Geometric Semantic Grammatical Evolution

Alberto Moraglio, James McDermott and Michael O'Neill

Abstract Geometric Semantic Genetic Programming (GSGP) is a novel form of Genetic Programming (GP), based on a geometric theory of evolutionary algorithms, which directly searches the semantic space of programs. In this chapter, we extend this framework to Grammatical Evolution (GE) and refer to the new method as Geometric Semantic Grammatical Evolution (GSGE). We formally derive new mutation and crossover operators for GE which are guaranteed to see a simple unimodal fitness landscape. This surprising result shows that the GE genotype-phenotype mapping does not necessarily imply low genotype-fitness locality. To complement the theory, we present extensive experimental results on three standard domains (Boolean, Arithmetic and Classifier).

1 Introduction

Geometric Semantic Genetic Programming (GSGP) is a novel form of Genetic Programming (GP), introduced by Moraglio et al. [1]. In GSGP, search operators act on the syntax of the programs but can be understood as acting directly on the underlying semantics of programs: mutation and crossover produce offspring which are, respectively, semantically close to and semantically intermediate between their parents. Specific GSGP operators for Boolean, Regression and Classification domains have been derived [1] and have a simple form. This is possible because the mapping from genotypes to semantics in these GP domains is simple, not complex as was widely believed before GSGP. Furthermore, the fitness landscape seen by GSGP is

Alberto Moraglio
University of Exeter, UK. e-mail: A.Moraglio@exeter.ac.uk

James McDermott
University College Dublin, Ireland. e-mail: james.mcdermott2@ucd.ie

Micheal O'Neill
University College Dublin, Ireland. e-mail: m.oneill@ucd.ie

always a simple unimodal landscape, and its search performance is provably good on large classes of problems [2, 3, 4].

GE [5] is a successful form of GP that represents programs indirectly as integer lists. Phenotypes are obtained by using the integers of the genotype to select among alternatives in the grammatical rules. One benefit of this indirect encoding is that it simplifies the application of search to different programming languages and constrained structures. A common criticism of GE is that because of the rather complex developmental genotype-phenotype mapping, search operators can be disruptive to both syntax and semantics, e.g. low locality of the genotype-phenotype mapping [6].

The purpose of the current chapter is to extend the ideas of GSGP to GE, giving Geometric Semantic Grammatical Evolution (GSGE). The remainder of the chapter is organised as follows. In Section 2, GSGP itself is reviewed. In Section 3, we describe theoretical requirements for translating GSGP concepts to GE, and in Section 4 we use these to derive new geometric semantic search operators for GE, and prove their properties for three domains (Boolean, Arithmetic, and Classifier). We give also a general recipe to derive GSGE operators from GSGP operators for any domain. In Section 5, we present an efficient implementation of GSGE (the size of the solutions grows only linearly even when using crossover). In Section 6, we present extensive experimental results and analysis. In Section 7, we provide a discussion, and in Section 8 a summary of the chapter.

2 Geometric Semantic Genetic Programming

Traditional genetic programming ignores the *meaning* of programs, as the search operators it employs act on their syntactic representations, regardless of their semantics. E.g., subtree swap crossover is used to recombine functions represented as parse trees, regardless of whether trees represent Boolean expressions, arithmetical functions, or classifier programs. While this guarantees the production of syntactically well-formed expressions, why should such a *blind* syntactic search work well for different problems and across domains? In the end, it is the meaning of programs that determines how successful search is at solving the problem.

The semantics of a program can be formally defined in a number of ways. It can be a canonical representation, so that any two programs with the same semantics or behaviour have the same canonical representation (e.g., Binary Decision Diagrams for Boolean expressions). It can instead be a description of program behaviour in a logical formalism, as used in formal methods. In the context of black-box search, it may be argued that the semantics of a program is just its fitness. Finally, semantics can be defined as the mathematical function computed by a program, i.e., the set of all possible input-output pairs making up the computed function. In practice, in GP, it is calculated over a restricted set of input-output pairs, and this is the definition we use in this paper.

In the literature, there are a number of works using the semantics of programs to improve GP. As many GP individuals may encode the same function, some researchers use canonical representations of functions to enforce semantic diversity by discarding individuals of duplicate semantics, in initialization [7, 8], crossover, and mutation [9, 10]. Nguyen et al. [11] measure semantic distance between individuals as distance between their outputs for the same set of inputs. This distance is used to semantically bias the search operators: mutation rejects offspring that are not sufficiently semantically similar to the parent, and crossover swaps only semantically similar subtrees between parents. Krawiec et al. [12, 13] have also used semantic distance to propose a crossover for GP trees that is approximately geometric [14, 15] in the semantic space. Interestingly, the fitness landscape induced by this operator has perfect fitness-distance correlation. The operator was implemented approximately by using a traditional crossover, generating a large number of offspring, and accepting only offspring semantically intermediate to their parents.

While the semantically aware methods above produce overall superior performance to traditional methods, they are *indirect*: search operators are implemented by acting on the syntax of the parents to produce offspring, which are accepted only if some semantic criterion is satisfied. This has two drawbacks: (i) these implementations are very wasteful, as they are heavily reliant on trial and error; (ii) they do not provide insights on how syntactic and semantic searches relate to each other. Would it then be possible to *directly* search the semantic space of programs? More precisely, would it be possible to build search operators that, acting on the syntax of the parent programs, produce offspring that are *guaranteed* by construction to respect some semantic criterion or specification? Krawiec et al. [12, 13] argued that due to the complexity of the genotype-phenotype mapping in GP, a direct implementation of exact semantic operators is probably impossible.

However, GSGP [1] shows that the genotype-phenotype (syntax to semantics) map of commonly considered GP domains is, in an important sense, easy — not complex. GSGP gives *exact* geometric semantic crossovers and mutations for different problem domains (Boolean, Arithmetic, Classifier). By construction these search operators see a simple unimodal fitness landscape for any problem in these domains [1].

2.1 Geometric Semantic Operators

A search operator $CX : S \times S \to S$ for a search space S is a *geometric crossover* [14, 15] w. r. t. the metric d if for any choice of parents $T1, T2 \in S$, each offspring $T3 = CX(T1, T2)$ is in the d-metric segment between parents, that is $d(T1, T3) + d(T3, T2) = d(T1, T2)$. A search operator $M : S \to S$ is a *geometric ε-mutation* w. r. t. the metric d if for any choice of the parent $T1$, each offspring $T2 = M(T1)$ is in the d-metric ball of radius ε centered in the parent, that is $d(T1, T2) \leq \varepsilon$. Suppose (as is typical) that the fitness function can be written as a distance $F(T) = d(O(T), t)$ between the output vector $O(T)$ of the program $T \in S$ on a fixed vector of inputs, and a target output vector t on the same inputs. Then the *semantic distance SD* between two programs $T1, T2 \in S$ is defined as the distance between their cor-

```
                                 AND
                          T1 =   /   \
              OR               X1   X2                    OR
            /    \                                      /    \
         AND      AND            OR                   AND    X3
T3 =    /   \    /   \    T2 =   /  \       T3 =     /   \
       T1   TR  NOT  T2        X2  X3              AND   NOT
                 |                                /   \   |
                TR              NOT              X1   X2  X3
                          TR =    |
                                X3
```

Fig. 1 Left: Semantic Crossover scheme for Boolean Functions; Centre: Example of parents (T1 and T2) and random mask (TR); Right: Offspring (T3) obtained by substituting T1, T2 and TR in the crossover scheme and simplifying.

responding output vectors $O(T1)$, $O(T2)$, measured with the metric d. That is, $SD(T1,T2) = d(O(T1),O(T2))$. *Geometric semantic operators* are operators on the space of functions which are geometric with respect to metric *SD*. E.g., geometric semantic crossover on Boolean functions returns offspring Boolean functions such that the output vectors of the offspring are in the Hamming segment between the output vectors of the parents.

This is however only an abstract specification of geometric semantic search operators. We require an algorithmic characterization. Note that there is a different type of geometric semantic crossover for each choice of space S and distance d. Consequently, there are different semantic crossovers for different GP domains. In the following, we provide algorithmic definitions of geometric semantic operators for Boolean, Arithmetic and Classification domains. A formal treatment and explicit derivations have been previously given [1].

Boolean crossover: Given two parent Boolean functions $T1,T2$, the geometric semantic crossover is the recombination that returns the offspring Boolean function $T3 = (T1 \wedge TR) \vee (\overline{TR} \wedge T2)$ where TR is a randomly generated Boolean function (see Fig. 1). TR is effectively a crossover mask, choosing a point in the semantic space intermediate to T1 and T2. **Boolean mutation**: Given a parent function T, the mutation SGMB returns the offspring $TM = T \vee M$ with probability 0.5 and $TM = T \wedge \overline{M}$ with probability 0.5 where M is a random *minterm* of all input variables. (A minterm is a term consisting of the product of all variables, each either negated or non-negated.)

Arithmetic crossover: Given two parent functions $T1,T2$, the geometric semantic crossover is the recombination that returns the real function $T3 = (T1 \cdot TR) + ((1-TR) \cdot T2)$ where TR is a random real constant in $[0,1]$. **Arithmetic mutation**: Given a parent function T, the mutation SGMR with mutation step *ms* returns the real function $TM = T + ms \cdot (TR1 - TR2)$ where $TR1$ and $TR2$ are random real functions.

Classifier crossover: Given two parent classifiers `T1,T2`, with symbols as inputs (*IS*) and outputs (*OS*), the geometric semantic crossover is the recombination that returns the offspring classifier `T3 = IF CONDR THEN T1 ELSE T2` where `CONDR` is a random condition (e.g. of the form $X_i == s$ where $s \in IS$). **Classifier mutation**: Given a parent classifier `T`, the mutation SGMP returns the offspring classifier `TM = IF CONDR THEN OUTR ELSE T` where `CONDR` is a condition which is true only for a single random setting of all input parameters, and `OUTR` is a random output symbol. The offspring can be expressed as nested `IF-THEN-ELSE` statements with simple conditions of a single input parameter each.

3 Foundations for Geometric Semantic Operators for GE

In this section, we first introduce the concept of *compositional semantics*, then we show that the GE mapping is compositional, and finally equipped with this we provide a formal recipe to derive geometric semantic operators for the GE encoding.

3.1 Compositional Semantics

In both linguistics – the study of natural languages – and theory of programming languages, *compositional semantics* refers to a relation between syntax of the sentences in a language and their semantics. The principle of compositional semantics states that the meaning (semantics) of a sentence (syntax) can be derived by combining the meanings of its sub-sentences. For example, the meaning of the sentence $S = \texttt{A and B}$ is $[S] = [\texttt{A and B}] = [\texttt{and}]([\texttt{A}],[\texttt{B}])$, where $[]$ is a function that maps a syntactic element to its meaning. This is a natural relation that holds for most languages, natural or artificial.

The relation between syntax and semantics in GSGP is compositional. Syntactically, geometric semantic crossovers plug parent trees `T1` and `T2` into a recombination tree `XT` to obtain an offspring tree `T3`. We are allowed to write this operation as `T3 = XT(T1, T2)` and interpret it as a functional composition because the syntactic operation of plugging the structures `T1` and `T2` in the structure `XT` is mirrored by the semantic operation of function composition of the function XT on the functions $T1$ and $T2$ producing the function $T3$, i.e., $[\texttt{T3}] = [\texttt{XT(T1, T2)}] = [\texttt{XT}]([\texttt{T1}],[\texttt{T2}])$. That is, geometric semantic crossovers are compositional. In contrast, traditional subtree swap crossover is not compositional. Formally this crossover could be similarly written as `T3 = XO(T1, T2)` denoting that the offspring structure `T3` can be obtained by some syntactic operation `XO` on the structures `T1` and `T2`. However, this time the semantics of `T3` cannot be written as `[T3] = [XO]([T1], [T2])` as the semantics of the operation `XO` (swapping two subtrees) is inherently linked to the syntactic representation of `T1` and `T2`, and cannot be defined solely on their semantics.

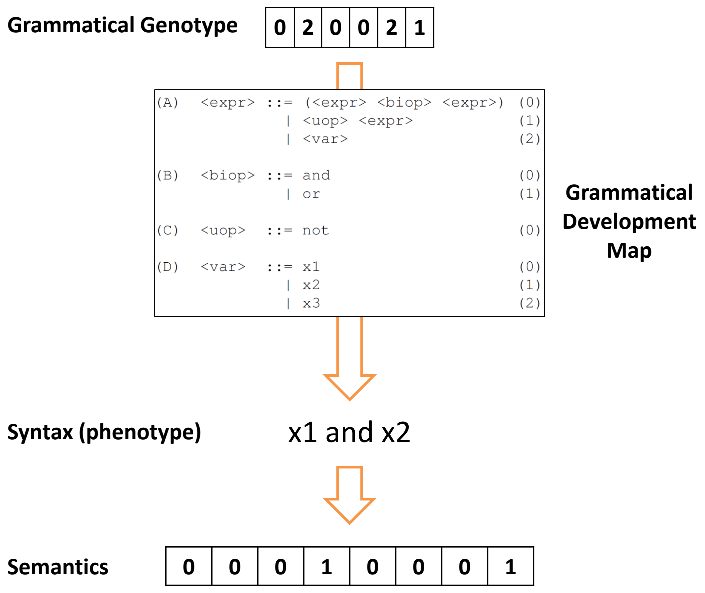
An immediate consequence of the semantic compositionality of GSGP is that, as the semantics of the offspring depend solely on the semantics of their parents, and not on their syntactic representations, functions and geometric semantic operators acting on these can also be equivalently represented in a form or language other than trees, *if it respects semantic compositionality*.

We will show that the genotype-phenotype mapping in GE is compositional, i.e., by stringing together linear representations of parents, we get the corresponding linear representations of the offspring.

3.2 Compositionality of the GE mapping

In Section 4, we will introduce simple GE search operators for several domains which are semantically geometric, i.e. perfectly well-behaved in terms of semantic effects. Given the non-trivial developmental encoding of GE, it is surprising that these operators are at all possible, especially in a simple form. In this section, we present a theory that explains rigorously how this is possible. The gist of the argument is as follows. We will observe that the GE developmental process mapping naturally preserves (compositional) modularity: phenotypic modules (derivation subtrees) correspond to genotypic modules (sublists). Together with a compositional interpretation of the geometric semantic operators, this implies the existence of a *genotypic* crossover/mutation scheme (on integer lists) equivalent to a *phenotypic* crossover/mutation scheme (on derivation trees), which is in turn equivalent to the GSGP crossover/mutation scheme (on GP trees): that is, an implementation of GSGP geometric semantic operators for GE. These considerations apply to the domains for which GSGP operators were derived by Moraglio et al. [1] (Boolean, Regression and Classification) and may extend to GSGP operators in other domains [16].

Let us now briefly review the GE genotype-phenotype mapping. Figure 2 illustrates the mapping. The genotype encoding a solution is the vector at the top. The corresponding derivation tree (not shown) is obtained through depth-first traversal of the grammar, using the genotype to select among multiple alternatives in the rules. The derivation tree is produced incrementally: at each step, the next gene (integer) in the genome is used to select the expansion for the left-most non-terminal node in the developing derivation tree. The value of each gene is taken modulo the number of available choices in the grammar for this non-terminal. When there are no non-terminal nodes left to expand, the derivation tree is complete. (In early versions of GE, a "wrapping" method was used, that is if the genotype has been exhausted and derivation is not finished, then indexing "wraps around" to the beginning of the genotype. Alternatively, it may be the case that derivation is completed before the genotype is exhausted. In this case, extra genes are simply ignored. The operators presented in the next section avoid these complications by design.) The phenotype (a string representing a program) is then extracted from the derivation tree by reading the derivation tree leaf nodes from left to right. Finally, the vector at the bottom of Fig. 2 is the semantics of the phenotype, that is the vector of the outputs of the



```
(A)   <expr> ::= (<expr> <biop> <expr>) (0)
               | <uop> <expr>           (1)
               | <var>                  (2)

(B)   <biop> ::= and                    (0)
               | or                     (1)

(C)   <uop>  ::= not                    (0)

(D)   <var>  ::= x1                     (0)
               | x2                     (1)
               | x3                     (2)
```

Fig. 2 Grammatical Evolution Genotype-Phenotype Mapping.

program for all possible combinations of inputs (or for some subset, depending on the domain).

To show that the GE mapping is semantically compositional, we will look more closely at several derivation trees. Figures 3, 4 and 5 show the derivation trees and the genotypes (bottom) for the expressions `x1 and x2`, `x2 or x3` and `not x3` respectively, obtained using the grammar in Figure 2. The number annotating each non-terminal node of the derivation tree identifies the grammatical production that was used to generate its child nodes out of the available applicable productions. For example, the number (0) annotating the root node (`expr`) of the derivation tree in Figure 3 signifies that its child nodes (`expr`, `biop` and `expr`) were obtained by selecting production rule 0 in the grammar in Figure 2, out of those whose LHS is `expr`. The choice of production rule 0 for the root node is dictated by the 0 as first entry of the genotype. The phenotype `x1 and x2` is just the terminal nodes of the derivation tree, read from left to right.

Let us now make three observations that together will show the semantic compositionality of the GE mapping, and provide a formal recipe to derive search operators for the grammatical genotype equivalent to the geometric semantic operators.

Observation 1: The derivation tree is effectively the *parse tree* of the given expression w.r.t. the given grammar. The *parsing* of a sentence w.r.t. a given grammar is the

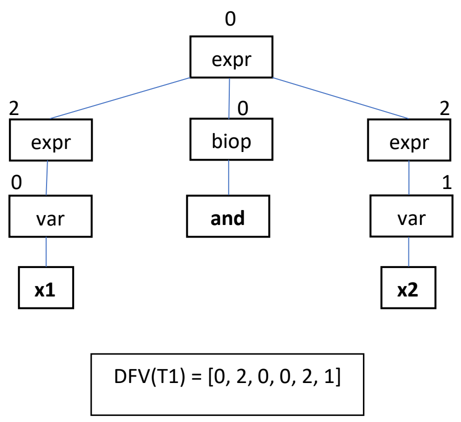


Fig. 3 Derivation tree for the expression `x1 and x2`.

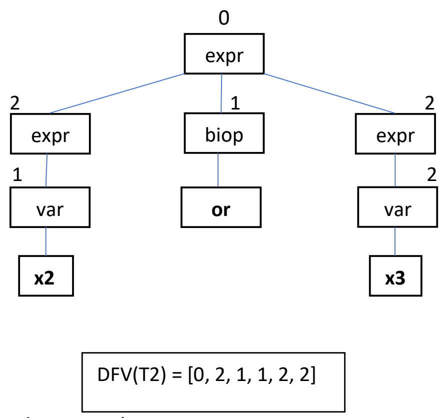


Fig. 4 Derivation tree for the expression `x2 or x3`.

inverse operation of generating (or deriving) a valid sentence of the grammar. This observation leads to an algorithmic recipe to *invert the GE genotype-to-phenotype mapping*, i.e., a mechanical way to compute the GE representation of any grammatically valid phenotypic expression: (i) using a standard parsing algorithm, parse the given phenotypic expression (sentence) w.r.t. the grammar; the obtained parse tree is the derivation tree of the phenotypic expression; (ii) use the numbering of the production rules in the grammar to annotate each non-terminal node of the derivation tree with the choice of production rule consistent with its child nodes (similar to GE Sensible Initialisation "unmodding" [17]); (iii) visit the derivation tree using depth-first traversal and collect the sequence of choices on the nodes. The resulting sequence is a genotype (one among many) of the given expression. For example, looking again at Figure 3 but from the bottom to the top this time, given the phenotypic expression `x1 and x2` and the grammar in Figure 2, a standard parsing algorithm can be used to obtain its (unannotated) parse tree, which is the same as the

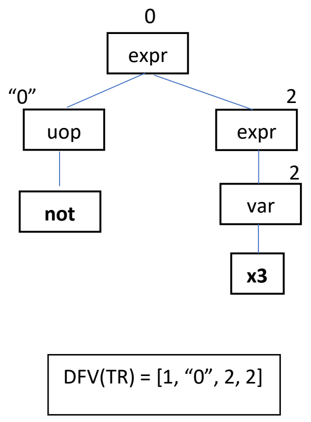
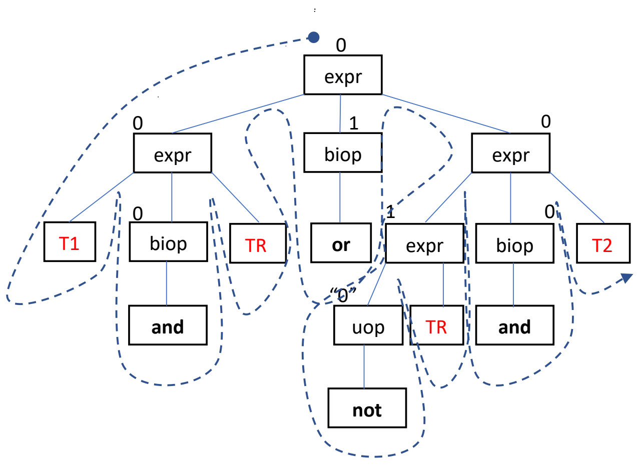


Fig. 5 Derivation tree for the expression `not x3`. The quotation marks (“0”) indicate that a codon is not strictly required, since only one production for the `uop` non-terminal exists; in some GE systems the codon is consumed regardless, and we follow this practice.

derivation tree. This can then be annotated by looking at the numbering of the grammatical productions in Figure 2, obtaining the same annotations. Then the genotype can be obtained by traversing depth-first the annotated tree obtaining the sequence $[0,2,0,0,2,1]$, which is the same as the original genotype.

Observation 2: The use of depth-first expansion of the parse tree makes the genotype-to-phenotype mapping *modular* in the following sense. As noted in the previous point, we can obtain the genotype associated with a parse tree by traversing depth-first the annotated tree (T) and collecting the numbers in sequential order obtaining the sequence S, i.e., $S = DFV(T)$. If we ‘hide’ any subtree of the derivation tree by replacing the subtree with a node X encapsulating the subtree and compute the genotype by depth-first traversal, we obtain that $DFV(T) = S1, DFV(X), S2$, which means that the depth-first visit of T is a sequence of the form: uninterrupted sequence $S1$, followed by the (unknown) uninterrupted sequence obtained by depth-first visit of the hidden tree X, followed by a second uninterrupted sequence $S2$. This holds for depth-first traversal because of its prioritisation of visit of the nodes in a tree, which has the property that when the traversal enters a subtree, it will then visit all its nodes before leaving it, and then it will not return to it anymore. This property does not hold for other tree traversal strategies. For example, it does not hold for breadth-first traversal of the tree. This is because breadth-first traversal could enter and leave any given subtree several times (more precisely, a number of times equal to the depth of the subtree) with the effect of interleaving the nodes of the subtree with the nodes of the rest of the tree in the output sequence. The modularity of the genotype-phenotype mapping is illustrated in Figure 6. The nodes with red labels are nodes encapsulating subtrees. The dash-line is the order of visit of the nodes of the depth-first traversal strategy. The genotype sequence contains the number associated to the non-terminal nodes, and when a hidden subtree is encountered (a red node), its genotype sequence is included as a self-contained subsequence. A similar



DFV(M) = [0, 0, DFV(T1), 0, DPV(TR), 1, 0, 1, "0", DPV(TR), 0, DPV(T2)]

Fig. 6 Derivation tree of crossover mask. The dash-line is the order of visit of the nodes of the depth-first traversal strategy.

concept of modularity in the genotype-phenotype mapping is implicit in the work of Hemberg [18] (p. 176) on the classification of operator behaviours in GE.

Observation 3: A geometric semantic operator is a function (i.e., recombination scheme) that when applied to input functions (parents) returns an output function (offspring). We observe that when viewed as a 'sentence' generated by a grammar, a geometric semantic operator is a syntactical expression representing the recombination scheme with 'holes' in which to plug the syntactical expressions representing the input functions. For example, geometric semantic crossover for Boolean functions, $T3 = (T1 \wedge TR) \vee (\overline{TR} \wedge T2)$, can be seen syntactically as a sentence of the grammar for Boolean expressions in Figure 2 where the unspecified input functions $T1$, $T2$ and TR (i.e., formal parameters of the recombination scheme) can be seen syntactically as 'holes' or 'hidden sub-sentences'. The corresponding syntax of the output function $T3$ can then be obtained by plugging in the syntactic expressions of $T1$, $T2$ and TR in the 'holes' of the syntactic representation of the recombination scheme.

From these observations it follows that we can obtain the GE genotypic representation corresponding to the recombination scheme, by applying the procedure outlined in observation 1 to invert the genotype-to-phenotype mapping to the syntactic representation of the recombination scheme (interpreted as in observation 3) i.e., parsing it, annotating the parse tree, and visiting the annotations depth-first. The 'holes' in the sentence correspond to 'hidden subtrees' in the parse tree of the sentence, which as argued in observation 2 correspond to self-contained subsequences in the genotype sequence. Figure 6 shows the parse tree of the syntactic representation of the recombination scheme $T3 = (T1 \wedge TR) \vee (\overline{TR} \wedge T2)$ and the corre-

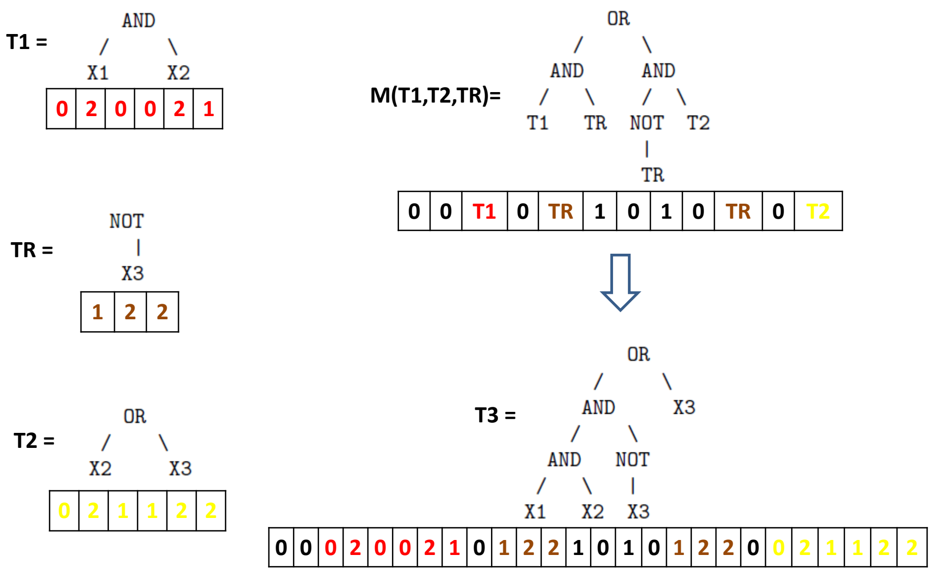


Fig. 7 Example of Geometric Semantic Search Operators on Grammatical Evolution Genotype.

sponding GE genotype. Given the GE genotypes of the functions $T1$, $T2$ and TR in Figures 3, 4 and 5 respectively, the GE genotype of the function $T3$ is then obtained by simply placing them in their places in the GE genotype of the recombination scheme (see Figure 7). This by construction is equivalent to the functional composition of the recombination scheme to generic functions $T1$, $T2$ and TR, hence it is the geometric semantic crossover for Boolean functions expressed using the GE representation.

4 Derivation of Geometric Semantic Operators for GE

The theory developed in the previous section is applied here to derive a complete set of geometric semantic operators for the GE genotype for Boolean, Arithmetic and Classifier domains. In particular, we aim at deriving crossover, mutation and initialisation operators acting *solely* on GE genotypes and being *guaranteed* by construction to be equivalent to geometric semantic operators acting on the corresponding expressed phenotypes. This allows an evolutionary process on GE genotypes exactly equivalent to an evolutionary process on the corresponding phenotypes. Note that the design of the search operators is inextricably dependent on the specific grammar used for each domain. The grammar however is used only in the design phase of

```
(A)  <expr> ::= (<expr> <biop> <expr>) (0)
              | <uop> <expr>          (1)
              | <var>                 (2)

(B)  <biop> ::= and                   (0)
              | or                    (1)

(C)  <uop>  ::= not                   (0)

(D)  <var>  ::= x1                    (0)
              | x2                    (1)
              | x3                    (2)
```

Fig. 8 Grammar for Boolean expressions.

the search operators. We do not allow it to be used during the search to e.g., repair the offspring generated by the operators. All the operators *by design* must be guaranteed to produce genotypes corresponding to grammatically valid phenotypic expressions. Furthermore, the offspring genotypes will be guaranteed to be perfectly formed without requiring genome "wrapping" or ignoring surplus genes.

4.1 Operators for Boolean Domains

In the following, we first introduce the grammar we use for Boolean expressions. We then derive crossover, mutation and initialisation operators on GE genotypes for Boolean expressions based on this grammar.

GRAMMAR: the grammar for Boolean expressions considered is in Figure 8. For simplicity of illustration, this grammar has only three variables (`x1`, `x2` and `x3`). This grammar can express any Boolean function of three variables. However, the grammar and the corresponding geometric semantic search operators on GE genotypes can be generalised to any number of variables and to expanded function sets.

CROSSOVER: The geometric semantic crossover for Boolean expressions is

$$T3 = (T1 \wedge TR) \vee (\overline{TR} \wedge T2)$$

where $T1$ and $T2$ are the parent Boolean expressions, TR is a random Boolean expression, and $T3$ is the offspring Boolean expression.

The geometric semantic crossover for GE is an operation on the genotype of parents that generates the genotype of the offspring such that the developmental process via the grammar produces the offspring whose expression is given above.

The corresponding geometric semantic crossover for this grammar is

```
0:        (<expr> <biop> <expr>)
0:        (<expr> <biop> <expr>)
g(T1):    T1
0:        and
g(TR):    TR
1:        or
0:        (<expr> <biop> <expr>)
1:        <uop> <expr>
0:        not
g(TR):    TR
0:        and
g(T2):    T2
```

Fig. 9 Derivation of phenotype for geometric semantic crossover for Boolean expressions.

$$g(T3) = [0,0,g(T1),0,g(TR),1,0,1,0,g(TR),0,g(T2)]$$

where $g(.)$ returns the genotype of its argument. The genotype $g(T3)$ of the offspring $T3$ is the sequence obtained by inserting the sequences $g(T1)$, $g(TR)$, and $g(T2)$ in the specified positions. Note that the genotypes of the parents ($g(T1)$ and $g(T2)$) are readily available from the previous stage of the evolutionary process. The genotype of the random expression ($g(TR)$) is generated using the initialisation procedure described below.

Figure 9 shows that expanding the expression $g(T3)$ using the grammar while considering $T1$, TR, and $T2$ as parameter expressions we obtain the geometric semantic crossover scheme on phenotypes.

MUTATION: The geometric semantic mutation for Boolean expressions returns the offspring Boolean expression $TM = T \vee M$ with probability 0.5 and $TM = T \wedge \overline{M}$ with probability 0.5 where T is the Boolean expression undergoing mutation, M is a random minterm of all input variables, and TM is the mutated Boolean expression.

The corresponding geometric semantic mutation for this grammar is $g(TM) = [0,g(T),1,g(M)]$ with probability 0.5 and $g(TM) = [0,g(T),0,1,g(M)]$ with probability 0.5. The genotype of the parent ($g(T)$) is readily available from the previous stage of the evolutionary process. The genotype of the random minterm ($g(M)$) is generated using the procedure in Figure 10, which illustrates it for three variables.

Figure 11 shows that expanding the expression $g(TM)$ using the grammar while considering T and M as parameter expressions we obtain the geometric semantic mutation scheme on phenotypes.

INITIALISATION: We aim at creating a random genotype that corresponds to a valid grammatical expression, i.e. a valid phenotype, without using wrapping or leaving unused codons or using modulus of the gene values. This would be easy to do by traversing the grammar to generate the genotypes. We however do not allow

```
def generate_mintermgeno(3 variables):
    result = [0] # <expr> <biop> <expr>

    result += random.choice([[], [1]]) # do-nothing, or negate
    result += [2, 0] # <var>, x1
    result += [0, 0] # and, <expr> <biop> <expr>

    result += random.choice([[], [1]]) # do-nothing, or negate
    result += [2, 1] # <var>, x2
    result += [0] # and

    result += random.choice([[], [1]]) # do-nothing, or negate
    result += [2, 2] # <var>, x3
    return result
```

Fig. 10 Procedure to build the genotype of a random minterm of three variables.

```
# T or M
0:        (<expr> <biop> <expr>)
g(T):     T
1:        or
g(M):     M

# T and not M
0:        (<expr> <biop> <expr>)
g(T):     T
0:        and
1:        (<uop> <expr>)
'0':      not
g(M):     M
```

Fig. 11 Derivation of phenotype for geometric semantic mutation for Boolean expressions.

explicit use of the grammar *at runtime* (apart from during fitness evaluation of genotypes, for which it is unavoidable), as we want the complete evolutionary process to happen on the genotypes only, i.e., all search operators, including initialisation, must not 'peep' through the genotype-phenotype mapping at runtime. We want the search operators, including initialisation, to work solely at genotype level, and induce via the genotype-phenotype map their intended effect at phenotype level, *entirely by design*. The design of these search operators naturally is inextricably grounded in the used grammar.

For Boolean expressions, the initialisation procedure used in GSGP is in Algorithm 1. We can design an initialisation operator acting entirely on genotypes inducing at the phenotype level the same behaviour by simply mapping each phenotypic sub-component to the corresponding genotypic sub-sequence, as illustrated in Algorithm 2.

Algorithm 1: Initialisation: Generate a random Boolean phenotype

```
1 Function RandomBoolean (depth)
2     if $depth = 1$ or $probability < 2^{-depth}$ then
3         return random.choice(x1, x2, ...)
4     else
5         with probability $1/3$: return (not RandomBoolean(depth-1))
6         with probability $1/3$: return ( RandomBoolean(depth-1) and
          RandomBoolean(depth-1))
7         with probability $1/3$: return ( RandomBoolean(depth-1) or
          RandomBoolean(depth-1))
```

Algorithm 2: Initialisation: Generate a valid random Boolean genotype

```
1  Function RandomBoolean (depth)
2      if $depth = 1$ or $probability < 2^{-depth}$ then
3          return [2] + [RndInt(numvar)]
4          (phenotype = x1 / x2 / etc)
5      else
6          with probability $1/3$: return [1] + RandomBoolean(depth-1)
7          (phenotype = not < expr >)
8          with probability $1/3$: return [0] + RandomBoolean(depth-1) + [0] +
           RandomBoolean(depth-1)
9          (phenotype = < expr > and < expr >)
10         with probability $1/3$: return [0] + RandomBoolean(depth-1) + [1] +
           RandomBoolean(depth-1)
11         (phenotype = < expr > or < expr >)
```

4.2 *Operators for Arithmetic Domains*

In the following, we first introduce the grammar we use for arithmetic expressions. We then derive crossover, mutation and initialisation operators on GE genotypes for arithmetic expressions based on this grammar.

GRAMMAR: the grammar for arithmetic expressions considered is in Figure 12. This grammar can express any polynomial of three variables. However, the grammar

```
(A) <expr>  ::= (<expr> <biop> <expr>) (0)
              | <var>                  (1)
              | <const>                (2)

(B) <biop>  ::= +                      (0)
              | -                      (1)
              | *                      (2)

(C) <var>   ::= x1                     (0)
              | x2                     (1)
              | X3                     (2)

(D) <const> ::= 0.0                    (0)
              | 0.1                    (1)
              ...                      ...
              | 1.0                    (10)
```

Fig. 12 Grammar for Arithmetic expressions.

and the corresponding geometric semantic search operators on GE genotypes can be readily generalised to any number of variables and other function sets.

CROSSOVER: The geometric semantic crossover for arithmetic expressions is

$$T3 = (T1 \cdot TR) + ((1 - TR) \cdot T2)$$

where $T1$ and $T2$ are the parent arithmetic expressions, TR is a random real constant in $[0,1]$, and $T3$ is the offspring arithmetic expression.

The corresponding geometric semantic crossover for this grammar is

$$g(T3) = [0,0,2,g(TR),2,g(T1),0,0,0,2,10,1,2,g(TR),2,g(T2)]$$

where $g(.)$ returns the genotype of its argument. The offspring $T3$ has the genotype formed by substituting the genotypes of $T1$, $T2$ and TR ($g(T1)$, $g(T2)$ and $g(TR)$) in the above pattern. For simplicity of illustration, we assume TR takes only values 0.0, 0.1, ..., 1.0, so that $g(TR)$ is a random integer between 0 and 10, producing floating-point values through use of the `<const>` non-terminal in Fig. 12.

MUTATION: The geometric semantic mutation for arithmetic expressions returns the offspring $TM = T + ms \cdot (TR1 - TR2)$ where T is the Boolean expression undergoing mutation, $TR1$ and $TR2$ are random arithmetic expressions, and ms is the mutation step, which is a constant real value.

The corresponding geometric semantic mutation for this grammar is $g(TM) = [0,g(T),0,0,g(ms),2,0,g(TR1),1,g(TR2)]$. The genotypes of the random arithmetic expressions ($g(TR1)$ and $g(TR2)$) are generated using the initialisation procedure for arithmetic expressions presented below. The genotype of the parameter *ms* can be obtained by factoring the parameter appropriately as a valid sentence of the grammar, and then deriving its genotype by parsing this sentence. For example, $ms = 0.001$ can be factored into $ms = 0.1 * 0.1 * 0.1$ which is a valid expression in the given grammar, and so its genotype can be derived, in this case obtaining $g(ms) = [0,0,2,1,2,2,1,2,2,1]$.

INITIALISATION: We can design an initialisation operator acting entirely on genotypes inducing at the phenotype level the same behaviour as the initialisation procedure used in GSGP. It works by mapping each phenotypic sub-component to the corresponding genotypic sub-sequence, as illustrated in Algorithm 3.

Algorithm 3: Initialisation: Generate a valid random Arithmetic genotype

```
1 Function RandomArithmetic (depth)
2     if $depth = 1$ or probability $< 2^{-depth}$ then
3         with probability $1/2$: return [1] + [RndInt(numvar)]
4         (phenotype = < var >, x1 / x2 / etc)
5         with probability $1/2$: return [2] + [RndInt(numconst)]
6         (phenotype = < const >, 0.0 / 0.1 / etc)
7     else
8         return [0] + RandomArithmetic(depth-1) + [RndInt(numop)]
          + RandomArithmetic(depth-1)
9         (phenotype = < expr >, (+ / - / * ), < expr >)
```

4.3 Operators for Classifier Domains

In the following, we first introduce the grammar we use for classifiers i.e., nested `if`-expressions. We then derive crossover, mutation and initialisation operators on GE genotypes for classifiers based on this grammar.

GRAMMAR: the grammar for classifiers considered is in Figure 13. For simplicity of illustration, this grammar has only three variables (`x1`, `x2` and `x3`), three input symbols (`is1`, `is2` and `is3`), and two output symbols (`os1` and `os2`). This grammar can express any classifier of three variables with three input classes and two output classes. However, the grammar and the corresponding geometric semantic search operators can be generalised to any number of variables, input symbols, and output symbols.

CROSSOVER: The geometric semantic crossover for classifiers is

$$T3 = T1 \, \texttt{IF} \, CONDR \, \texttt{ELSE} \, T2$$

```
(A) <cf> ::= (<cf> if <cond> else <cf>) (0)
           | <os>                       (1)

(B) <cond> ::= <var> == <is>            (0)
           | <cond> and <var> == <is>   (1)

(C) <is> ::= is1                        (0)
         | is2                          (1)
         | is3                          (2)

(D) <os> ::= os1                        (0)
         | os2                          (1)

(E) <var> ::= x1                        (0)
          | x2                          (1)
          | x3                          (2)
```

Fig. 13 Grammar for Classifiers.

where $T1$ and $T2$ are the parent classifiers, *CONDR* is a random condition depending on one or more input variables, and $T3$ is the offspring classifier [1].

The corresponding geometric semantic crossover for this grammar is

$$g(T3) = [0, g(T1), 0, g(Rvar), g(Ris), g(T2)]$$

where $g(.)$ returns the genotype of its argument. For simplicity of illustration, the random condition *CONDR* is of the form $Rvar == Ris$, where *Rvar* is a random variable and *Ris* is a random input symbol. The offspring $T3$ has the genotype formed by substituting the genotypes of $T1$, $T2$, *Rvar* and *Ris*, ($g(T1)$, $g(T2)$, $g(Rvar)$ and $g(Ris)$) in the above pattern. The genotype of the random variable and the random input symbol ($g(Rvar)$ and $g(Ris)$) are both integers randomly chosen from {0, 1, 2}, since in the grammar there are three input variables and three input symbols.

MUTATION: The geometric semantic mutation for classifiers returns the offspring classifier `TM = IF CONDR THEN OUTR ELSE T` where `T` is the parent classifier undergoing mutation, `CONDR` is a condition which is true only for a single random setting of all input parameters, and `OUTR` is a random output symbol. The offspring can be expressed as nested `IF-THEN-ELSE` statements with simple conditions of a single input parameter each.

The corresponding geometric semantic mutation for this grammar is $g(TM) = [0, 1, g(OUTR), g(CONDR), g(T)]$. The genotype of the parent ($g(T)$) is readily available from the previous stage of the evolutionary process. The genotype of the

[1] Implementation note: The unusual IF-ELSE syntax here means that (in Python) the code is a single expression – which can be evaluated using Python's `eval()` – rather than a statement, which cannot.

```
def generate_conjunction(3 variables):
     result = []

     result += [1, 1, 0]
     # <cond> ->
     # <cond> and <var> == <is> ->
     # <cond> and <var> == <is> and <var> == <is> ->
     # <var> == <is> and <var> == <is> and <var> == is

     result += [0, random.rangrange(n)]
     # <var> == <is> -> x1 == i3 (eg)

     result += [1, random.rangrange(n)]
     # <var> == <is> -> x2 == i2 (eg)

     result += [2, random.rangrange(n)]
     # <var> == <is> -> x3 == i2 (eg)

     return result
```

Fig. 14 Procedure to build the genotype of a random condition of three variables.

random output symbol ($g(OUTR)$) is an integer randomly chosen from $\{0, 1\}$, since in the grammar there are two output symbols. The genotype of the random condition ($g(CONDR)$) is generated using the procedure in Figure 14, which illustrates it for three variables where each can take on n possible values.

INITIALISATION: We design an initialisation operator acting entirely on genotypes inducing at the phenotype level the same behaviour as the initialisation procedure used in GSGP by simply mapping each phenotypic sub-component to the corresponding genotypic sub-sequence, as illustrated in Algorithm 4.

Algorithm 4: Initialisation: Generate a valid random classifier genotype

```
1 Function RandomClassifier (depth)
2     if $depth = 1$ or probability $< 2^{-depth}$ then
3         return [1] + [RndInt(numos)]
4         (phenotype = $<os>$, o1 / o2 / o3 etc (output symbols))
5     else
6         return [0] + RandomClassifier(depth-1) + [0,
          RndInt(numvar), RndInt(numis)] +
          RandomClassifier(depth-1)
7         (phenotype = ($<cf>$ if $<cond>$ else $<cf>$), $<expr>$, $<var>$ == $<is>$,
          (x1 / x2 / x3 etc), (i1 / i2 / i3 etc), $<expr>$)
```

5 An Efficient Implementation of GSGE

A drawback of GSGP with crossover is the exponential growth of individuals due to the fact that the offspring tree contains both parent trees, hence individuals double their size at each generation. This problem applies to GSGE also. One solution, proposed in [1], is to keep program size manageable using automated simplification during the run.

Castelli et al. [19] proposed an implementation of GSGP that avoids exponential growth by referring via pointers to a trace of the ancestry of individuals, rather than storing them directly. We propose a new implementation of GSGP and GSGE also based on tracing the ancestry of individuals, that however does not explicitly build and maintain a new data structure, but uses higher-order functions and memoization to achieve the same effect, leaving the burden of book-keeping to the compiler. The resulting implementation is fast, elegant and concise. A Python implementation of GSGP with this feature (under 100 lines without comments) is on GitHub at https://github.com/amoraglio/GSGP, while the GSGE code used in this paper is available at https://github.com/jmmcd/GSGE.

SOLUTION REPRESENTATION: We represent solutions directly using functions of the programming language used to program the GSGP system. E.g., in a GSGP to evolve Boolean expressions written in Python, the representation of a Boolean expression is a Python (anonymous) function computing that Boolean expression, and not a data structure (e.g., a tree) representing the Boolean expression.

SEARCH OPERATORS: Geometric semantic crossover and mutation can be interpreted as higher-order functions. We implement them directly as such: they do not manipulate data structures representing solutions, but take directly as inputs the (anonymous) parent functions and return (anonymous) offspring functions. The returned offspring function *calls* the parent functions in its definition. In particular, the parent function definitions *are not substituted* in the offspring definition, hence there is no growth of the offspring function. The function calls to the parents in the offspring implicitly build the data structure that relates offspring to parents all the way up the ancestry without the need to use pointers, manage memory and maintain an archive of past solutions.

FITNESS EVALUATION: Even if individuals do not grow, evaluating them takes exponential time, as querying a function for some input requires calling both its parents on that input, which in turn need to call their parents on it and so forth, doubling the number of calls at each generation. The complexity of queries on training data can be reduced from exponential to constant time by memoization (i.e., caching the output values of a function of previously encountered inputs rather than recomputing them) of all individuals generated in the course of evolution. This works because each individual caches its outputs on the training examples the first time its fitness is computed, and later re-uses them when its descendents call it. This reduces the number of calls needed to compute the fitness of an individual from exponential to the number of parents, i.e. two, constant. Memoization is easily implemented as a higher-order wrapping function (it is a standard library function in Python 3.2+).

DISPLAY OF BEST INDIVIDUAL: As solutions are represented directly as compiled Python functions, displaying them (in particular the best-of-run individual) would require decompilation, which is not very practical. The technique we have used to display functions that avoids both decompilation and direct representations of functions during evolution consists of adding an extra implicit call structure in individuals, where the extra structure implicitly keeps track of how to reconstruct the final genotype of the individual (its source code) mirroring the first call structure (its semantics) interpreting subroutine calls as function body substitutions (i.e. asking the parents to return their source code to embed in the offspring source code). Then individuals can be asked to display themselves by calling their associated 'source code' function. This can be implemented with minor additions to the code. Naturally displaying the best individual after evolution takes exponential time as its genotype is exponentially long. However, querying the final solution on unseen values (i.e. making predictions) takes only time linear in the number of distinct ancestors thanks to the memoization of individuals. The number of distinct ancestors grows linearly with the number of generations (not exponentially, in the long term, because the population size is fixed).

6 Computational Experiments

We next present extensive experimental results. Our goal is to compare GE and GSGE representations. For comparison with previous work, we will include the GSGP representation also. As the fitness landscape is unimodal, we expect a semantic stochastic hill-climber to find the optimal solution efficiently. Therefore, we will test both hill-climbing and evolutionary search algorithms. Finally, our choice of test problems mimics that of [1]. Thus, we will compare:

- GE, GSGE, and GSGP representations;
- stochastic hill-climbing and evolutionary search algorithms;
- symbolic regression, Boolean, and classifier problems.

Based on theory, we expect that GSGE will obtain the same very good performance as GSGP in these experiments, as the two systems perform an equivalent search: the search done by GSGE on genotypes projected through the genotype-phenotype mapping coincides with the search done by GSGP on phenotypes.

The training data in the symbolic regression problems is synthesized from polynomials with coefficients uniformly sampled in $[-1,1]$. The degree of the polynomials is varied from 3 to 10, in order to scale problem difficulty. The test data is resampled independently in the same way.

The Boolean problems are True, n-Parity, Comparator, Multiplexer, and Random. True is the Boolean function which returns True for any input. Random is a Boolean function whose truth table is randomly generated. Again, each problem is tested in several sizes in order to scale problem difficulty. The training data consists of all possible cases, and the test data is the same.

The classifier problems are synthetic. Each problem is characterised by its number of input variables n_v, number of possible values of these variables n_i, and number of possible output values n_o. Each input variable may take integer values in the range $[0, n_i - 1]$. The output is an integer in the range $[0, n_o - 1]$. It is a simple synthetic function of the input variables, $(x_0 + x_1) \bmod n_o$. For example, with $n_v = n_i = n_o = 2$, the classifier is equivalent to Boolean addition.

To facilitate easy comparison, we will report the *percentage of hits* and the standard deviation in this figure, for each problem, each representation, and each search algorithm. A hit is a test case correctly solved by the best individual of the run. On Boolean and classifier problems, a hit means the correct answer. On symbolic regression problems, a hit means an output value within 0.01 of the correct value.

Table 1 shows results. The GSGP results effectively replicate those reported by Moraglio et al. [1], with very strong performance using both search algorithms, often slightly better using hill-climbing versus evolutionary search. As expected, the GSGE results are effectively identical to the GSGP results, confirming that GSGE operators "see" the cone landscape characteristic of GSGP.

In contrast, GE itself does poorly, especially on the symbolic regression and n-Parity problems. With GE, evolutionary search tends to work better than hill-climbing. Note that the comparison to GE may be called unfair for two reasons. Our implementations of both GE and GSGE do not use a feature which has come to be common in GE implementations, *sensible initialisation* [17]; and in our implementations of both GE and GSGE, non-coding tails have been cut. Recent work has suggested that non-coding tails can improve performance in GE [20].

7 Discussion

This work had a two-fold motivation. The first was to extend the GSGP framework to a new representation. The second was to show how to design provably good search operators for GE. In the following we discuss these two perspectives in the light of the work presented in this chapter.

Why apply GSGP to GE? On one hand, GSGE has provably good performance. On the other hand, the search on GE genotypes is exactly equivalent to the search done by GSGP on phenotypes. If they are equivalent, why bother using GSGE instead of GSGP? Expressing geometric semantic search operators in the various GP representations (GE, Cartesian GP, PushGP, etc.) and more generally for evolutionary approaches to evolving functions (e.g., evolving neural networks, finite state machines, etc.) is a good thing for three reasons: (i) it allows for unification and direct comparison of very different representations; (ii) it unveils the specific prop-

Table 1 Results with GE, GSGE, and GSGP representations on various problems, at various sizes, using hill-climbing (HC) and evolutionary (Evo) search. For classifier problems, problem size is given as n_v, n_i, n_o.

		GE/HC		GE/Evo		GSGE/HC		GSGE/Evo		GSGP/HC		GSGP/Evo	
problem	size	avg	sd	avg	sd	avg	sd	avg	sd	avg	sd	avg	sd
polynomial	3	4.2	8.4	21.0	25.5	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0
	4	4.5	7.0	10.8	18.5	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0
	5	3.0	5.9	10.0	12.8	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0
	6	3.2	5.8	10.0	8.0	99.8	0.9	99.3	2.8	100.0	0.0	99.5	2.0
	7	3.2	5.4	5.5	4.9	100.0	0.0	91.5	16.0	100.0	0.0	93.3	12.6
	8	2.0	3.8	10.2	11.6	99.5	2.0	84.5	14.9	99.5	2.0	86.8	13.8
	9	2.2	4.9	7.5	7.8	91.2	14.0	69.3	25.8	94.7	8.0	70.3	25.5
	10	2.3	5.4	5.2	6.4	87.2	16.3	64.5	22.9	88.8	14.8	67.5	22.9
boolean true	5	89.0	17.4	98.3	5.1	99.1	1.6	99.5	1.4	99.7	0.9	99.0	1.5
	6	88.5	19.9	100.0	0.0	99.7	0.6	99.2	1.0	99.8	0.5	99.1	1.1
	7	87.5	21.2	100.0	0.0	99.9	0.2	99.8	0.4	99.9	0.3	99.9	0.2
	8	84.2	22.8	100.0	0.0	100.0	0.1	100.0	0.1	100.0	0.1	99.9	0.2
nparity	5	50.2	0.8	50.3	0.9	99.4	1.5	94.6	3.2	99.7	0.9	95.1	3.3
	6	50.0	0.0	50.0	0.0	99.9	0.4	96.9	1.9	99.9	0.4	97.5	1.7
	7	50.0	0.0	50.0	0.0	100.0	0.1	98.9	0.7	100.0	0.0	99.0	1.0
	8	50.0	0.0	50.0	0.0	100.0	0.1	98.6	0.9	100.0	0.1	98.6	0.6
	9	50.0	0.0	50.0	0.0	100.0	0.0	98.8	0.5	100.0	0.0	98.9	0.4
	10	50.0	0.0	50.0	0.0	100.0	0.0	98.8	0.3	100.0	0.0	98.7	0.3
comparator	6	75.6	1.9	73.8	4.7	99.9	0.4	98.8	1.2	99.9	0.4	98.4	1.8
	8	75.5	1.2	78.9	3.9	100.0	0.1	99.6	0.4	100.0	0.1	99.6	0.4
	10	75.3	1.0	79.3	3.2	100.0	0.0	99.9	0.1	100.0	0.0	99.9	0.1
multiplexer	6	64.5	2.5	64.3	2.6	99.9	0.4	98.5	1.4	99.8	0.5	98.2	1.8
	11	57.8	1.8	63.2	2.5	100.0	0.0	99.8	0.1	100.0	0.0	99.8	0.1
random boolean	5	66.5	4.5	64.2	4.6	99.7	0.9	96.8	3.4	99.6	1.1	96.6	2.9
	6	61.5	4.3	61.6	4.2	99.9	0.4	97.9	1.6	99.8	0.5	97.9	2.0
	7	58.0	3.1	60.5	2.6	99.9	0.4	99.2	0.8	99.9	0.3	99.2	0.7
	8	56.7	2.3	58.3	2.0	100.0	0.1	99.2	0.5	100.0	0.1	99.1	0.5
	9	55.1	1.3	56.7	1.3	100.0	0.0	99.3	0.3	100.0	0.0	99.3	0.4
	10	53.6	1.2	55.0	0.9	100.0	0.0	99.4	0.2	100.0	0.0	99.3	0.2
	11	52.6	0.8	53.8	0.8	100.0	0.0	99.1	0.2	100.0	0.0	99.2	0.2
classifier	3,3,2	55.6	0.0	55.4	0.7	55.3	0.9	55.6	0.0	55.6	0.0	55.6	0.0
	3,3,4	34.2	2.8	34.1	4.7	77.5	0.9	74.9	3.9	77.4	1.1	74.3	3.6
	3,3,8	34.6	3.4	35.8	8.1	99.6	1.1	84.9	5.0	99.5	1.3	85.1	4.6
	3,4,2	50.0	0.0	49.7	1.0	49.9	0.4	49.8	0.5	50.0	0.0	50.0	0.0
	3,4,4	29.1	3.2	27.3	3.0	74.9	0.3	73.0	1.6	74.8	0.5	71.9	1.6
	3,4,8	25.7	1.9	27.9	4.5	99.9	0.4	86.0	2.8	99.9	0.3	84.1	3.7
	4,3,2	55.6	0.0	53.7	10.0	55.6	0.0	55.6	0.0	55.6	0.0	55.6	0.0
	4,3,4	35.8	4.2	38.2	9.9	77.7	0.2	76.7	1.0	77.8	0.0	75.9	1.1
	4,3,8	34.8	3.8	41.7	6.8	100.0	0.2	88.0	2.6	100.0	0.0	87.9	2.6
	4,4,2	50.0	0.0	50.0	0.0	50.0	0.1	50.0	0.0	50.0	0.0	50.0	0.0
	4,4,4	28.7	3.0	35.2	6.4	75.0	0.0	74.3	0.5	75.0	0.1	74.5	0.5
	4,4,8	25.3	1.2	37.3	6.3	100.0	0.0	94.7	0.9	100.0	0.1	94.9	1.1

erties of a representation that are ultimately linked to good performance (unimodal landscapes); and (iii) it allows us to understand GSGP ideas in more detail.

The ideas of GSGP have transferred successfully to the GE representation. GE search operators that see a unimodal landscape can be built for any problem, and they can be built mechanically for any new grammar. By transferring GSGP ideas to GE, we have learned that the GE map is *modular*, with compositional semantics, and that this is a requirement for any new representation for GSGP. We have also seen that GSGE solutions grow exponentially, but that their growth can be reduced to linear.

Some of the specific benefits of GE are:

Constrained The grammar in GE can be used to enforce regularities and other constraints to solutions.
Linearity The linear genotype allows for simple search operators.
Developmental In GE a small genotype can express a large phenotype (via wrapping). Even without wrapping, developmental effects can come in to play, such as a greater importance of the earliest genes in the genotype.
Neutrality Unused codons can function as a "memory" of previous solutions.

How do these beneficial aspects of GE transfer to GSGE?

Constrained This property is linked to using grammars to enforce constraints, and has been used at a phenotypic level with GSGE, i.e., grammars can be used directly in GSGP, see e.g. [16].
Linearity The GSGE operators are not as simple as those in GE.
Developmental In GSGE, the developmental mapping is of less importance. The size of the phenotype is directly proportional to the size of the genotype.
Neutrality Unused codons do not occur in GSGE. However, because GSGE individuals functionally incorporate all ancestors, there is a type of "memory".

From this analysis, it seems that GSGE uses the GE language to express a fundamentally different search than that done by GE itself. When two different perspectives are presented in a common language, it is often the case that their features can be fruitfully combined to produce unexpected novel ideas and results. This is where we are at the moment! In a broader sense, GSGE "completes" GE as it makes a link with semantics and the unimodal landscape. All of these seem to be ingredients necessary for evolving programs, the holy grail of GE. It would be interesting to investigate how different ways of including semantics in GE (e.g. attribute grammars) can be linked to GSGE.

8 Summary

In this chapter, we have recalled that GSGP sees a unimodal fitness landscapes for any problem. Geometric semantic search operators are purely functional operators that do not depend on the underlying representation. In principle, any representation could be used if sufficiently expressive to describe these operators algorithmically.

In practice, geometric semantic search operators are naturally expressed in functional languages as higher order functions. Even if in principle possible, it could be practically impossible to express these operators in a language or representation which does not naturally express functional relations and operations.

The GE encoding is rather complex, especially when using wrapping. It has been shown to have low locality [21]: small changes of the genotype may correspond to large changes on the phenotype, leading to highly disruptive operators (i.e., ripple effect) and highly discontinuous fitness landscapes. We have thus asked the question: can we express geometric semantic search operators using the GE encoding?

Expressing geometric semantic search operators on GE genotypes that act equivalently to geometric semantic operators on expressions (phenotypes) requires an understanding of how to invert the GE genotype-phenotype map, and project through this mapping search operators on the phenotype space to corresponding search operators on the genotype space. Given the complexity of the GE mapping, determining such operators rigorously may seem hopeless. Surprisingly, in this chapter this goal has been achieved.

The key property of the GE mapping that allows this is its modularity: a subexpression in the phenotype corresponds to an uninterrupted subsequence in the genotype. This allows functional composition (at the phenotypic level) to be expressed as plugging a subsequence into a sequence schema (at the genotypic level). Geometric semantic operators are then expressed at a genotypic level as specific sequence schema.

The genotypic definitions of geometric semantic search operators depend inextricably on the specific grammar used, as they are designed around the genotype-phenotype mapping. However, these operators can be derived mechanically by parsing their phenotypic expression using the grammar, and then linearizing the parse tree by depth-first traversal. We have put this methodology into practice, deriving geometric semantic crossover, mutation and initialisation for GE, equivalent to existing GSGP operators for Boolean, Arithmetic and Classifier domains.

The new GSGE operators produce exponentially large solutions, similar to GSGP. However, we have provided an elegant implementation of these operators based on interpreting the operators as higher-order functions and making use of memoization, which reduces the growth from exponential to linear (in the number of ancestors).

Finally, we have reflected that GSGE, even if phrased using the same representation, is fundamentally quite different from standard GE.

References

1. Moraglio, A., Krawiec, K., Johnson, C.: Geometric semantic genetic programming. In: Proc. PPSN XII, Springer (2012) 21–31
2. Moraglio, A., Mambrini, A., Manzoni, L.: Runtime analysis of mutation-based geometric semantic genetic programming on boolean functions. In: Proceedings of the Twelfth Workshop on Foundations of Genetic Algorithms XII. FOGA XII '13, New York, NY, USA, ACM (2013) 119–132
3. Moraglio, A., Mambrini, A.: Runtime analysis of mutation-based geometric semantic genetic programming for basis functions regression. In: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation. GECCO '13, New York, NY, USA, ACM (2013) 989–996
4. Mambrini, A., Manzoni, L., Moraglio, A.: Theory-laden design of mutation-based geometric semantic genetic programming for learning classification trees. In: 2013 IEEE Congress on Evolutionary Computation. (June 2013) 416–423
5. O'Neill, M., Ryan, C.: Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language. Genetic programming. Kluwer Academic Publishers (2003)
6. Rothlauf, F., Oetzel, M.: On the locality of grammatical evolution. In Collet, P., et al., eds.: EuroGP. Volume 3905 of LNCS., Budapest, Hungary, Springer (10 - 12 April 2006) 320–330
7. Beadle, L., Johnson, C.G.: Semantic analysis of program initialisation in genetic programming. Genetic Programming and Evolvable Machines **10**(3) (2009) 307–337
8. Jackson, D.: Phenotypic diversity in initial genetic programming populations. In: Proc. of EuroGP 2010. (2010) 98–109
9. Beadle, L., Johnson, C.G.: Semantically driven mutation in genetic programming. In: Proc. of IEEE CEC '09. (2009) 1336–1342
10. Beadle, L., Johnson, C.G.: Sematically driven crossover in genetic programming. In: Proc. of IEEE WCCI '08. (2008) 111–116
11. Uy, N.Q., Hoai, N.X., O'Neill, M., McKay, R., Galván-López, E.: Semantically-based crossover in genetic programming: Application to real-valued symbolic regression. Genetic Programming and Evolvable Machines **12**(2) (2011) 91–119
12. Krawiec, K., Lichocki, P.: Approximating geometric crossover in semantic space. In: Proc. of GECCO '09. (2009) 987–994
13. Krawiec, K., Wieloch, B.: Analysis of semantic modularity for genetic programming. Foundations of Computing and Decision Sciences **34**(4) (2009) 265–285
14. Moraglio, A., Poli, R.: Topological interpretation of crossover. In: Proc. of GECCO '04. (2004) 1377–1388
15. Moraglio, A.: Towards a Geometric Unification of Evolutionary Algorithms. PhD thesis, University of Essex (2007)
16. Moraglio, A., Krawiec, K.: Geometric semantic genetic programming for recursive boolean programs. In: Proceedings of the Genetic and Evolutionary Computation Conference, ACM (2017) 993–1000
17. Ryan, C., Azad, A.: Sensible initialisation in grammatical evolution. In Barry, A.M., ed.: GECCO Bird of a Feather Workshops, Chicago, IL, USA (2003) 142–145
18. Hemberg, E.A.P.: An exploration of grammars in grammatical evolution. PhD thesis, University College Dublin (2010)
19. Castelli, M., Silva, S., Vanneschi, L.: A C++ framework for geometric semantic genetic programming. Genetic Programming and Evolvable Machines **16**(1) (2015) 73–81
20. Nicolau, M., O'Neill, M., Brabazon, A.: Termination in grammatical evolution: Grammar design, wrapping, and tails. In: Evolutionary Computation (CEC), 2012 IEEE Congress on, IEEE (2012) 1–8
21. Rothlauf, F., Oetzel, M.: On the locality of grammatical evolution. In Collet, P., Tomassini, M., Ebner, M., Gustafson, S., Ekárt, A., eds.: EuroGP. Volume 3905 of LNCS., Springer (2006) 320–330