Understanding Grammatical Evolution: Grammar Design

Miguel Nicolau and Alexandros Agapitos

Abstract A frequently overlooked consideration when using Grammatical Evolution (GE) is grammar design. This is because there is an infinite number of grammars that can specify the same syntax. There are, however, certain aspects of grammar design that greatly affect the speed of convergence and quality of solutions generated with GE. In this chapter, general guidelines for grammar design are presented. These are domain-independent, and can be used when applying GE to any problem. An extensive analysis of their effect and results across a large set of experiments are reported.

1 Introduction

One of the attractive aspects of Grammatical Evolution (GE) is how it can be easily applied to a multitude of problem domains: just design a grammar specifying the syntax of potential solutions, and supply a fitness function to evaluate them.

Easily and *just* are large over-simplifications. While specifying the syntax of solutions with a context-free grammar is a relatively simple task, depending on the problem domain (a multitude of grammars exist in the literature for symbolic regression applications, for example), there is an infinite number of grammars that can specify the same syntax. But not all of them are adequate for use with GE.

In fact, the effectiveness of a grammar is deeply tied to GE's mapping process, and its effect on the search operators. In this study, this effect is analysed, and general guidelines are provided, with the aim of improving GE's search process.

There are many aspects to consider, when designing a grammar for use with GE. Some of these include which and how many non-terminal symbols to use,

M. Nicolau (✉)
College of Business, University College Dublin, Dublin, Ireland
ORCiD: orcid.org/0000-0002-1981-1300
e-mail: Miguel.Nicolau@ucd.ie

A. Agapitos
School of Computer Science, University College Dublin, Dublin, Ireland

C. Ryan et al. (eds.), *Handbook of Grammatical Evolution*,
https://doi.org/10.1007/978-3-319-78717-6_2

recursiveness, mapping probabilities, symbol biases, length of derivation sequences, prefix vs. infix vs. postfix notation, readability/understandability/maintenance of grammars, and many more.

These topics are analysed in this chapter, in terms of initial search space sampling (i.e. their combined effect with initialisation), search effectiveness (combined effect with the search operators), and quality of final solutions (fitness and size of final solutions). A large set of experiments are also executed, for empirical evidence. The results obtained confirm and highlight just how much grammar design can affect the performance of GE.

Based on these findings, a set of general grammar design guidelines are proposed for GE with linear genome representations. Although the resulting grammars can be substantially larger and more complex, most of these transformations can be automatically applied to well-designed base grammars.

2 Previous Work

Although a large volume of work exists in the literature on grammar design, particularly in linguistics and computer science, it almost exclusively relates to their use in parsing applications, i.e. syntax verification, compiler design, text mining, etc. In GE, on the contrary, grammars are used in a constructive manner, which, when combined with linear numerical sequences (genotypes) to determine derivation sequences, creates a mostly unique role for grammars.

There is surprisingly little work in the literature on the design of grammars for GE. This is probably due to the remarkable resilient nature of the evolutionary search process: given a correct and reasonably designed grammar, GE tends to produce a working solution. This is not always ideal, however, both in terms of the search effort required, and also the quality of the final solutions produced.

One of the earliest studies of the influence of grammar design on the performance of GE [22] looked specifically at reducing the number of non-terminal symbols in grammars, and proposed an automatic process of achieving this. This resulted in a small increase in performance across all problems attempted.

Hemberg et al. [11] studied the design of grammars using prefix, infix and postfix notation, and their relative performance on a series of symbolic regression problems. The most relevant conclusion is indeed that “the choice of grammar can produce performance advantage”.

One of the most comprehensive analysis of the influence of grammar design in the performance of GE is found in Hemberg’s doctoral thesis [10]. By allowing the grammar to evolve at the same time as the linear genome structures, knowledge is uncovered about the influence of different grammars on the performance of GE. Not surprisingly, it was again concluded that the choice of grammar can influence the performance of GE, for the problems examined.

Byrne et al. [1] analysed two types of mutation events in GE, structural or nodal in nature, and showed how these can be related to exploration and exploitation, respectively. These mutation events are directly dependent on the design of the grammars used.

Harper [9] highlighted the problem of having more production rules adding non-terminal symbols to the mapping sequence than removing them, and the negative impact of this on the performance of GE, if using linear genomes. Nicolau et al. [26] also analysed the effect of grammar design, focusing on the issue of mapping termination. Both studies highlighted the importance of good grammar design.

Grammar design, and its corresponding effect in representational bias, can also greatly influence the size of the resulting solutions. In Genetic Programming (GP), a possible outcome of this is bloat, i.e. a substantial growth in solution size, with negligible effect in performance increase [20]. Although bloat in GE is not quite as prevalent, studies have shown how grammar design directly influence the generation of very small [25] or very large [9] solutions.

More recently, work has been made on the design of grammars for sorting networks [6] and automatic program synthesis [7], although the results obtained only apply to grammar-based genetic programming systems using derivation trees.

All these studies have a common theme, which is how grammar design in GE can greatly affect its performance. The following section takes this into account, and presents a series of grammar design guidelines, or transformations for existing grammars, aimed at improving the performance of GE in different levels.

3 Grammar Design

A series of grammar transformations are presented in this section. To illustrate their application, Grammar 0 (G0) is used as a starting point. This is a typical grammar for symbolic regression applications with GE, slightly simplified (no unary operators) to illustrate the design techniques presented.

```
<s> ::= <e>
<e> ::= <e> <o> <e>
      | ( <e> <o> <e> )
      | <f> ( <e> , <e> )
      | <v>
<o> ::= + | - | *
<f> ::= pdiv
<v> ::= x | 1.0
```

Grammar 0 Simple arithmetical expressions grammar. Division uses a protected implementation, termed `pdiv` (more details in Sect. 5.3)

3.1 Balanced Grammars

In order to generate variable-length, unbounded phenotype solutions for any problem, GE makes use of two components:

– Variable-length genotype structures[1];
– Recursively-defined grammar rules.

Both conditions are required, in order to generate phenotype solutions of any size. If genotypes are unbounded but the grammar has no recursively defined symbols, a phenotype solution is only as large as the largest sequence of terminal symbols generated by the longest derivation path through the grammar (some studies [18] use this as both a means to limit solution size, and also to ensure validity of mapping, by always using genotype strings sufficiently long to terminate any derivation sequence).

If on the other hand the grammar is recursive, but the genotype is a fixed-length structure (of length l), the maximum phenotype solution length is the longest derivation path through the grammar smaller or equal to l mapping steps (unless wrapping is used).

The use of recursiveness in grammars with GE appears right from its first publication [38], although some interesting GE applications make use of non-recursive grammars, defining either fixed-length solutions (such as the design of a genetic algorithm using GE [28]), or maximum-length bounded solutions (such as the design of an ant-colony optimisation algorithm using GE [40]).

Recursiveness should be applied with care, however. Single levels of recursiveness are easy to implement and understand (see e.g. the second line of G0), but multiple recursive symbols or multiple line recursiveness easily become hard to design and/or understand. See for example the history of attempts at solving the Santa Fe Ant Trail Problem [15] with GE, from the original incorrect grammar [29], to a first [35] and then second [9] correction, and its analysis [26]. And yet, recent publications [17, 18, 21, 42] still use incorrect grammars, not respecting the original problem syntax.

The influence of grammar recursiveness in the ability to terminate mapping, and thus in the effectiveness of GE, has been studied as early as 2003 [39], where productions were categorised based on whether they added, maintained or reduced the number of non-terminal symbols left to map, when applied.

A way to label productions as recursive or not was proposed by Ryan and Azad [37], within the context of initialisation. Subsequently, Harper [9] labelled grammars as *explosive* or *balanced*, depending on whether there is a higher probability of adding non-terminal symbols during mapping over adding terminal symbols.

[1]Technically, genotypes used with GE are length-bounded, in the sense that they cannot be smaller than zero, or larger than what the memory of the machine running the experiments can hold. This maximum size is, however, a technical limitation, rather than a conceptual bound.

```
<s> ::= <e>
<e> ::= <e> <o> <e> | <v>
      | ( <e> <o> <e> ) | <v>
      | <f> ( <e> , <e> ) | <v>
<o> ::= + | - | *
<f> ::= pdiv
<v> ::= x | 1.0
```

Grammar 1 Balanced recursion grammar

These analyses are all related. In this chapter, we define as *explosive* grammars where at least one symbol has more recursive productions than non-recursive. An example is grammar G0: the symbol `<e>` has three recursive productions associated, and a single non-recursive production. This kind of grammar is explosive, and has a low probability of generating a fully mapped phenotype string, when used in conjunction with a randomly-generated genotype string.

For a grammar to be *balanced*, each recursive production should have a corresponding "consuming" production. Grammar 1 (G1) achieves this, by having a non-recursive production for every recursive production associated with `<e>`.

Note that this does not alter any biases, other than that of replacing `<e>` using a recursive or non-recursive production. There is also a downside: G1 now has a 50% probability of generating expressions consisting solely of either `x` or `1.0`.

3.2 Unlinked Productions

When using GE with a linear genome, if the grammar has several non-terminal symbols, the function of a codon (the production it will choose) is dependent on the non-terminal symbol to be mapped, at a given stage of the mapping process. This means that, when crossover occurs, the function of a codon might change.

The potential destructive nature of such changes is addressed in Sect. 3.3, by reducing the number of non-terminal symbols. But if several non-terminal symbols are needed, production rules associated with different symbols can be *functionally linked*, in the sense that all codon values that choose a specific production for a symbol, if used to make a choice for a different symbol, will always choose another specific production.

Take grammar G1 as an example. Symbol `<e>` has six associated productions, which is a multiple of both the number of productions associated with `<o>` (three) and `<v>` (two). So if codon values transforming `<e>` into `<e> <o> <e>` are used to choose a production for `<o>` or `<v>`, they will always transform them into + and `x`, respectively. Table 1 illustrates this.

This can introduce biases in the exploration of the search space. As individuals grow in size (only achievable in standard GE through the use of the crossover operator), a larger proportion of even codon values are required at the start of the genome (see Table 1), and a larger proportion of odd values towards the end (to

Table 1 Functionally-linked productions of grammar G1: codon values transforming `<e>` into `<e> <o> <e>` will always transform `<o>` into + and `<v>` into x; by contrast, `<o>` and `<v>` are unlinked, as codon values transforming `<o>` into + can transform `<v>` into either `x` or `1.0`

Codon values	Parity	`<e>`	`<o>`	`<v>`
0,6,12,...	Even	`<e> <o> <e>`	`+`	`x`
1,7,13,...	Odd	`<v>`	`-`	`1.0`
2,8,14,...	Even	`( <e> <o> <e> )`	`*`	`x`
3,9,15,...	Odd	`<v>`	`+`	`1.0`
4,10,16,...	Even	`<f> ( <e> , <e> )`	`-`	`x`
5,11,17,...	Odd	`<v>`	`*`	`1.0`

terminate the recursion of the `<e>` symbol). However, this will also result in a higher proportion of symbol `x` at the start of the genome, and of `1.0` towards the end. Only through later specific mutation events can suitable proportions of `x` and `1.0` be evolved, assuming the unbalanced solution will survive until then. In other words, the structural function of the `<e>` symbol, and the nodal content function of the `<v>` symbol are linked.

The link between `<e>` and `<o>` is less obvious, but biases also exist: a solution requiring only the use of the `+`, `-` and `*` operators will be biased towards `+` and `*`, when solutions grow in size (first and third productions associated with `<e>`).

Note that these biases can be either beneficial or detrimental, depending on the problem domain. But for a black-box approach, with no domain knowledge, it is desirable to explore the search space in the most unbiased way.

An example is the typical grammar used with GE for the Max problem [8]. Inconsistent results were obtained with different GE mapping orders [4], and further analysis [5] produced no clear explanations for this issue; it was subsequently shown [26] that the grammar used in those experiments suffered from functionally linked productions.

The problem of linked productions was identified as early as 2002 [13]. The solution proposed then was the adoption of a different mapping procedure, called the *Bucket Rule*. However, it was later shown [26] that these biases can also be removed without modifying the standard GE mapping process, through careful grammar design.

Grammar 2 (G2) illustrates how to achieve this, through production rule repetition. In this case, six copies of each production associated with `<o>` are introduced (for a total of $6 * 3 = 18$ productions), whereas 18 copies of each production associated with `<v>` are used, for a total of $18 * 2 = 36$ productions. Note that sufficiently large codon value ranges are required, when using this technique, to ensure minimal production choice biases, but this is good GE practice anyway [31].

```
<s> ::= <e>
<e> ::= <e> <o> <e> | <v>
      | ( <e> <o> <e> ) | <v>
      | <f> ( <e> , <e> ) | <v>
<o> ::= + | + | + | + | + | +
      | - | - | - | - | - | -
      | * | * | * | * | * | *
<f> ::= pdiv
<v> ::= x | x | x | x | x | x
      | x | x | x | x | x | x
      | x | x | x | x | x | x
      | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0
      | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0
      | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0
```

Grammar 2 Unlinked productions grammar

3.3 Reduced Non-terminals

The use of linear genomes and a one point crossover with GE leads to what has been termed the *ripple effect* [32]. This means that, when viewed at a derivation tree level, crossover removes several sub-trees from each parent, which are filled with genetic material from the other parent. Given that the exchanged genetic material consists of a numerical sequence, the actual phenotypic material received may or may not correspond to the original phenotypic material from the other parent: codons reinterpreted under different derivation tree nodes (i.e. non-terminal symbols) will generate different, potentially never-seen before phenotypic material.

This change of interpretation of genetic material can be very damaging to the already fragile locality of crossover in GE [36]. However, many grammars in the literature define more non-terminal symbols than strictly required, creating further cases of reinterpretation of exchanged genetic material.

A solution is thus to reduce the number of non-terminal symbols as much as possible [22]. In fact, for symbolic regression (the most common application domain of GP-like systems [43]), grammars with a single non-terminal symbol can be used (effectively single-type languages, corresponding to GP's closure requirement).

Grammar 3 (G3) shows a single non-terminal symbol version of G1 (using G1 or G2 as a base is irrelevant, given that a single non-terminal symbol remains, `<e>`, so no linked productions will occur). This process works by replacing non-recursively defined non-terminal symbols by all of their productions, wherever they are used, while keeping the biases of the original grammar. A more detailed description and step by step illustration can be found in the relevant publication [22].

```
<e> ::= <e> + <e> | <e> - <e> | <e> * <e>
      | <e> + <e> | <e> - <e> | <e> * <e>
      | x | x | x
      | 1.0 | 1.0 | 1.0
      | ( <e> + <e> ) | ( <e> - <e> ) | ( <e> * <e> )
      | ( <e> + <e> ) | ( <e> - <e> ) | ( <e> * <e> )
      | x | x | x
      | 1.0 | 1.0 | 1.0
      | pdiv ( <e> , <e> ) | pdiv ( <e> , <e> )
      | pdiv ( <e> , <e> )
      | pdiv ( <e> , <e> ) | pdiv ( <e> , <e> )
      | pdiv ( <e> , <e> )
      | x | x | x
      | 1.0 | 1.0 | 1.0
```

Grammar 3 Single non-terminal grammar

```
<e> ::= ( <e> + <e> )
      | <e> + <e>
      | ( <e> - <e> )
      | <e> - <e>
      | ( <e> * <e> )
      | <e> * <e>
      | <e> pdiv <e>
      | ( <e> pdiv <e> )
      | x | x | x | x
      | 1.0 | 1.0 | 1.0 | 1.0
```

Grammar 4 Corrected-biases grammar

3.4 Grammar Biases

Grammars such as G0 are common in the literature. However, it has a 66.666% bias towards the use of one of the operators (+, -, *), resulting in a 22% bias for each, and a 33.333% bias towards the use of pdiv, which may not be desired.

To ensure an unbiased exploration of the search space, all four operators should have the same biases. This also makes the search space more comparable to that of GP. Grammar 4 (G4) shows a transformation of G3 to take this into account.

3.5 Infix/Prefix Notation

From a mathematical point of view, using a single non-terminal symbol grammar, a prefix or postfix notation will essentially produce the same performance (subject to the stochastic nature of the search process), as they explore the same (inverted) syntax space. However, infix will not, if used both with and without parenthesised expressions. For example, a prefix expression *xx may become *x+xx, if the

```
<e> ::= + <e> <e>
      | - <e> <e>
      | * <e> <e>
      | pdiv <e> <e>
      | x | x
      | 1.0 | 1.0
```

Grammar 5 Prefix-notation grammar

```
<e> ::= + <e> <e>
      | - <e> <e>
      | * <e> <e>
      | pdiv <e> <e>
      | <v>
      | <v>
      | <v>
      | <v>
<v> ::= x | x | x | x | x | x | x | x
      | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0
```

Grammar 6 Compromise transformations for a compact, understandable grammar. Note that unlinking of productions associated with `<e>` and `<v>` was required

codon encoding the second argument is mutated; with infix notation, however, `x*x` can become either `x*(x+x)`, which is equivalent, or `x*x+x`, which is not.

There is no obvious choice to make here. Infix provides a more connected search space, but at the expense of further loss of locality for the genetic operators; prefix/postfix do the opposite. They also provide a more comparable search space to that of tree-based GP. Grammar 5 (G5) shows a prefix version of G4.

3.6 Compromise Grammars

Although the transformations presented can be achieved through an algorithmic process (and thus automated), the resulting grammars can easily grow exponentially in both size and complexity, and become very hard to understand or modify. The addition of a carefully chosen single non-terminal symbol (`<v>` in this case) is often enough to maintain the readability of a grammar, at the expense of a slight worsening of crossover locality. Grammar 6 (G6) illustrates this. Although not apparent when compared to G5, in problems with large numbers of operators and variables, this can drastically reduce the number of productions in a grammar (see Table 5, grammars G5 (5625 productions) and G6 (17 productions)).

4 Transformations Analysis

A series of detailed experiments were performed, using grammars G0–G6, to analyse the effect of grammar design on search space biases, size, termination, repetition, and performance. Table 2 shows the experimental setup used.

Populations were initialised using Random Initialisation (RND) (random integer strings), or using a depth-less variant of Probabilistic Tree Creation 2 (PTC2) [19, 23]. These were chosen as many recent publications still use RND initialisation [17, 21], whereas PTC2 was chosen for its proven performance [23]. Depending on the grammar used, the specified number of codons/expansions for initialisation is sufficient to generate expressions of up to a corresponding syntax tree depth of 5.

4.1 Initialisation Biases

This first experiment analyses the initial populations generated using the two initialisation methods. The proportion of each phenotypic symbol (`+`, `-`, `*`, `pdiv`, `1.0` (const) and `x` (var)) in all successfully mapped individuals was recorded, along with measures related to the mapping process: average phenotype length, number of invalid (non-mapping) individuals generated, and number of repeated (valid) phenotypic solutions. Figure 1 shows the results obtained, for 100 independent runs.

The top half of the figure illustrates how grammar design affects the initial sampling of the search space. G0–G3 exhibit a bias towards the use of division. In the first three grammars, this is because 2/3 of the recursive definitions of the `<e>` symbol use a function of the set (`+`, `-`, `*`), whereas 1/3 use division; this results in a biased sampling of $2/3 \div 3 = 2/9$ for each of (`+`, `-`, `*`), and 3/9 for division. These biases are held with the reduction of non-terminal symbols used in G3.

Table 2 Experimental setup

Population size	500[a]
Number of generations	50
Random initialisation genome length	31
PTC2 max expansions	31
Maximum genome length	–
Selection tournament size	1%
Elitism (for generational replacement)	1%
Crossover ratio	50%
Average mutation events per individual	1
Max wrapping events	0

[a]200 for Shape Match (Easy and Medium) (see Sect. 5); 1000 for V4, K12, Housing, EPar5 and Mux11 (see Sect. 5); 2000 for Dow (see Sect. 5)

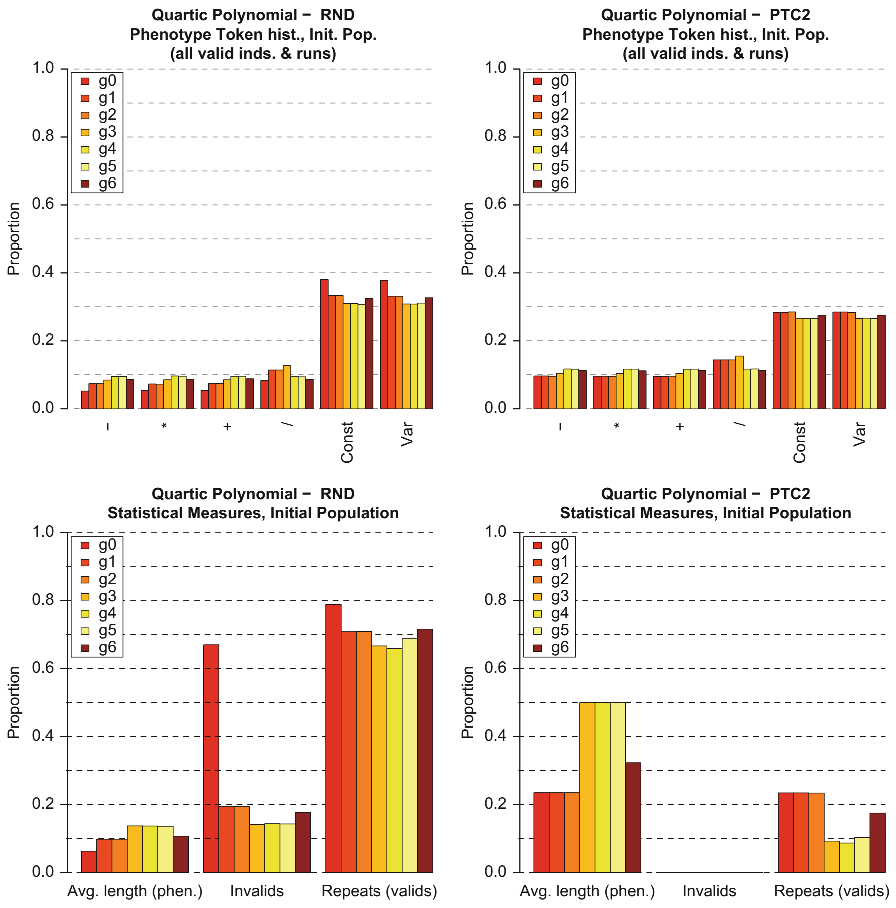

Fig. 1 Symbol frequency proportions (top) in the initial populations (measured over all successfully mapped solutions), and mapping process related statistics (bottom), using RND (left) and PTC2 (right). Results obtained from 100 independent runs

Other biases are seen in this figure. G0–G2 and G6, due to their larger number of non-terminal symbols, generate smaller expressions than the other grammars, when provided with the same number of genes (RND) or expansions (PTC2); this explains their slightly smaller frequency of functions, and higher frequency of terms (`1.0` and `x`). In any case, all grammars exhibit a larger proportion of terms in their phenotypes, when using RND; this is due to the very high probability of transforming an `<e>` symbol into a term (1/4 for G0, and 1/2 for all the other grammars), meaning many solutions will consist of a single term. The use of PTC2 substantially reduces the appearance of such solutions.

The bottom half of the figure shows statistics related to the mapping process. The average length is a ratio of the average number of functions/terms in the generated phenotypes (e.g. `1.0 + x` and `(1.0 + x)` both have length 3) over 31. RND creates much shorter solutions than PTC2, due to the 50% probability of creating very short solutions. As for G0, although it is biased towards larger solutions, it usually cannot create such solutions, due to its reduced probability of terminating the mapping process. In any case, G0–G2 and G6 create shorter phenotypes, due to the intermediate symbols used during the mapping process.

When using RND, the proportion of non-mapping individuals is alarmingly high for explosive G0. Although far lower, G1, G2 and G6 also have some difficulty to map integer strings, due to the number of derivation steps required to terminate the mapping process (a problem shared with G0). G3–G5 map around 85% of the random integer strings. Naturally, all phenotypes generated by PTC2 are valid.

Within the successfully-mapped individuals, there is a very high proportion of repeated solutions with RND, again a consequence of the high probability of generating single term solutions. This is slightly higher for G0, due to the increasing difficulty in successfully mapping long individuals. PTC2, ramping by size, generates longer solutions, and thus generates less repetition.

4.2 Random Walk Biases

This second experiment tests the effect of recombination and mutation on the initial populations seen in the previous experiment. To this end, 50 generations were performed in each run, but all individuals (mapping or not) were assigned the same flat fitness. This effectively means random walks are performed, biased both by grammar design, and by the genetic operators used (linear crossover and integer mutation). Figure 2 shows the results obtained, at the last generation.

The top half of the figure shows no major differences between RND and PTC2. There is an even stronger bias towards using `+` and `1.0`, due to shorter solutions being generated. However, the biases introduced by grammar design are still quite present, particularly towards division, in G0–G3.

The bottom plot shows that grammar design affects the dissemination of illegal individuals by a large amount: by generation 50, over 90% of individuals in runs using G0 are invalid, due to its non-terminating bias. G1, G2 and G6 exhibit $\approx$ 70% invalids, suffering from their complex, many non-terminal symbols mapping (a problem shared with G0). G3–G5 have an expected 50% chance of generating mapping individuals under random search. This is the case using both RND and PTC2, which shows that this bias exists irrespective of the starting population.

Finally, there is a huge amount of repetition in the (valid) solutions generated. The majority are short, single term solutions, as they are more likely to survive unharmed (and unchanged) genetic operators applied with no fitness pressure.

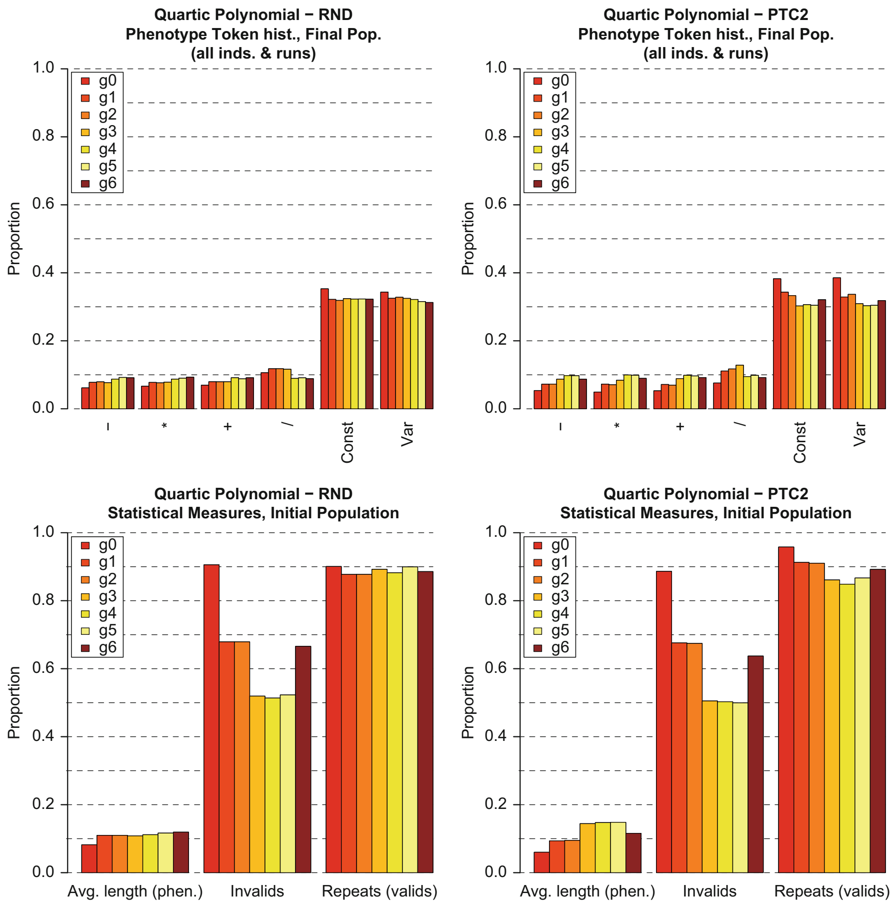

Fig. 2 Symbol frequency proportions (top) in the final populations (measured over all successfully mapped solutions), and mapping process related statistics (bottom), using RND (left) and PTC2 (right). Results obtained from 100 independent runs

4.3 Termination Biases

Mapping termination has always been a hotly debated topic in GE. This third experiment investigates how it is influenced by grammar design. The experimental setup is the same as in the previous sub-section, except that non-mapping individuals are assigned a very bad fitness score (the usual way of dealing with non-mapping solutions in GE [30]), whereas all others are assigned a fixed (good) fitness. The results obtained are shown in Fig. 3.

The top half of the figure shows once again that, irrespectively of using RND or PTC2, symbol biases are practically the same. The previously analysed bias towards division is still visible in G0–G3. Particularly worrying is the high bias of `1.0` over

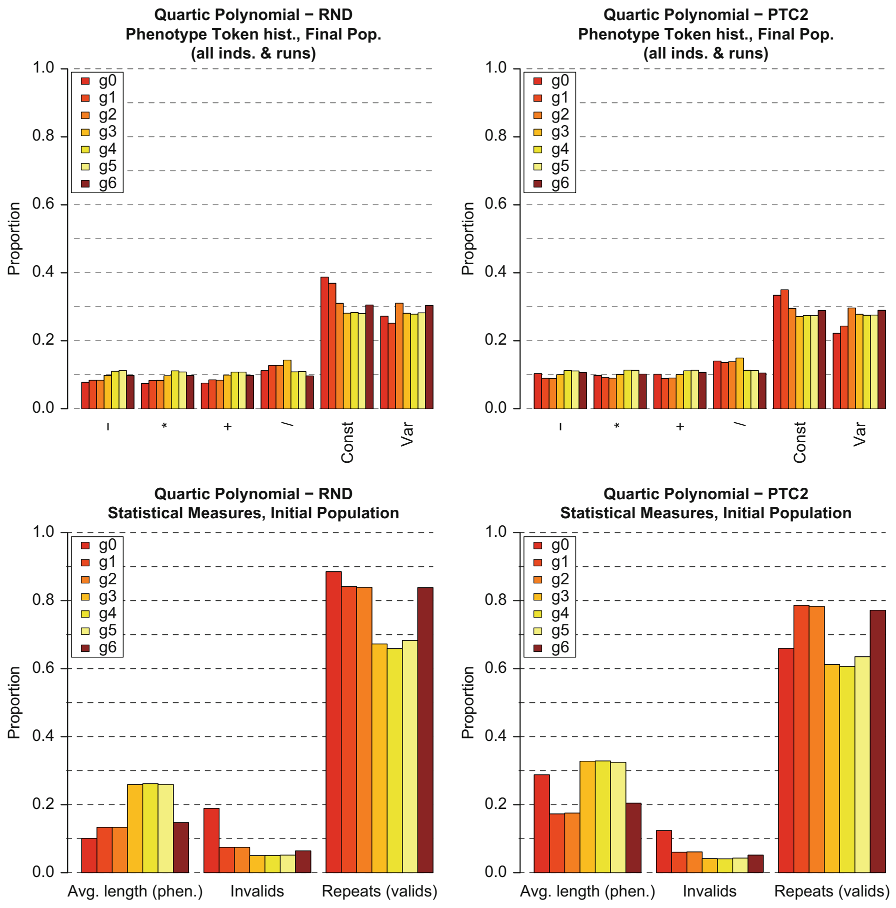

Fig. 3 Symbol frequency proportions (top) in the final populations (measured over all successfully mapped solutions), and mapping process related statistics (bottom), using RND (left) and PTC2 (right). Results obtained from 100 independent runs. Non-mapping solutions were assigned the worst possible fitness

`x` in G0 and G1. This is a direct reflection of the propagation and recombination of mapping individuals, in combination with linked grammar productions. In G0, a large amount of codons with a value $c_i\%4 = 3$ are required, to choose the recursion stopping production "`<e>` becomes `<v>`"; but those same codons, when interpreted under the context of `<v>`, will choose the term `1.0`. The same effect is seen in G1: odd codon values are required to stop recursion of the `<e>` symbol, but these will choose the symbol `1.0` over the symbol `x` (see Table 1). These problems are removed by unlinking production choices (grammars G2–G6).

The use of bad fitness for non-mapping individuals is effective at reducing the number of illegal solutions, without requiring the propagation of extremely small solutions. There is still a large proportion of repetition, driven in this case by genetic drift: similar individuals undergoing crossover are less likely to produce illegal offspring, leading to increased repetition in the final population. Finally, note the smaller difference between RND and PTC2 at the end of these runs.

4.4 Performance Biases

To test how these findings affect performance, a series of symbolic regression experiments were ran with all seven grammars, to solve the following problems:

1. Quartic Polynomial: $x^4 + x^3 + x^2 + x$;
2. Sextic Polynomial: $x^6 + x^5 + \cdots + x^2 + x$;
3. Octic Polynomial: $x^8 + x^7 + \cdots + x^2 + x$;
4. Dectic Polynomial: $x^{10} + x^8 + \cdots + x^2 + x$.

These problems are easy to solve, with a controlled degree of difficulty, and can be correlated to the effects of symbol biases. Note that fully configured GE runs were performed; this included the use of tails [26] at a 50% ratio at initialisation, for better mapping termination. Figure 4 shows the measured biases and statistics for the quartic and dectic polynomial, using PTC2 (RND results were similar).

There is a clear bias towards the use of multiplication (mostly), addition, and `x`, the symbols required to solve the problem. The smaller uses of division, subtraction, and `1.0` are mostly due to non-effective code (bloat). It is interesting to observe that a bias towards division is still found when using G3. Solution length is short, a reflection of GE's mapping process and also the easy nature of the problems, with the downside of still a large amount of repetition in the last generation (due both to genetic drift and to smaller solutions being generated). Finally, experiments using G0 still generate a large amount of illegal solutions.

Figure 5 plots the results obtained. Most configurations solve the easier problem on every run, but as the polynomial degree increases, performance slowly worsens. Results using PTC2 are better than those using RND, as expected [23].

The graphs also illustrate how grammar design can affect (or not) the performance of GE. The most obvious observation is that the reduction of grammar complexity and the associated termination biases can vastly improve performance: setups using G0–G2 are consistently worse than all other setups. Also interesting is how the bias of G2 and G3 towards the use of division has almost no effect on performance. This is because division can be used almost as effectively as multiplication to increase the degree of the polynomial ($x \times x$ versus $\frac{x}{1/x}$).

Finally, the relative differences between grammars are consistent across the two initialisers, apart from G0: the lack of invalid solutions in the initial population of PTC2, along with its larger initial solution size (see Fig. 1), substantially improve the final performance of runs using G0.

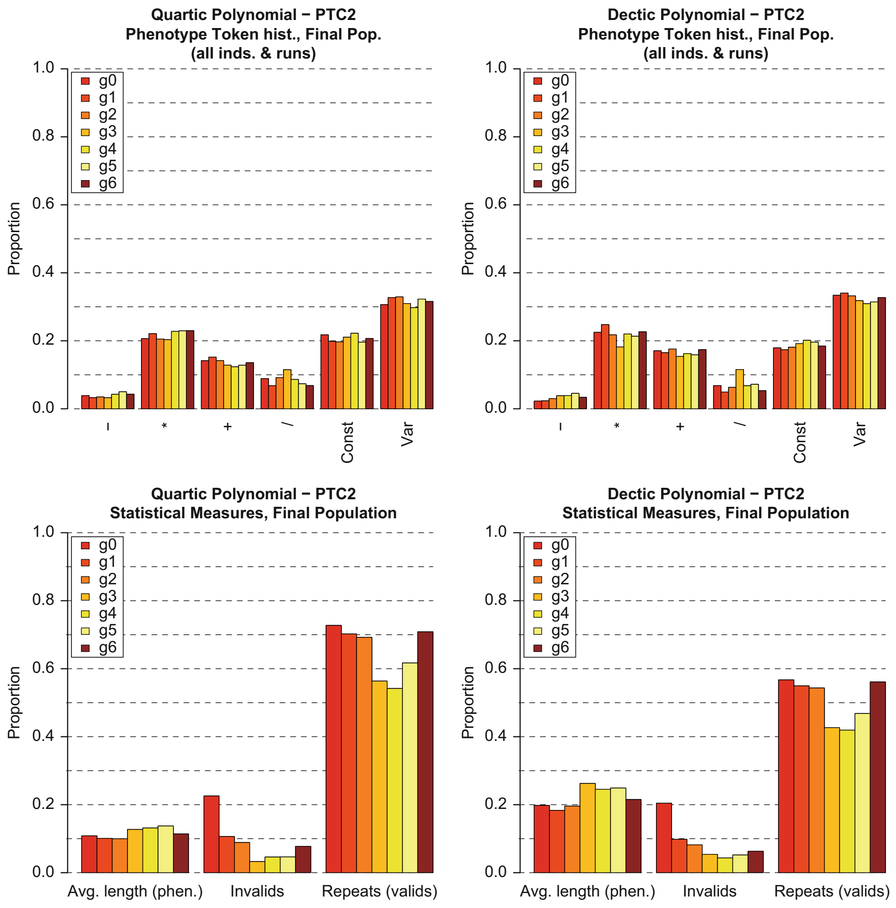

Fig. 4 Symbol frequency proportions (top) in the final populations (measured over all successfully mapped solutions), and mapping process related statistics (bottom), using PTC2, on quartic (left) and dectic (right) polynomial. Results obtained from 100 independent runs

5 Performance Analysis

5.1 Problems

To measure the effect of grammar design on the final outcome of GE's evolutionary process, a series of experiments were ran across several problem types. These include regression, classification and design problems, across several application domains and difficulty ranges. Table 3 lists all the problems attempted.

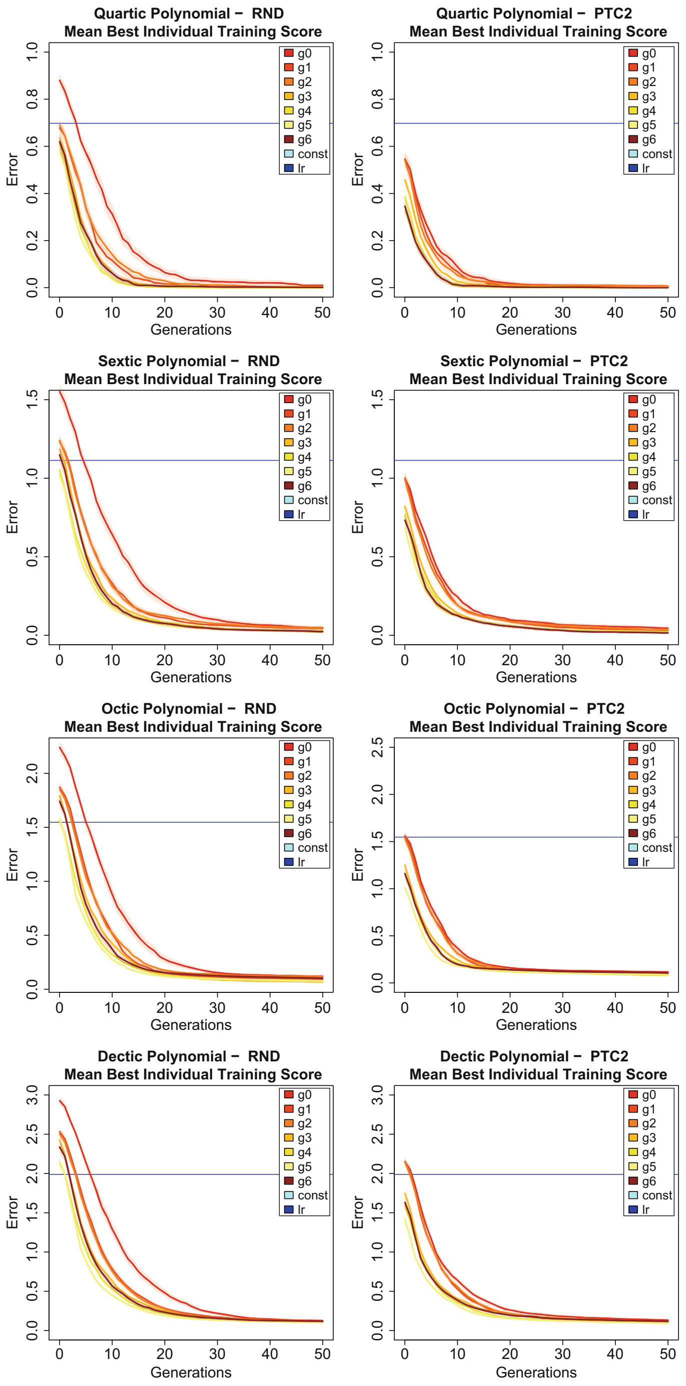

Fig. 5 Mean best individual fitness for the polynomial regression experiments, using RND (left) and PTC2 (right), averaged across 100 independent runs; error is measured as RMSE. Shades indicate 95% confidence intervals about the mean. Cyan and blue lines (where visible) show the RMSE of a constant predictor (mean of the train response variable) and a linear regression model of the training set, respectively

Table 3 Benchmark problems; if specified, $E[a, b, c]$ means a grid of points evenly spaced with an interval of c, from a to b inclusive, whereas $U[a, b, c]$ means c uniform random samples drawn from a to b inclusive; the specified type is regression (R), classification (C#) (# = number of classes), or image matching (IM)

Name	Vars.	Data source	Type	Training set / Test Set
Keijzer-6 (K6) [12]	1	$y = \sum_i^x \frac{1}{i}$	R	$E[1, 50, 1]$
				$E[1, 120, 1]$
Pagie-1 (P1) [34]	2	$y = \frac{1}{1+x_1^{-4}} + \frac{1}{1+x_2^{-4}}$	R	$E[-5, 5, 0.4]$
				$E[-5, 5, 0.1]$
Vladislavleva-4 (V4) [41]	5	$y = \frac{10}{5+\sum_{i=1}^{5}(x_i-3)^2}$	R	$U[0.05, 0.05, 1024]$
				$U[-0.25, 6.35, 5k]$
Tower [41]	5	Gas chromatography data	R	4721 points
				278 points
Korns-12 (K12) [14]	5	$y = 2 - 2.1cos(9.8x_1)sin(1.3x_5)$	R	$U[-50, 50, 10k]$
				$U[-50, 50, 10k]$
Forest [2]	12	Forest Fires data	R	414 points
				103 points
Housing [16]	13	Housing values	R	354 points
				152 points
Dow Chemical (Dow)[a]	57	Chemical process data	R	747 points
				319 points
Even-parity 5 (EPar5) [15]	5	Parity of boolean inputs	C2	32 points
				–
Multiplexer 11 (Mux11) [15]	11	Boolean multiplexer	C2	2048 points
				–
Breast Cancer Wisconsin [16]	9	Diagnostic data	C2	400 points
				283 points
Wine Quality: red [3]	11	Physicochemical test data	C10	1000 points
				599 points
Wine Quality: white [3]	11	Physicochemical test data	C10	3000 points
				1898 points
Shape Match (Easy) [33]	–	Generated 250x250 shape	IM	–
				–
Shape Match (Medium) [33]	–	Generated 250x250 shape	IM	–
				–
Shape Match (Hard) [33]	–	Generated 250x250 shape	IM	–
				–

[a]Source: http://gpbenchmarks.org/symbolicregressioncompetition/

5.2 Grammars

For every problem attempted, seven grammar versions were designed, G0–G6, as detailed in Sect. 3. Each of the grammars respects as much as possible the original function and terminal sets as defined in their original publications; this is important for benchmarking purposes, as attempting to solve a problem using different functions can severely alter the difficulty of the benchmarks [27]. For all problems, constants were created using digit concatenation, with 100 possible values within the range [0.0 . . . 9.9], and a step of 0.1.

Each of the problems was attempted using both RND and PTC2, over 100 independent runs. To illustrate the grammar transformation process, and its impact on the search effectiveness of GE, three problems are examined in detail: Keijzer-6, Vladislavleva-4, and Shape Match (hard).

5.2.1 Keijzer-6

The Keijzer-6 (K6) problem [12], also known as the Harmonic function, is a single-variable problem, using only addition, multiplication and three unary functions. A typical GE grammar defining possible solutions for this problem is shown in Table 4, cell G0. The symbol `<e>` has more productions creating new `<e>` symbols rather than mapping them to something else, so the G1 version addresses this, by having a "| `<v>`" production for each production replacing one `<e>` symbol with two. There are two sets of non-terminal symbols with linked productions: `<o>` with `<v>`, and `<e>` with `<d>`. This is addressed in G2, through repetition of productions.

G3 reduces the number of non-terminal symbols to a single one, while maintaining the biases of G1 and G2. This results in a much larger grammar, with 3000 production rules. Grammar G4 addresses the slightly higher bias towards use of the operators $+$ and $*$ over the functions inv, neg and $sqrt$.

Grammar G5 is a conversion to prefix notation, which considerably reduces the complexity of the grammar, by removing bracketed versions of the $+$ and $*$ operators. Finally, G6 separates the definition of functions and terminals into two symbols (`<e>` and `<v>`), resulting in a very compact grammar, but without the extra complexity of grammars G0–G2.

5.2.2 Vladislavleva-4

The Vladislavleva-4 (V4) problem [41], also known as the UBall5D function, is a five-variable problem, and its function set, as defined in its original publication, is extensive:

- Functions: $+, -, *, /, square, x^{real}, x + real, x \cdot real$
- Terminals: $x0, x1, x2, x3, x4$

Table 4 Grammars used for the Keijzer-6 (K6) problem, versions G0–G6; a set of productions followed by $\{n\}$ means they are repeated n times, whereas the notation $\alpha|\ldots|\delta$ is shorthand for a production for each of the elements in the sequence $[\alpha..\delta]$; constants in G6 are created using `GECodonValue` [24]

G0	G1	G2
<s>::=<e> <e>::=<e><o><e> |(<e><o><e>) | <f> (<e>) | <v> <o>::= +|* <f>::= *inv*|*neg*|*sqrt* <v>::=<d>.<d> |*x*0 <d>::= 0|1|2|3|4|5|6|7|8|9	<s>::=<e> <e>::=<e><o><e> | <v> |(<e><o><e>) | <v> | <f> (<e>) <o>::= +|* <f>::= *inv*|*neg*|*sqrt* <v>::=<d>.<d> |*x*0 <d>::= 0|1|2|3|4|5|6|7|8|9	<s>::=<e> <e>::=<e><o><e> | <v> | (<e><o><e>) | <v> | <f> (<e>) <o>::= +|+ | * |* <f>::= *inv*|*neg*|*sqrt* <v>::=<d>.<d> | *x*0 <d>::= 0|0|0|0|0|1|1|1|1|1 |2|2|2|2|2|3|3|3|3|3 |4|4|4|4|4|5|5|5|5|5 |6|6|6|6|6|7|7|7|7|7 |8|8|8|8|8|9|9|9|9|9
G3	**G4**	**G5**

<e>::=<e> + <e> {300}
| <e> * <e> {300}
| *x*0 {300}
| 0.0| . . . |9.9 {3}
| (<e> + <e>) {300}
| (<e> * <e>) {300}
| *x*0{300}
| 0.0| . . . |9.9 {3}
| *inv*(<e>) {200}
| *neg*(<e>) {200}
| *sqrt*(<e>) {200}
	<e>::=<e> + <e> {300}
	| <e> * <e> {300}
	| *x*0 {300}
	| 0.0| . . . |9.9 {3}
	| (<e> + <e>) {300}
	| (<e> * <e>) {300}
	| *x*0{300}
	| 0.0| . . . |9.9 {3}
	| *inv*(<e>) {300}
	| *neg*(<e>) {300}
	| *sqrt*(<e>) {300}
		<e>::= + <e><e> {100}
		| * <e><e> {100}
		| *x*0 {100}
		| 0.0| . . . |9.9
		| *inv*(<e>) {100}
		| *neg*(<e>) {100}
		| *sqrt*(<e>) {100}

G6		
<e>::= + <e><e> | <v> | * <e><e> | <v> |*inv* <e> |*neg* <e> |*sqrt* <e> <v>::= *x*0| <*GECodonValue*{0.0 : 9.9 : 0.1}>		

This results in a complex base grammar, shown in Table 5, cell G0. The definition of the `<e>` symbol has a heavy bias towards growth, which is addressed in G1. It also has a more complex set of linked production, between the symbols `<a>` (five productions), `<v>` (five productions) and `<d>` (ten productions); G2 addresses this by introducing 10 copies of each original production associated with `<a>`, and 50 copies of each original production associated with `<v>`.

As before, G3 reduces the number of non-terminal symbols to just one, but to ensure the same bias choices as G1 and G2, and due to the required combinations of 5 variables and 100 constants for certain operators, it defines 17,498 production rules.[2] Grammar G4 balances the bias between all four binary operators ($+$, $-$, $*$, $/$), with 1250 productions using each (625 bracketed and 625 non-bracketed), and the single unary operator, *square* (625 productions); the resulting grammar, while smaller, still contains 10,625 productions.

[2]The required number of $2500/3$ copies of each of the productions using the operators $+, -, *$ were rounded to 833, resulting in a negligible bias.

Table 5 Grammars used for the Vladislavleva-4 (V4) problem, versions G0–G6; a set of productions followed by $\{n\}$ means they are repeated n times, whereas the notation $\alpha \vert \ldots \vert \delta$ is shorthand for a production for each of the elements in the sequence $[\alpha..\delta]$; constants in G6 are created using `GECodonValue` [24]

G0	G1	G2
$\langle s\rangle ::= \langle e\rangle$	$\langle s\rangle ::= \langle e\rangle$	$\langle s\rangle ::= \langle e\rangle$
$\langle e\rangle ::= \langle e\rangle \langle o\rangle \langle e\rangle$	$\langle e\rangle ::= \langle e\rangle \langle o\rangle \langle e\rangle$	$\langle e\rangle ::= \langle e\rangle \langle o\rangle \langle e\rangle$
$\vert (\langle e\rangle \langle o\rangle \langle e\rangle)$	$\vert\ \langle c\rangle$	$\vert\ \langle c\rangle$
$\vert\ \langle f1\rangle\ (\langle e\rangle, \langle e\rangle)$	$\vert (\langle e\rangle \langle o\rangle \langle e\rangle)$	$\vert (\langle e\rangle \langle o\rangle \langle e\rangle)$
$\vert\ \langle f2\rangle\ (\langle e\rangle, 2)$	$\vert\ \langle c\rangle$	$\vert\ \langle c\rangle$
$\vert\ \langle c\rangle$	$\vert\ \langle f1\rangle\ (\langle e\rangle, \langle e\rangle)$	$\vert\ \langle f1\rangle\ (\langle e\rangle, \langle e\rangle)$
$\langle o\rangle ::= + \vert - \vert *$	$\vert\ \langle c\rangle$	$\vert\ \langle c\rangle$
$\langle f1\rangle ::= \mathit{pdiv}$	$\vert\ \langle f2\rangle\ (\langle e\rangle, 2)$	$\vert\ \langle f2\rangle\ (\langle e\rangle, 2)$
$\langle f2\rangle ::= \mathit{pow}$	$\langle o\rangle ::= + \vert - \vert *$	$\langle o\rangle ::= + \vert - \vert *$
$\langle c\rangle ::= \mathit{pow}(\langle v\rangle, \langle d\rangle.\langle d\rangle)$	$\langle f1\rangle ::= \mathit{pdiv}$	$\langle f1\rangle ::= \mathit{pdiv}$
$\vert (\langle v\rangle + \langle d\rangle.\langle d\rangle)$	$\langle f2\rangle ::= \mathit{pow}$	$\langle f2\rangle ::= \mathit{pow}$
$\vert (\langle v\rangle * \langle d\rangle.\langle d\rangle)$	$\langle c\rangle ::= \mathit{pow}(\langle v\rangle, \langle d\rangle.\langle d\rangle)$	$\langle c\rangle ::= \mathit{pow}(\langle v\rangle, \langle d\rangle.\langle d\rangle)\{10\}$
$\vert\ \langle v\rangle$	$\vert (\langle v\rangle + \langle d\rangle.\langle d\rangle)$	$\vert (\langle v\rangle + \langle d\rangle.\langle d\rangle)\{10\}$
$\vert\ \langle d\rangle.\langle d\rangle$	$\vert (\langle v\rangle * \langle d\rangle.\langle d\rangle)$	$\vert (\langle v\rangle * \langle d\rangle.\langle d\rangle)\{10\}$
$\langle v\rangle ::= x0 \vert x1 \vert x2 \vert x3 \vert x4$	$\vert\ \langle v\rangle$	$\vert\ \langle v\rangle\ \{10\}$
$\langle d\rangle ::= 0 \vert 1 \vert 2 \vert 3 \vert 4 \vert 5 \vert 6 \vert 7 \vert 8 \vert 9$	$\vert\ \langle d\rangle.\langle d\rangle$	$\vert\ \langle d\rangle.\langle d\rangle\ \{10\}$
	$\langle v\rangle ::= x0 \vert x1 \vert x2 \vert x3 \vert x4$	$\langle v\rangle ::= x0 \vert x1 \vert x2 \vert x3 \vert x4\{50\}$
	$\langle d\rangle ::= 0 \vert 1 \vert 2 \vert 3 \vert 4 \vert 5 \vert 6 \vert 7 \vert 8 \vert 9$	$\langle d\rangle ::= 0 \vert 1 \vert 2 \vert 3 \vert 4 \vert 5 \vert 6 \vert 7 \vert 8 \vert 9$

G3	G4	G5
$\langle e\rangle ::= \langle e\rangle + \langle e\rangle\ \{833\}$	$\langle e\rangle ::= \langle e\rangle + \langle e\rangle\ \{625\}$	$\langle e\rangle ::= + \langle e\rangle \langle e\rangle\ \{625\}$
$\vert\ \langle e\rangle - \langle e\rangle\ \{833\}$	$\vert\ \langle e\rangle - \langle e\rangle\ \{625\}$	$\vert - \langle e\rangle \langle e\rangle\ \{625\}$
$\vert\ \langle e\rangle * \langle e\rangle\ \{833\}$	$\vert\ \langle e\rangle * \langle e\rangle\ \{625\}$	$\vert * \langle e\rangle \langle e\rangle\ \{625\}$
$\vert (\langle e\rangle + \langle e\rangle)\ \{833\}$	$\vert\ \langle e\rangle\ \mathit{pdiv}\ \langle e\rangle\ \{625\}$	$\vert \mathit{pdiv}\ \langle e\rangle \langle e\rangle\ \{625\}$
$\vert (\langle e\rangle - \langle e\rangle)\ \{833\}$	$\vert (\langle e\rangle + \langle e\rangle)\ \{625\}$	$\vert \mathit{pow}\ \langle e\rangle\ 2\ \{625\}$
$\vert (\langle e\rangle * \langle e\rangle)\ \{833\}$	$\vert (\langle e\rangle - \langle e\rangle)\ \{625\}$	$\vert \mathit{pow}\ x0\ 0.0 \vert \ldots \vert \mathit{pow}\ x0\ 9.9$
$\vert \mathit{pdiv}(\langle e\rangle, \langle e\rangle)\ \{2500\}$	$\vert (\langle e\rangle * \langle e\rangle)\ \{625\}$	$\vert \mathit{pow}\ x1\ 0.0 \vert \ldots \vert \mathit{pow}\ x1\ 9.9$
$\vert \mathit{pow}(\langle e\rangle, 2)\ \{2500\}$	$\vert (\langle e\rangle\ \mathit{pdiv}\ \langle e\rangle)\ \{625\}$	$\vert \mathit{pow}\ x2\ 0.0 \vert \ldots \vert \mathit{pow}\ x2\ 9.9$
$\vert \mathit{pow}(x0, 0.0) \vert \ldots \vert \mathit{pow}(x0, 9.9)\ \{3\}$	$\vert \mathit{pow}(\langle e\rangle, 2)\ \{2500\}$	$\vert \mathit{pow}\ x3\ 0.0 \vert \ldots \vert \mathit{pow}\ x3\ 9.9$
$\vert \mathit{pow}(x1, 0.0) \vert \ldots \vert \mathit{pow}(x1, 9.9)\ \{3\}$	$\vert \mathit{pow}(x0, 0.0) \vert \ldots \vert \mathit{pow}(x0, 9.9)\ \{2\}$	$\vert \mathit{pow}\ x4\ 0.0 \vert \ldots \vert \mathit{pow}\ x4\ 9.9$
$\vert \mathit{pow}(x2, 0.0) \vert \ldots \vert \mathit{pow}(x2, 9.9)\ \{3\}$	$\vert \mathit{pow}(x1, 0.0) \vert \ldots \vert \mathit{pow}(x1, 9.9)\ \{2\}$	$\vert + x0\ 0.0 \vert \ldots \vert + x0\ 9.9$
$\vert \mathit{pow}(x3, 0.0) \vert \ldots \vert \mathit{pow}(x3, 9.9)\ \{3\}$	$\vert \mathit{pow}(x2, 0.0) \vert \ldots \vert \mathit{pow}(x2, 9.9)\ \{2\}$	$\vert + x1\ 0.0 \vert \ldots \vert + x1\ 9.9$
$\vert \mathit{pow}(x4, 0.0) \vert \ldots \vert \mathit{pow}(x4, 9.9)\ \{3\}$	$\vert \mathit{pow}(x3, 0.0) \vert \ldots \vert \mathit{pow}(x3, 9.9)\ \{2\}$	$\vert + x2\ 0.0 \vert \ldots \vert + x2\ 9.9$
$\vert (x0 + 0.0) \vert \ldots \vert (x0 + 9.9)\ \{3\}$	$\vert \mathit{pow}(x4, 0.0) \vert \ldots \vert \mathit{pow}(x4, 9.9)\ \{2\}$	$\vert + x3\ 0.0 \vert \ldots \vert + x3\ 9.9$
$\vert (x1 + 0.0) \vert \ldots \vert (x1 + 9.9)\ \{3\}$	$\vert (x0 + 0.0) \vert \ldots \vert (x0 + 9.9)\ \{2\}$	$\vert + x4\ 0.0 \vert \ldots \vert + x4\ 9.9$
$\vert (x2 + 0.0) \vert \ldots \vert (x2 + 9.9)\ \{3\}$	$\vert (x1 + 0.0) \vert \ldots \vert (x1 + 9.9)\ \{2\}$	$\vert * x0\ 0.0 \vert \ldots \vert * x0\ 9.9$
$\vert (x3 + 0.0) \vert \ldots \vert (x3 + 9.9)\ \{3\}$	$\vert (x2 + 0.0) \vert \ldots \vert (x2 + 9.9)\ \{2\}$	$\vert * x1\ 0.0 \vert \ldots \vert * x1\ 9.9$
$\vert (x4 + 0.0) \vert \ldots \vert (x4 + 9.9)\ \{3\}$	$\vert (x3 + 0.0) \vert \ldots \vert (x3 + 9.9)\ \{2\}$	$\vert * x2\ 0.0 \vert \ldots \vert * x2\ 9.9$
$\vert (x0 * 0.0) \vert \ldots \vert (x0 * 9.9)\ \{3\}$	$\vert (x4 + 0.0) \vert \ldots \vert (x4 + 9.9)\ \{2\}$	$\vert * x3\ 0.0 \vert \ldots \vert * x3\ 9.9$
$\vert (x1 * 0.0) \vert \ldots \vert (x1 * 9.9)\ \{3\}$	$\vert (x0 * 0.0) \vert \ldots \vert (x0 * 9.9)\ \{2\}$	$\vert * x4\ 0.0 \vert \ldots \vert * x4\ 9.9$
$\vert (x2 * 0.0) \vert \ldots \vert (x2 * 9.9)\ \{3\}$	$\vert (x1 * 0.0) \vert \ldots \vert (x1 * 9.9)\ \{2\}$	$\vert x0 \vert x1 \vert x2 \vert x3 \vert x4\ \{100\}$
$\vert (x3 * 0.0) \vert \ldots \vert (x3 * 9.9)\ \{3\}$	$\vert (x2 * 0.0) \vert \ldots \vert (x2 * 9.9)\ \{2\}$	$\vert 0.0 \vert \ldots \vert 9.9\ \{5\}$
$\vert (x4 * 0.0) \vert \ldots \vert (x4 * 9.9)\ \{3\}$	$\vert (x3 * 0.0) \vert \ldots \vert (x3 * 9.9)\ \{2\}$	
$\vert x0 \vert x1 \vert x2 \vert x3 \vert x4\ \{300\}$	$\vert (x4 * 0.0) \vert \ldots \vert (x4 * 9.9)\ \{2\}$	
$\vert 0.0 \vert \ldots \vert 9.9\ \{15\}$	$\vert x0 \vert x1 \vert x2 \vert x3 \vert x4\ \{200\}$	
	$\vert 0.0 \vert \ldots \vert 9.9\ \{10\}$	

G6
$\langle e\rangle ::= + \langle e\rangle \langle e\rangle$
$\vert - \langle e\rangle \langle e\rangle$
$\vert * \langle e\rangle \langle e\rangle$
$\vert \mathit{pdiv}\ \langle e\rangle \langle e\rangle$
$\vert \mathit{pow}\ \langle e\rangle\ 2$
$\vert \mathit{pow}\ \langle v\rangle \langle GECodonValue\{0.0 : 9.9 : 0.1\}\rangle$
$\vert + \langle v\rangle \langle GECodonValue\{0.0 : 9.9 : 0.1\}\rangle$
$\vert * \langle v\rangle \langle GECodonValue\{0.0 : 9.9 : 0.1\}\rangle$
$\vert\ \langle v\rangle$
$\vert\ \langle v\rangle$
$\vert\ \langle GECodonValue\{0.0 : 9.9 : 0.1\}\rangle$
$\vert\ \langle GECodonValue\{0.0 : 9.9 : 0.1\}\rangle$
$\langle v\rangle ::= x0 \vert x1 \vert x2 \vert x3 \vert x4$

The conversion to a prefix notation in G5 further reduces complexity, but it still contains 5625 productions. Finally, the separation of variables to a different symbol (`<v>`) removes the complexity of variable and constant combination, and the resulting grammar is far more compact and understandable.

5.2.3 Shape Match (Hard)

The Shape Match problem [33] was setup as a demonstration of the use of shape grammars with GE. The objective is to match a pre-defined image, defined in a 250x250 binary pixel matrix, using a sequence of shape creation and manipulation instructions. It was defined using three variants, *easy*, *medium* and *hard*, with increasingly more complex target images. The drawing functions available are as follows: `s0` moves the shape right 10 pixels; `s1` moves the shape down 10 pixels; `s2` moves the shape left 10 pixels; `s3` moves the shape up 10 pixels; `gro` doubles the size of the shape; `shrnk` halves the size of the shape; `[` and `]` push and pop the pen's state (position where it will draw next) onto and off the stack. Finally, `sqr` draws a square, and `crcl` draws a circle.

Table 6 shows the original grammar [33] as G0, with a call to a Python interpreter. It has no recursively defined binary operators: a single call to `<p>::=<e>` stops recursion of symbol `<p>`, and likewise, a single call to `<e>::=<v>` terminates the recursion of `<e>`. As such, G1 is identical to G0. There are, however, two sets of non-terminal symbols with linked productions: `<p>` with `<v>`, and `<e>` with `<o>`. G2 addresses this, through the explained approach of repetition of productions.

G3 reduces the number of non-terminal symbols to three. Although the first symbol (`<s>`) is of minor importance (it has a single associated production, so no codon is used, and it appears only once, at the start), the recursive nature of symbols `<p>` and `<e>` in G0–G2 requires the presence of both. Also, after the incorporation of symbols `<o>` and `<v>` into `<e>`, the resulting number of productions is even, which links them with those associated with `<p>`, so the explained unlinking process was employed. G4 is similar, but reduces the bias of the [] operator to that of all other transformations.

G5 replaces [] with a prefix equivalent operator, *pushed*. This allows the definition of what is essentially a single non-terminal symbol grammar (not counting the `<s>` symbol, as explained above). The exact biases of G4 are impossible to keep, so a compromise was chosen. Finally, G6 uses three symbols, `<p>`, `<o>` and `<v>`, to create a compact and highly readable grammar.

5.3 Experimental Setup

The experimental setup used was the same as in Table 2, but using 50% non-coding tails at initialisation [26]. Also, as seen in Sect. 4.1, different grammars will generate different solution sizes at initialisation, with direct influence in their initial fitnesses. In order to properly analyse the effect of grammar design in the search capability

Table 6 Grammars used for the Shape Match (hard) problem, versions G0–G6; a set of productions followed by $\{n\}$ means they are repeated n times, whereas the notation $\alpha | \dots | \delta$ is shorthand for a production for each of the elements in the sequence $[\alpha..\delta]$; constants in G6 are created using `GECodonValue` [24]

G0	G1 (same as G0)	G2
<s> ::= *python hard.py* <p>	<s> ::= *python hard.py* <p>	<s> ::= *python hard.py* <p>
<p> ::= <e>	<p> ::= <e>	<p> ::= <e>
| <e> <p>	| <e> <p>	| <e> <p>
<e> ::= <v>	<e> ::= <v>	<e> ::= <v>
| <o> <e>	| <o> <e>	| <o> <e>
| [<e>]	| [<e>]	| [<e>]
<o> ::= *s*0	<o> ::= *s*0	<o> ::= *s*0|*s*0|*s*0
| *s*1	| *s*1	| *s*1|*s*1|*s*1
| *s*2	| *s*2	| *s*2|*s*2|*s*2
| *s*3	| *s*3	| *s*3|*s*3|*s*3
| *gro*	| *gro*	| *gro*|*gro*|*gro*
| *shrnk*	| *shrnk*	| *shrnk*|*shrnk*|*shrnk*
<v> ::= *sqr*	<v> ::= *sqr*	<v> ::= *sqr*|*sqr*
| *crcl*	| *crcl*	| *crcl*|*crcl*
G3	G4	G5
<s> ::= *python hard.py* <p>	<s> ::= *python hard.py* <p>	<s> ::= *python hard.py* <p>
<p> ::= <e> {18}	<p> ::= <e> {42}	<p> ::= *sqr* {7}
| <e> <p> {18}	| <e> <p> {42}	| *crcl* {7}
<e> ::= *sqr* {3}	<e> ::= *sqr* {7}	| *s*0 <p> {2}
| *crcl* {3}	| *crcl* {7}	| *s*1 <p> {2}
| *s*0 <e>	| *s*0 <e> {4}	| *s*2 <p> {2}
| *s*1 <e>	| *s*1 <e> {4}	| *s*3 <p> {2}
| *s*2 <e>	| *s*2 <e> {4}	| *gro* <p> {2}
| *s*3 <e>	| *s*3 <e> {4}	| *shrnk* <p> {2}
| *gro* <e>	| *gro* <e> {4}	| *pushed* <p> {2}
| *shrnk* <e>	| *shrnk* <e> {4}	| *sqr* <p> {7}
| [<e>] {6}	| [<e>] {4}	| *crcl* <p> {7}
G6		
<s> ::= *python hard.py* <p>		
<p> ::= <v> | <o> <p> | <v> <p>		
<o> ::= *s*0 *s*1 | *s*2 | *s*3 | *gro* | *shrnk* | *pushed*		
<v> ::= *sqr* | *crcl*		

Table 7 Genome length (for RND) and min/max derivation steps (for PTC2) used during initialisation, for the three problems analysed

Problem	Parameter	g0	g1	g2	g3	g4	g5	g6
Keijzer6	Genome length	30	30	30	15	15	15	23
	Min. der. steps	3	3	3	1	1	1	2
	Max. der. steps	31	31	31	15	15	15	23
Vladislavleva4	Genome length	54	54	54	15	15	15	31
	Min. der. steps	4	4	4	1	1	1	2
	Max. der. steps	55	55	55	15	15	15	31
Shape Match (hard)	Genome length	24	24	24	14	14	14	23
	Min. der. steps	4	4	4	3	3	2	3
	Max. der. steps	25	25	25	15	15	16	24

of GE, experiments using different grammars were setup such that they generate similarly sized phenotype solutions at initialisation. As such, RND and PTC2 were setup as shown in Table 7. Finally, protected versions of some operators were used, such as division (1.0 if divisor $< 1e-5$), inversion (1.0 if argument $< 1e-5$), and square root ($\sqrt{|x|}$).

5.4 Results

Figure 6 plots the mean best train results for the K6, V4 and Shape Match (hard) problem, using RND and PTC2 initialisation.[3] All grammar variants find similarly good solutions for the K6 experiment at the 50th generation, using PTC2, but much larger differences are found between the results obtained with grammars G0–G2 versus those with grammars G3–G6, when using RND. A similar result is observed for the V4 experiment, except that in this case, the difference between G0–G2 and G3–G6 is more evident when using PTC2.

Finally, the Shape Match (hard) experiments again show a large difference between both clusters of grammars, along with the positive effect in convergence of removing the high bias towards the [] operator in G4–G6 (applying it more than once has no practical effect). Although there is a very marked change in the speed of convergence towards an optimal solution, particularly with the reduction of non-terminal symbols, eventually all grammar solutions find similarly good solutions.

5.4.1 Significance and Test Results

In order to quantify the effect of each grammar design on the search effectiveness of GE, two-sample Mann-Whitney U-tests were calculated for final median best fit results, for all grammars. The results are shown in Table 8.

These are similar to what was observed in the K6, V4 and Shape Match problems. Overall, runs using G3–G5 are significantly better than all others. Small differences between G3–G5 are mostly problem domain specific. G0–G2 are often significantly better only amongst themselves, with the exception of a few noisy real-world datasets, such as Tower and Wine Quality. G6 seems to provide a compromise between performance and complexity of grammar. As before, G0 does particularly bad with RND initialisation, and differences between different grammars are less evident when using PTC2.

Regarding test performance, no validation or early-stopping approaches were employed in these experiments, so it is not unreasonable to expect some level of overfitting, particularly from approaches with very good training performance. The statistical significance results are shown in Table 9.

As expected, the test results are far less clear cut. Results are still better when using G3–G5, but are very problem-dependent. This mostly results from problems such as K12: it is such a hard problem to solve using the original function and terminal set [14] (which uses no trigonometric functions), that any small improvement in training performance invariably led to a degradation in test performance.

[3] As the focus of this study is on grammar design, no regression performance improving techniques such as linear scaling [12] or cross-validation were used.

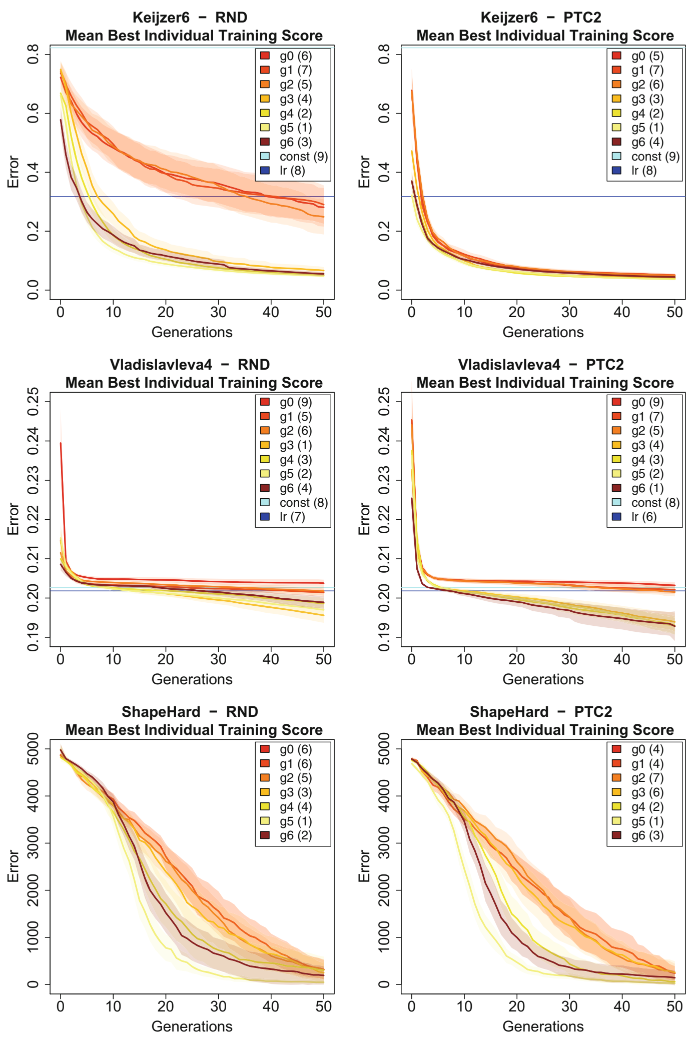

Fig. 6 Mean best train scores for the K6 (top), V4 (middle) and Shape match (hard) (bottom) experiments, using RND (left) or PTC2 (right), averaged across 100 independent runs; error is measured as the Root Mean Squared Error (RMSE) for K6 and V4, or the number of mismatched pixels for shape matching. Shades indicate 95% confidence intervals about the mean. Cyan and blue lines (where visible) indicate the RMSE of a constant predictor (mean of the train response variable) and a linear regression model of the training set, respectively

Table 8 Mann-Whitney U-tests of final median best training fit results, for all problems attempted; each number is a count of all grammars against which a significantly differently better performance was measured (results across 100 independent runs)

		G0	G1	G2	G3	G4	G5	G6
RND	BreastCancerW	0	0	0	5	3	0	0
	Dow	0	0	0	3	3	3	3
	EvenParity5	0	0	0	4	4	4	0
	Forest	0	1	1	4	4	5	1
	Housing	0	1	1	3	3	3	3
	Keijzer6	0	0	0	3	4	4	3
	Korns12	0	0	0	3	4	6	4
	Multiplexer11	0	1	1	3	3	1	4
	Pagie1	0	0	0	4	4	4	0
	Tower	0	2	2	4	5	2	1
	Vladislavleva4	0	0	0	4	0	4	3
	WineQualityRed	0	1	1	5	5	2	2
	WineQualityWhite	0	1	1	3	3	3	3
	ShapeEasy	0	0	0	0	0	0	0
	ShapeMedium	0	0	0	0	0	0	0
	ShapeHard	0	0	0	0	0	1	0
	Sum	0	7	7	48	45	42	27
	Mean	0	0.438	0.438	3	2.812	2.625	1.688
	Median	0	0	0	3	3	3	1.5
	Std. dev.	0	0.6292	0.6292	1.633	1.797	1.857	1.580
PTC2	BreastCancerW	0	0	0	4	4	1	0
	Dow	0	0	0	2	0	2	3
	EvenParity5	0	0	0	1	1	4	1
	Forest	0	0	0	2	4	1	0
	Housing	0	0	0	3	2	2	2
	Keijzer6	0	0	0	3	4	4	0
	Korns12	0	0	0	3	4	5	6
	Multiplexer11	0	1	0	0	0	0	4
	Pagie1	0	0	0	4	4	4	2
	Tower	0	0	1	1	3	1	1
	Vladislavleva4	0	0	0	3	3	3	3
	WineQualityRed	1	1	0	2	2	0	0
	WineQualityWhite	1	1	3	2	1	0	1
	ShapeEasy	0	0	0	0	0	0	0
	ShapeMedium	0	0	0	0	0	0	0
	ShapeHard	0	0	0	0	0	0	0
	Sum	2	3	4	30	32	27	23
	Mean	0.125	0.188	0.25	1.875	2	1.688	1.438
	Median	0	0	0	2	2	1	1
	Std. dev.	0.342	0.403	0.774	1.408	1.713	1.778	1.788

Table 9 Mann-Whitney U-tests of final median best test fit results (of best training solutions), for all problems with a test set; each number is a count of all grammars against which a significantly differently better performance was measured (results across 100 independent runs)

		G0	G1	G2	G3	G4	G5	G6
RND	BreastCancerW	0	0	0	0	0	1	0
	Dow	0	0	0	4	2	2	0
	Forest	6	2	1	0	0	0	3
	Housing	0	1	1	3	3	3	3
	Keijzer6	0	0	0	2	4	4	3
	Korns12	3	3	6	3	0	0	1
	Pagie1	0	0	0	4	4	4	0
	Tower	1	0	0	0	0	0	1
	Vladislavleva4	0	1	1	0	0	0	1
	WineQualityRed	0	1	1	5	5	2	2
	WineQualityWhite	5	0	0	0	0	0	0
	Sum	15	8	10	21	18	16	14
	Mean	1.364	0.727	0.909	1.909	1.636	1.455	1.273
	Median	0	0	0	2	0	1	1
	Std. dev.	2.248	1.009	1.758	1.973	2.014	1.635	1.272
PTC2	BreastCancerW	0	0	0	0	4	4	0
	Dow	0	0	0	1	2	0	0
	Forest	1	0	0	0	0	0	1
	Housing	0	0	0	3	2	2	2
	Keijzer6	1	0	0	1	2	2	0
	Korns12	4	3	3	3	2	0	0
	Pagie1	0	0	0	4	3	4	2
	Tower	1	0	0	0	0	0	1
	Vladislavleva4	1	1	0	0	0	0	1
	WineQualityRed	0	1	0	2	1	0	0
	WineQualityWhite	0	0	0	0	0	0	0
	Sum	8	5	3	14	16	12	7
	Mean	0.727	0.455	0.273	1.273	1.455	1.091	0.636
	Median	0	0	0	1	2	0	0
	Std. dev.	1.191	0.934	0.905	1.489	1.368	1.640	0.809

5.5 Analysis

The results obtained show just how much the design of a grammar can affect the search capability of GE. The most obvious performance improvements come from the construction of balanced grammars, and the reduction of the number of non-terminal symbols. These come at a price, however, particularly the latter: the complexity of the generated grammars can make them particularly hard to read and/or modify by hand. However, the application of these transformations is a deterministic process, meaning that they can be applied automatically: this allows manual grammar modifications to be applied to the simpler versions of the grammars, with subsequent application of automatic transformations. In any case, even a partial (manual) application of some of these modifications can be of use, as seen with the results for G6 variants.

The usefulness of some of these applications is problem-dependent. This is particularly the case of bias-related transformations, expressed in the results obtained with G2 and G4. For the problems attempted, the unlinking process employed with G2 grammars does not seem to confer any performance advantage when compared to G1 grammars. Likewise, the removal of biases towards certain operators employed in G4 grammars does not seem to confer any advantage, in the problems attempted. Finally, the switch from infix to prefix notation only makes a small difference for a few problems (for better or worse).

It is worth pointing out again that these results relate only to GE using a linear genome representation. Derivation tree based approaches make use of different search operators, and as such, the effect of grammar design is markedly different.

Another observation is the need to understand grammar design, when setting up any initialisation procedure (even something as simple as RND initialisation). The size of genotype structures required to generate certain solution (phenotype) size ranges needs to be adapted, depending on the grammar used.

6 Conclusions

Grammar design is one of the main tasks in GE: in order to write a good grammar, deep knowledge in both GE and the application domain are required. The experiments examined in this chapter, however, show that this is not necessarily always the case: there are specific grammar design principles that can be applied when attempting to solve any problem using GE, which do not require domain knowledge. Of the transformations analysed, the creation of recursion-balanced grammars and the reduction of the number of non-terminal symbols are particularly useful for improving the performance of GE, when using a linear genome representation.

Fixing symbol biases does not necessarily lead to better performance: this is completely problem dependent, and it can both degrade or improve performance. But it does affect search space exploration, as shown in the experiments conducted. This leads to two recommendations:

– When designing a GE grammar to compare its performance with systems such as GP, respect the symbol biases of the original system;
– When applying GE to a real-world problem, where symbol biases are known, use this knowledge to bias the exploration of the search space.

GE is like any other search algorithm, and in fact like any tool: it has its advantages and disadvantages, and will only provide its best performance if correctly used. There has been a recent surge of publications criticising GE's performance [17, 18, 21, 42], some even deeming that its performance "resembles that of random search" [42]. But most of the results provided were the result of using badly designed grammars, and poor experimental setup. The analysis and results presented in this chapter aim to provide another step towards achieving the goal of *Understanding Grammatical Evolution*.

References

1. J. Byrne, M. O'Neill, J. McDermott, A. Brabazon, An analysis of the behaviour of mutation in grammatical evolution, in *European Conference on Genetic Programming, EuroGP 2010*, ed. by A.I. Esparcia-Alcázar et al. Lecture Notes in Computer Science, vol. 6021 (Springer, Berlin, 2010), pp. 14–25
2. P. Cortez, A. Morais, A data mining approach to predict forest fires using meteorological data, in *Portuguese Conference on Artificial Intelligence, EPIA 2007*, ed. by J. Neves et al. (APPIA, Lisboa, 2007), pp. 512–523
3. P. Cortez, A. Cerdeira, F. Almeida, T. Matos, J. Reis, Modeling wine preferences by data mining from physicochemical properties. Decis. Support. Syst. **47**(4), 547–553 (2009)
4. D. Fagan, M. O'Neill, E. Galván-López, A. Brabazon, S. McGarraghy, An analysis of genotype-phenotype maps in grammatical evolution, in *European Conference on Genetic Programming, EuroGP 2010*, ed. by A.I. Esparcia-Alcázar et al. Lecture Notes in Computer Science, vol. 6021 (Springer, Berlin, 2010), pp. 62–73
5. D. Fagan, M. Nicolau, E. Hemberg, M. O'Neill, A. Brabazon, S. McGarraghy, Investigation of the performance of different mapping orders for GE on the max problem, in *European Conference on Genetic Programming, EuroGP 2011, Torino, Italy, April 27–29, 2011, Proceedings*, ed. by S. Silva et al. Lecture Notes in Computer Science, vol. 6621 (Springer, Berlin, 2011), pp. 286–297
6. S. Forstenlechner, M. Nicolau, D. Fagan, M. O'Neill, Grammar design for derivation tree based genetic programming systems, in *European Conference on Genetic Programming, EuroGP 2016*, ed. by M. Heywood et al. Lecture Notes in Computer Science, vol. 9594 (Springer, Cham, 2016), pp. 199–214
7. S. Forstenlechner, D. Fagan, M. Nicolau, M. O'Neill, A grammar design pattern for arbitrary program synthesis problems in genetic programming, in *European Conference on Genetic Programming, EuroGP 2017*, ed. by J. McDermott et al. Lecture Notes in Computer Science, vol. 10196 (Springer, Berlin, 2017), pp. 262–277
8. C. Gathercole, P. Ross, An adverse interaction between crossover and restricted tree depth in genetic programming, in *Genetic Programming 1996: First Annual Conference*, ed. by J.R. Koza et al. (MIT Press, Cambridge, 1996), pp. 291–296
9. R. Harper, GE, explosive grammars and the lasting legacy of bad initialisation, in *IEEE Congress on Evolutionary Computation, CEC 2010*, 2010, pp. 2602–2609
10. E. Hemberg, An exploration of grammars in grammatical evolution. Ph.D. Thesis, University College Dublin, University College Dublin, Ireland, 2010
11. E. Hemberg, N. McPhee, M. O'Neill, A. Brabazon, Pre-, in- and postfix grammars for symbolic regression in grammatical evolution, in *IEEE Workshop and Summer School on Evolutionary Computing, 2008*, 2008, pp. 18–22
12. M. Keijzer, Improving symbolic regression with interval arithmetic and linear scaling, in *European Conference on Genetic Programming, EuroGP 2003*, ed. by C. Ryan et al. Lecture Notes in Computer Science, vol. 2610 (Springer, Berlin, 2003), pp. 70–82
13. M. Keijzer, M. O'Neill, C. Ryan, M. Cattolico, Grammatical evolution rules: the mod and the bucket rule, in *European Conference on Genetic Programming, EuroGP 2002*, ed. by J.A. Foster et al. Lecture Notes in Computer Science, vol. 2278 (Springer, Berlin, 2002), pp. 123–130

14. M.F. Korns, Accuracy in symbolic regression, in *Genetic Programming Theory and Practice IX*, ed. by R. Riolo et al. Genetic and Evolutionary Computation (Springer, New York, 2011), pp. 129–151
15. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (MIT Press, Cambridge, 1992)
16. M. Lichman, UCI machine learning repository (2013), http://archive.ics.uci.edu/ml
17. N. Lourenço, J. Ferrer, F.B. Pereira, E. Costa, A comparative study of different grammar-based genetic programming approaches, in *European Conference on Genetic Programming, EuroGP 2017*, ed. by J. McDermott et al. Lecture Notes in Computer Science, vol. 10196 (Springer, Cham, 2017), pp. 311–325
18. N. Lourenço, F. B. Pereira, E. Costa, Unveiling the properties of structured grammatical evolution. Genet. Program. Evolvable Mach. **17**(3), 251–289 (2017)
19. S. Luke, Two fast tree-creation algorithms for genetic programming. IEEE Trans. Evol. Comput. **4**(3), 274–283 (2000)
20. S. Luke, L. Panait, A comparison of bloat control methods for genetic programming. Evol. Comput. **14**(3), 309–344 (2006)
21. E. Medvet, A comparative analysis of dynamic locality and redundancy in grammatical evolution, in *European Conference on Genetic Programming, EuroGP 2017*, ed. by J. McDermott et al. *Lecture Notes in Computer Science*, vol. 10196 (Springer, Cham, 2017), pp. 326–342
22. M. Nicolau, Automatic grammar complexity reduction in grammatical evolution, in *Genetic and Evolutionary Computation Conference, GECCO 2004*, ed. by R. Poli et al. (2004)
23. M. Nicolau, Understanding grammatical evolution: initialisation. Genet. Program. Evolvable Mach. **18**(4), 1–41 (2017)
24. M. Nicolau, I. Dempsey, Introducing grammar based extensions for grammatical evolution, in *IEEE Congress on Evolutionary Computation, CEC 2006* (2006), pp. 2663–2670
25. M. Nicolau, M. Fenton, Managing repetition in grammar-based genetic programming, in *Genetic and Evolutionary Computation Conference - GECCO 2016, Denver, CO, USA, July 20–24, 2016, Proceedings*, ed. by T. Friedrich (ACM, New York, 2016), pp. 765–772
26. M. Nicolau, M. O'Neill, A. Brabazon, Termination in grammatical evolution: grammar design, wrapping, and tails, in *IEEE Congress on Evolutionary Computation, CEC 2012* (2012), pp. 1–8
27. M. Nicolau, A. Agapitos, M. O'Neill, A. Brabazon, Guidelines for defining benchmark problems in genetic programming, in *IEEE Congress on Evolutionary Computation, CEC 2015* (2015)
28. M. O'Neill, A. Brabazon, mGGA: the meta-grammar genetic algorithm, in *European Conference on Genetic Programming, EuroGP 2005*, ed. by M. Keijzer et al. Lecture Notes in Computer Science, vol. 3447 (Springer, Berlin, 2005), pp. 311–320
29. M. O'Neill, C. Ryan, Evolving multi-line compilable c programs, in *European Workshop on Genetic Programming, EuroGP 99*, ed. by R. Poli et al. Lecture Notes in Computer Science, vol. 1598 (Springer, Berlin, 1999), pp. 83–92
30. M. O'Neill, C. Ryan, *Grammatical Evolution - Evolutionary Automatic Programming in an Arbitrary Language*. Genetic Programming, vol. 4 (Kluwer Academic, Dordrecht, 2003)
31. M. O'Neill, C. Ryan, M. Nicolau, Grammar defined introns: an investigation into grammars, introns, and bias in grammatical evolution, in *Genetic and Evolutionary Computation Conference, GECCO 2001*, ed. by L. Spector et al. (Morgan Kaufmann, Burlington, 2001), pp. 97–103
32. M. O'Neill, C. Ryan, M. Keijzer, M. Cattolico, Crossover in grammatical evolution. Genet. Program. Evolvable Mach. **4**(1), 67–93 (2003)
33. M. O'Neill, J.M. Swafford, J. McDermott, J. Byrne, A. Brabazon, E. Shotton, C. McNally, M. Hemberg, Shape grammars and grammatical evolution for evolutionary design, ed. by G. Raidl et al. *Genetic and Evolutionary Computation Conference, GECCO 2009* (ACM, New York, 2009), pp. 1035–1042
34. L. Pagie, P. Hogeweg, Evolutionary consequences of coevolving targets. Evol. Comput. **5**(4), 401–418 (1997)

35. D. Robilliard, S. Mahler, D. Verhaghe, C. Fonlupt, Santa fe trail hazards, in *International Conference on Evolution Artificielle, EA 2005*, ed. by E.-G. Talbi et al. Lecture Notes in Computer Science, vol. 3871 (Springer, Berlin, 2005), pp. 1–12
36. F. Rothlauf, M. Oetzel, On the locality of grammatical evolution, in *European Conference on Genetic Programming, EuroGP 2006*, ed. by P. Collet et al. Lecture Notes in Computer Science, vol. 3905 (Springer, Berlin, 2006), pp. 320–330
37. C. Ryan, A. Azad, Sensible initialisation in grammatical evolution, in *Genetic and Evolutionary Computation Conference, GECCO 2003*, ed. by E. Cantú-Paz et al. (AAAI, Menlo Park, 2003)
38. C. Ryan, J. Collins, M. O'Neill, Grammatical evolution: evolving programs for an arbitrary language, in *European Workshop on Genetic Programming, EuroGP 1998*, ed. by W. Banzhaf et al. Lecture Notes in Computer Science, vol. 1391 (Springer, Berlin, 1998), pp. 83–95
39. C. Ryan, M. Keijzer, M. Nicolau, On the avoidance of fruitless wraps in grammatical evolution, in *Genetic and Evolutionary Computation Conference, GECCO 2003*, ed. by E. Cantú-Paz et al. Lecture Notes in Computer Science, vol. 2724 (Springer, Berlin, 2003), pp. 1752–1763
40. J. Tavares, F. B. Pereira, Automatic design of ant algorithms with grammatical evolution, in *European Conference on Genetic Programming, EuroGP 2012*, ed. by A. Moraglio et al. Lecture Notes in Computer Science, vol. 7244 (Springer, Berlin, 2012), pp. 206–217
41. E.J. Vladislavleva, G.F. Smits, D. den Hertog, Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. IEEE Trans. Evol. Comput. **13**(2), 333–349 (2009)
42. P.A. Whigham, G. Dick, J. Maclaurin, C.A. Owen, Examining the "best of both worlds" of grammatical evolution, in *Genetic and Evolutionary Computation Conference, GECCO 2015*, ed. by S. Silva (ACM, New York, 2015) pp. 1111–1118
43. D.R. White, J. McDermott, M. Castelli, L. Manzoni, B.W. Goldman, G. Kronberger, W. Jaśkowski, U.-M. O'Reilly, S. Luke, Better GP benchmarks: community survey results and proposals. Genet. Program. Evolvable Mach. **14**(1), 3–29 (2013)