London (2002). https://doi.org/10.1007/978-1-4471-0123-9_54
10. Paterson, M., Dančík, V.: Longest common subsequences. In: Prívara, I., Rovan, B., Ruzička, P. (eds.) MFCS 1994. LNCS, vol. 841, pp. 127–142. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58338-6_63
11. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, Cambridge (2001)
12. Koza, J.R.: Genetic programming: On the Programming of Computers by Means of Natural Selection. MIT press, Cambridge (1992)
13. Keijzer, M.: Improving Symbolic regression with interval arithmetic and linear scaling. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 70–82. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36599-0_7
14. Lichman, M.: UCI Machine Learning Repository (2013). http://archive.ics.uci.edu/ml