Grammatical Evolution: A Tutorial using **gramEvol**

Farzad Noorian, Anthony M. de Silva, Philip H.W. Leong

July 18, 2020

1 Introduction

Grammatical evolution (GE) is an evolutionary search algorithm, similar to genetic programming (GP). It is typically used to generate programs with syntax defined through a grammar. The original author's website [1] is a good resource for a formal introduction to this technique:

- `http://www.grammatical-evolution.org/`

This document serves as a quick and informal tutorial on GE, with examples implemented using the **gramEvol** package in R.

2 Grammatical Evolution

The goal of using GE is to automatically generate a program that minimises a cost function:

1. A *grammar* is defined to describe the syntax of the programs.
2. A *cost function* is defined to assess the quality (the *cost* or *fitness*) of a program.
3. An *evolutionary algorithm*, such as GA, is used to search within the space of all programs definable by the grammar, in order to find the program with the lowest cost.

Notice that by a *program*, we refer to any sequence of instructions that perform a specific task. This ranges from a single expression (e.g., `sin(x)`), to several statements with function declarations, assignments, and control flow.

The rest of this section will describe each component in more details.

2.1 Grammar

A grammar is a set of rules that describe the syntax of sentences and expressions in a language. While grammars were originally invented for studying natural languages, they are extensively used in computer science for describing programming languages.

2.1.1 Informal introduction to context-free grammars

GE uses a *context-free grammar* to describe the syntax of programs.

A grammar in which the rules are not sensitive to the sentence's context is called a *context-free grammar* (CFG), and is defined using a collection of *terminal* symbols, *non-terminal* symbols, *production rules*, and a *start* symbol [2]:

- Terminal symbols are the lexicon of the language.

- Non-terminal symbols are used to describe the class of words in the language, or *variables* that can take different values. For example, a <*subject*>, a <*verb*>, or an <*object*>.
- A production rule defines what symbols replace a non-terminal. For example, each of the four following lines is a production rule:

  $\langle sentence \rangle$ ::= <*subject*> <*verb*> <*object*>. | <*subject*> <*verb*>. (1.a), (1.b)

  $\langle subject \rangle$ ::= I | You | They (2.a), (2.b), (2.c)

  $\langle verb \rangle$ ::= read | write | check (3.a), (3.b), (3.c)

  $\langle object \rangle$ ::= books | stories | academic papers (4.a), (4.b), (4.c)

  In each rule, the "—" symbol separates different replacement possibilities; such as <*subject*>, that can be replaced with "I", "You" or "They". One must note that a non-terminal symbol can be replaced with ther non-terminals as well as terminal symbols, such as in the example's <*sentence*>.

  This style of notation, including the use of angle brackets (¡ and ¿) is known as the *Backus–Naur Form* (BNF).
- A start symbol determines a non-terminal where the generation of the expression starts. For example:
  - Start: <*sentence*>

Informally, only the start symbol and the production rules are required to define a grammar.

2.1.2 Formal definition of a context-free grammar

In formal language theory, a context-free grammar is a *formal grammar* where every production rule, formalized by the pair (n, V), is in form of $n \to V$. The CFG is defined by the 4-tuple $(\mathcal{T}, \mathcal{N}, \mathcal{R}, \mathcal{S})$, where $\mathcal{T}$ is the finite set of terminal symbols, $\mathcal{N}$ is the finite set of non-terminal symbols, $\mathcal{R}$ is the production rule set, $\mathcal{S} \in \mathcal{N}$ is the start symbol.

A production rule $n \to V$ is realized by replacing the non-terminal symbol $n \in \mathcal{N}$ with the symbol $v \in V$, where $V \in (\mathcal{T} \cup \mathcal{N})^*$ is a sequence of terminal and/or non-terminal symbols.

For more details on CFGs, their relation to context-free languages, parsing, compilers and other related topics refer to [2] or Wikipedia:

- `https://en.wikipedia.org/wiki/Context-free_grammar`

2.1.3 From grammar to an expression

Notice that each rule in the grammar of Section 2.1.1 is numbered. Using these numbers, one can precisely refer to a certain expression. This is performed by replacing the first non-terminal symbol with the nth rule of that non-terminal, starting with the start symbol.

For example, the sequence [2, 3, 1] selects rules (1.b), (2.c) and (3.a) in the following four-step sequence:

Step	Sequence	Rule	Current state
0		Start	<*sentence*>.
1	2	(1.b)	<*subject*> <*verb*>.
2	3	(2.c)	They <*verb*>.
3	1	(3.a)	They read.

2.2 Evolutionary optimisation

Evolutionary optimisation algorithms are a class of optimisation techniques inspired by natural evolution. They are used in cases where:

- The solution to the problem can be represented by a certain structure. For example, the solution is an array of binary variables, or integer numbers.
  - Typically the array size is fixed and each unique value arrangement is considered a candidate solution.
  - Using biological terminology, this structure is referred to as the *chromosome* or *genotype*.
- There exist a cost function which can quickly return the *cost* or *fitness* of any candidate solution.
- Solving the problem using gradient descent techniques is hard or impossible, because the cost function is non-smooth, or has multiple local optimas, or is simply discrete, such as the travelling salesman problem (or in hindsight, a program generated by grammars).

It most be noted that the stochastic nature of evolutionary algorithms does not guarantee the optimal solution, since most practical problems involve very large search spaces, and it is often not computationally feasible to search the whole space.

The oldest and simplest of these algorithms is the genetic algorithm (GA), which optimises a vector of binary variables. In this vignette, when referring to GA, we refer to an extended GA which handles integers numbers.

For an in depth introduction, readers are referred to Wikipedia:

- `https://en.wikipedia.org/wiki/Evolutionary_algorithm`

2.2.1 Optimising a program by evolution

GA only optimises numeric arrays. By *mapping* an integer array to a program using a grammar, GA can be readily applied to evolve programs:

1. The solution is represented by an array of integers.
2. The array is mapped to a program through the grammar using the technique explained is Section 2.1.3.
   - Using biological terminology, the program is called a *phenotype*, and the mapping is referred to as *genotype to phenotype mapping*.
3. The cost function measures the fitness of the program.
4. Any evolutionary optimisation technique is applied on the integer array.

2.3 Applications of grammatical evolution

Any application which needs a program definable by grammar, is creatable in GE. Using a grammar allows integration of domain knowledge and a custom program syntax, which adds flexibility and precision to GE compared to other techniques such as GP.

Applications of GE include computational finance, music, and robotic control, among others. See `http://www.grammatical-evolution.org/pubs.html` for a collection of publications in this area.

3 gramEvol Package

The package **gramEvol** simplifies defining a grammar and offers a GA implementation. **gramEvol** hides many details, including the grammar mapping and GA parameters, and the only things the user has to do is to:

1. Define a grammar using `CreateGrammar`.
2. Define a cost function. It should accept one (or more) R `expression`(s) and return a numeric value.

3. Call `GrammaticalEvolution`.

In this section, examples are used to demonstrate its usage.

3.1 Rediscovery of Kepler's law by symbolic regression

Symbolic regression is the process of discovering a function, in symbolic form, which fits a given set of data. Evolutionary algorithms such as GP and GE are commonly used to solve Symbolic Regression problems. For more information, visit `https://en.wikipedia.org/wiki/Symbolic_regression` or `http://www.symbolicregression.com/`.

Rediscovery of Kepler's law has been used as a benchmark for symbolic regression [3, 4, 5]. Here, the goal is to find a relationship between orbital periods and distances of solar system planets from the sun. The distance and period data, normalised to Earth, is shown in Table 1.

Planet	Distance	Period
Venus	0.72	0.61
Earth	1.00	1.00
Mars	1.52	1.84
Jupiter	5.20	11.90
Saturn	9.53	29.40
Uranus	19.10	83.50

Table 1: Orbit period and distance from the sun for planets in solar system.

Kepler's third law states:

$$period^2 = constant \times distance^3 \tag{1}$$

3.1.1 Defining a grammar

To use grammatical evolution to find this relationship from the data, we define a grammar as illustrated in Table 2. Here $\mathcal{S}$ denotes the starting symbol and $\mathcal{R}$ is the collection of production rules.

$\mathcal{S} = <expr>$

Production rules : $\mathcal{R}$

$\langle expr \rangle$	::= $\langle expr \rangle \langle op \rangle \langle expr \rangle$ | $\langle sub\text{-}expr \rangle$	(1.a), (1.b)
$\langle sub\text{-}expr \rangle$	::= $\langle func \rangle (\langle var \rangle)$ | $\langle var \rangle$ | $\langle var \rangle \hat{} \langle n \rangle$	(2.a), (2.b), (2.c)
$\langle func \rangle$	::= log | sqrt | sin | cos	(3.a), (3.b), (3.c), (3.d)
$\langle op \rangle$	::= + | - | ×	(4.a), (4.b), (4.c)
$\langle var \rangle$	::= `distance` | `distance`$\hat{}\langle n \rangle$ | $\langle n \rangle$	(5.a), (5.b), (5.c)
$\langle n \rangle$	::= `1` | `2` | `3` | `4`	(6.a), (6.b), (6.c), (6.d)

Table 2: Grammar for discovering Kepler's equation.

This is a general purpose grammar, and it can create different expressions corresponding to different formulas which can explain and model the data.

The first step for using **gramEvol** is loading the grammar:

```
library("gramEvol")

ruleDef <- list(expr  = grule(op(expr, expr), func(expr), var),
                func  = grule(sin, cos, log, sqrt),
                op    = grule(`+`, `-`, `*`),
                var   = grule(distance, distance^n, n),
                n     = grule(1, 2, 3, 4))

grammarDef <- CreateGrammar(ruleDef)
```

Here, the BNF notation is implemented in R:

- Rules are defined as a `list`.
- Each rule is defined using `non.terminal.name = grule(replacement1, replacement2, ...)` format.
- `CreateGrammar` is used to load the list and create the grammar object.

The print function reproduces the grammar in a format similar to Table 2:

```
print(grammarDef)

## <expr> ::= <op>(<expr>, <expr>) | <func>(<expr>) | <var>
## <func> ::= `sin` | `cos` | `log` | `sqrt`
## <op>   ::= `+` | `-` | `*`
## <var>  ::= distance | distance^<n> | <n>
## <n>    ::= 1 | 2 | 3 | 4
```

Note that `'+'` and `op(expr, expr)` are used in the code above because `grule` expects R expressions, and `expr op expr` is not valid in R. As it is tedious to convert between the functional form and the operator form, the package also provides `gsrule` (or grammar string rule), which accepts strings with `<>`:

```
ruleDef <- list(expr  = gsrule("<expr><op><expr>", "<func>(<expr>)", "<var>"),
                func  = gsrule("sin", "cos", "log", "sqrt"),
                op    = gsrule("+", "-", "*"),
                var   = grule(distance, distance^n, n),
                n     = grule(1, 2, 3, 4))

CreateGrammar(ruleDef)

## <expr> ::= <expr><op><expr> | <func>(<expr>) | <var>
## <func> ::= sin | cos | log | sqrt
## <op>   ::= + | - | *
## <var>  ::= distance | distance^<n> | <n>
## <n>    ::= 1 | 2 | 3 | 4
```

Note that `gsrule` and `grule` can be mixed, as in the example above.

3.1.2 Defining a cost function

We use the following equation to normalise the error, adjusting its impact on small values (e.g., Venus) versus large values (e.g., Uranus):

$$e = \frac{1}{N} \sum log(1 + |p - \hat{p}|) \tag{2}$$

where e is the normalised error, N is the number of samples, p is the orbital period and $\hat{p}$ is the result of symbolical regression. We implement this as the fitness function `SymRegFitFunc`:

```
planets <- c("Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus")
distance <- c(0.72, 1.00, 1.52, 5.20, 9.53, 19.10)
period <- c(0.61, 1.00, 1.84, 11.90, 29.40, 83.50)

SymRegFitFunc <- function(expr) {
  result <- eval(expr)

  if (any(is.nan(result)))
    return(Inf)

  return (mean(log(1 + abs(period - result))))
}
```

Here, the `SymRegFitFunc` receives an R `expression` and evaluates it. It is assumed that the `expression` uses `distance` to estimate the `period`. Invalid expressions are handled by returning a very high cost (infinite error). Valid results are compared with the actual period according to (2) to compute the expression's fitness.

3.1.3 Evolving the grammar

`GrammaticalEvolution` can now be run. All of the parameters are determined automatically. To avoid wasting time, and as the best possible outcome and its error are known (because we know the answer), a `terminationCost` is computed and set to terminate GE when the Kepler's equation is found.

```
ge <- GrammaticalEvolution(grammarDef, SymRegFitFunc,
                           terminationCost = 0.021)
ge

## Grammatical Evolution Search Results:
##   No. Generations:  11
##   Best Expression:  sqrt(distance^3)
##   Best Cost:        0.0201895728693592
```

Now that the result is found, it can be used in production. Here we only use it in a simple comparison:

```
best.expression <- ge$best$expression

data.frame(distance, period, Kepler = sqrt(distance^3),
           GE = eval(best.expression))

##   distance period     Kepler         GE
## 1     0.72   0.61  0.6109403  0.6109403
## 2     1.00   1.00  1.0000000  1.0000000
## 3     1.52   1.84  1.8739819  1.8739819
## 4     5.20  11.90 11.8578244 11.8578244
## 5     9.53  29.40 29.4197753 29.4197753
## 6    19.10  83.50 83.4737743 83.4737743
```

3.1.4 Monitoring evolution

As a real-world optimisation may take a long time, a feedback of the state of optimisation is desirable. `GrammaticalEvolution` allows monitoring this status using a callback function. This function, if provided to the parameter `monitorFunc`, receives an object similar to the return value of `GrammaticalEvolution`. For example, the following function prints the current generation, the best individual's expression and its error:

```
customMonitorFunc <- function(results){
  cat("--------------------\n")
  print(results)
}

ge <- GrammaticalEvolution(grammarDef, SymRegFitFunc,
                           terminationCost = 0.021,
                           monitorFunc = customMonitorFunc)
```

or even using the `print` function directly:

```
ge <- GrammaticalEvolution(grammarDef, SymRegFitFunc,
                           terminationCost = 0.021,
                           monitorFunc = print)
```

which prints:

```
Grammatical Evolution Search Results:
No. Generations:  1
Best Expression:  distance
Best Cost:        1.60700784338907
Grammatical Evolution Search Results:
No. Generations:  2
Best Expression:  distance
Best Cost:        1.60700784338907
```

. . . until:

```
Grammatical Evolution Search Results:
No. Generations:  9
Best Expression:  distance + distance
Best Cost:        1.54428158317392
Grammatical Evolution Search Results:
No. Generations:  10
Best Expression:  1 - distance + (cos(distance) - 1) * sin(distance^2) + distance + (log(distance) + di
Best Cost:        1.4186428597461
Grammatical Evolution Search Results:
No. Generations:  11
Best Expression:  sqrt(distance^3)
Best Cost:        0.0201895728693592
```

3.2 Discovering Regular Expressions

A regular expressions (RE) is a string that determines a character pattern. REs are more expressive and precise in determining sub-string matches compared to wildcards, and are widely used in many string pattern

matching tasks, such as searching through log files or parsing a program's output. See the Wikipedia entry at `https://en.wikipedia.org/wiki/Regular_expression` for an in-depth introduction to REs.

Creating a regular expressions requires careful assembly of symbols and operators to match the desired pattern. While this is usually performed by an expert programmer, it is possible to use evolutionary optimisation techniques to infer a RE from examples [6].

In this example, we demonstrate how **gramEvol** can be used to learn REs.

3.2.1 Regular expression in R

In formal language theory, a regular expression is a sequence of symbols and operators that describes a character pattern. REs are translated by RE processors into a non-deterministic finite automaton (NFA) and subsequently into a deterministic finite automaton (DFA). The DFA can then be executed on any character string to recognize sub-strings that match the regular expression. For a theoretical introduction to REs, including their relationship with context-free grammars, readers are referred to [2].

R supports standard regular expression with both the POSIX and the Perl syntax. In addition, the **rex** Package [7] offers a functional interface for creating REs in R.

3.2.2 Matching a decimal real number

Consider matching a decimal real number in the form of $[\pm]nnn[.nnn]$, where [ ] means optional and nnn denotes one or more digits. The following table compares this notation with the syntax of Perl, POSIX, and **rex**:

	Notation	Perl	POSIX	**rex**
One digit	n	\d	[[:digit:]]	`number`
One or more digits	nnn	\d+	[[:digit:]]+	`numbers`
Optional presence of X	[X]	X?	X?	`maybe(X)`
alternate presence of X or Y	X—Y	X—Y	X—Y	`or(X, Y)`
Plus sign	+	\+	\+	`"+"`
Minus sign	-	-	-	`"-"`
Dot	.	\.	\.	`"."`

Using the above table, $[\pm]nnn[.nnn]$ is translated to:

- Perl: `(\+|-)?\d+(\.\d+)?`
- POSIX: `(\+|-)?[[:digit:]]+(\.[[:digit:]]+)?`
- **rex**: `maybe(or("+", "-")), numbers, maybe(".", numbers)`

To use a RE, the expression has to be wrapped in a start and stop symbol (`^...$` in POSIX and Perl, and `rex(start, ..., end)` for **rex**):

```
re <- "^(\\+|-)?[[:digit:]]+(\\.[[:digit:]]+)?$"
```

`grepl` can be used to check if a string matches the RE pattern or not:

```
grepl(re, "+1.1")

## [1] TRUE

grepl(re, "1+1")

## [1] FALSE
```

Some matching and non-matching examples are listed below:

```
matching <- c("1", "11.1", "1.11", "+11", "-11", "-11.1")
non.matching <- c("a", "1.", "1..1", "-.1", "-", "1-", "1.-1",
                  ".-1", "1.-", "1.1.1", "", ".", "1.1-", "11-11")
```

3.2.3 Inferring a regular expression

In this section, we use **gramEvol** to learn a RE that matches a decimal real number, as explained in the previous section.

Defining a cost function: The objective is to infer a RE that matches the decimal numbers in the vector `matching`, but not in the `non.matching`. Consequently, the score of any RE is determined by counting the number of matches and non-matches:

```
re.score <- function(re) {
  score <- sum(sapply(matching, function(x) grepl(re, x))) +
           sum(sapply(non.matching, function(x)  !grepl(re, x)))
  return (length(matching) + length(non.matching) - score)
}
```

The fitness function in **gramEvol** receives an R `expression`, which has to be evaluated before being passed to `re.score`:

```
fitfunc <- function(expr) re.score(eval(expr))
```

Defining a grammar: We use **rex** RE functions to create a grammar. The grammar only includes the functions explored in Section 3.2.1, and is designed such that the search space is reduced:

```
library("rex")
library("gramEvol")
grammarDef <- CreateGrammar(list(
  re    = grule(rex(start, rules, end)),
  rules = grule(rule, .(rule, rules)),
  rule  = grule(numbers, ".", or("+", "-"), maybe(rules))))
grammarDef

## <re>    ::= rex(start, <rules>, end)
## <rules> ::= <rule> | <rule>, <rules>
## <rule>  ::= numbers | "." | or("+", "-") | maybe(<rules>)
```

- The first rule, <*re*>, creates a valid **rex** command that uses <*rules*> for pattern matching.
- The second element, <*rules*>, is *recursive* and can create a collection of rules by repeating itself, e.g., <*rule*>, <*rule*>, <*rule*>. The `.()` allows using a comma inside a `grule` definition, where otherwise it would have been interpreted as another replacement rule in the list.
- The last element, <*rule*>, expands to a RE function or character pattern. These include `numbers` and `maybe` from **rex**, a decimal point, and + or −.

Evolving the grammar: The last step is to perform a search for a regular expression that minimises the score function. Here the minimum `terminationCost` is known (i.e., zero error), and `max.depth` is increased to allow for more expansion of the recursive <*rules*>. We use `GrammaticalExhaustiveSearch` to exhaustively search for the answer among all possible combinations of the grammar:

```
GrammaticalExhaustiveSearch(grammarDef, fitfunc, max.depth = 7, terminationCost = 0)
```

```
GE Search Results:
Expressions Tested: 6577
Best Chromosome:    0 1 3 0 2 1 3 1 0 0 1 1 0 0 3 0 0
Best Expression:    rex(start, maybe(or("+", "-")), maybe(numbers, "."), numbers, maybe(numbers), end)
Best Cost:          0
```

The result, while correct, is different from what we expected: $[\pm][nnn.]nnn[nnn]$, which is true for any real number. Furthermore, the search takes a considerable amount of time:

```
system.time(GrammaticalExhaustiveSearch(grammarDef, fitfunc,
                                        max.depth = 7, terminationCost = 0))
```

```
   user  system elapsed
380.469  17.022 392.637
```

which was measured a 3.40 GHz Intel Core i7-2600 CPU.

In conclusion, one might find it easier to design REs by hand in real-world scenarios, rather than using evolutionary optimisation techniques.

4 Other gramEvol functionality

In this section, some of the other functionalities of the **gramEvol** are introduced. Here, all of the examples are demonstrated using the following grammar:

```
grammarDef <- CreateGrammar(list(
  expr = gsrule("(<expr>)<op>(<expr>)", "<coef>*<var>"),
  op   = gsrule("+", "-", "*", "/"),
  coef = gsrule("c1", "c2"),
  var  = gsrule("v1", "v2")))

grammarDef

## <expr> ::= (<expr>)<op>(<expr>) | <coef>*<var>
## <op>   ::= + | - | * | /
## <coef> ::= c1 | c2
## <var>  ::= v1 | v2
```

4.1 Manual mapping

To *map* a numeric sequence to an expression manually, use `GrammarMap`:

```
GrammarMap(c(0, 1, 0, 0, 1, 1, 0, 0), grammarDef)

## (c1 * v1) - (c1 * v1)
```

The sequence is zero-indexed (the first rule is zero). To see the step by step mapping, use the `verbose` parameter option:

```
GrammarMap(c(0, 1, 0, 0, 1, 1, 0, 0), grammarDef, verbose = TRUE)

##  Step Codon Symbol Rule                 Result
##  0                 starting:            <expr>
##  1    0     <expr> (<expr>)<op>(<expr>) (<expr>)<op>(<expr>)
##  2    1     <expr> <coef>*<var>         (<coef>*<var>)<op>(<expr>)
##  3    0     <coef> c1                   (c1*<var>)<op>(<expr>)
##  4    0     <var>  v1                   (c1*v1)<op>(<expr>)
##  5    1     <op>   -                    (c1*v1)-(<expr>)
##  6    1     <expr> <coef>*<var>         (c1*v1)-(<coef>*<var>)
##  7    0     <coef> c1                   (c1*v1)-(c1*<var>)
##  8    0     <var>  v1                   (c1*v1)-(c1*v1)
## Valid Expression Found
## (c1 * v1) - (c1 * v1)
```

If the length of a sequence is insufficient for the mapping process, such that a few non-terminal elements still remain in the resulting expression, a wrapping of up to `wrappings` is performed. For example:

```
GrammarMap(c(0, 1, 0, 0, 1, 1), grammarDef, verbose = TRUE)

##  Step Codon Symbol Rule                 Result
##  0                 starting:            <expr>
##  1    0     <expr> (<expr>)<op>(<expr>) (<expr>)<op>(<expr>)
##  2    1     <expr> <coef>*<var>         (<coef>*<var>)<op>(<expr>)
##  3    0     <coef> c1                   (c1*<var>)<op>(<expr>)
##  4    0     <var>  v1                   (c1*v1)<op>(<expr>)
##  5    1     <op>   -                    (c1*v1)-(<expr>)
##  6    1     <expr> <coef>*<var>         (c1*v1)-(<coef>*<var>)
## Non-terminal expression
## Wrapping string to position 0
##  Step Codon Symbol Rule Result
##  7    0     <coef> c1   (c1*v1)-(c1*<var>)
##  8    1     <var>  v2   (c1*v1)-(c1*v2)
##  9    0     <var>  v2   (c1*v1)-(c1*v2)
## Valid Expression Found
## (c1 * v1) - (c1 * v2)
```

4.2 Examining a grammar

gramEvol offers several functions to examine grammar definitions.

`summary` reports a summary of what grammar presents:

```
summary(grammarDef)
```

```
## Start Symbol:                <expr>
## Is Recursive:                TRUE
## Tree Depth:                  Limited to 4
## Maximum Rule Choices:        4
## Maximum Sequence Length:     18
## Maximum Sequence Variation:  2 2 2 2 4 4 2 2 2 4 2 2 2 2 4 2 2 2
## No. of Unique Expressions:   18500
```

Many of these properties are available through individual functions:

`GetGrammarDepth` computes the depth of grammar tree. The parameter `max.depth` is used to limit recursion in *cyclic* grammars. For example, this grammar is cyclic because of rule $<expr> \rightarrow <expr><op><expr>$, i.e., replacing a $<expr>$ with other $<expr>$s. By default `GetGrammarDepth` limits recursion to the number of symbols defined in the grammar:

```
GetGrammarDepth(grammarDef)

## [1] 4

GetGrammarDepth(grammarDef, max.depth = 10)

## [1] 10
```

For grammars without recursion, the value returned by `GetGrammarDepth` is the actual depth of the tree:

```
grammarDef2 <- CreateGrammar(list(
  expr    = gsrule("(<subexpr>)<op>(<subexpr>)"),
  subexpr = gsrule("<coef>*<var>"),
  op      = gsrule("+", "-", "*", "/"),
  coef    = gsrule("c1", "c2"),
  var     = gsrule("v1", "v2")))

GetGrammarDepth(grammarDef2)

## [1] 3
```

`GetGrammarDepth` also supports computing the depth from any symbol:

```
GetGrammarDepth(grammarDef2, startSymb = "<subexpr>")

## [1] 2

GetGrammarDepth(grammarDef2, startSymb = "<coef>")

## [1] 1
```

`GetGrammarMaxRuleSize` returns the maximum number of production rules per symbol. Here, $<op>$ has the highest number of production rules:

```
GetGrammarMaxRuleSize(grammarDef)

## [1] 4
```

`GetGrammarNumOfExpressions` returns the number of possible expressions existing in the grammar space. This function also uses the optional argument `max.depth` to limit the number of recursions and `startSymb` to set the starting symbol:

```
GetGrammarNumOfExpressions(grammarDef)

## [1] 18500

GetGrammarNumOfExpressions(grammarDef, max.depth = 2)

## [1] 4

GetGrammarNumOfExpressions(grammarDef, startSymb = "<coef>")

## [1] 2
```

Here, the only expressions with depth of 2 or less are constructed if rule ($<coef>\times<var>$) is applied first, creating 4 expressions (i.e., $c_1 \times v_1$, $c_1 \times v_2$, $c_2 \times v_1$ and $c_2 \times v_2$). Also if $<coef>$ is chosen as the starting symbol, the expressions are limited to c_1 and c_2.

`GetGrammarMaxSequenceLen` computes the length of integer sequence required for iterating through the grammar space without wrapping. As with the previous functions, `max.depth` is set to the number of symbols defined in the grammar.

```
GetGrammarMaxSequenceLen(grammarDef)

## [1] 18

GetGrammarMaxSequenceLen(grammarDef, max.depth = 3)

## [1] 8

GetGrammarMaxSequenceLen(grammarDef2, startSymb = "<subexpr>")

## [1] 3
```

4.3 Grammatical evolution options

`GrammaticalEvolution` is defined as follows:

```
GrammaticalEvolution(grammarDef, evalFunc,
                     numExpr = 1,
                     max.depth = GrammarGetDepth(grammarDef),
                     startSymb = GrammarStartSymbol(grammarDef),
                     seqLen = GrammarMaxSequenceLen(grammarDef, max.depth, startSymb),
                     wrappings = 3,
                     suggestions = NULL,
                     optimizer = c("auto", "es", "ga"),
                     popSize = 8, newPerGen = "auto", elitism = 2,
                     mutationChance = NA,
                     iterations = 1000, terminationCost = NA,
                     monitorFunc = NULL,
                     plapply = lapply, ...)
```

`max.depth` and `startSymb` determine recursive grammar limitations, similar to what was explained in the previous section.

The rest of the parameters are the evolutionary optimisation options:

- `GrammaticalEvolution` evolves a population of `popSize` chromosomes for a number of `iterations`.
- if `optimizer` is set to "auto", using the information obtained about the grammar (e.g., number of possibles expressions and maximum sequence length), `GrammaticalEvolution` uses a heuristic algorithm based on [8] to automatically determine a suitable value for `popSize` (i.e., the population size) `iterations` (i.e., the number of iterations) parameters.
- The ordinary cross-over operator of GA is considered destructive when homologous production rules are not aligned, such as for cyclic grammars [9]. Consequently, `GrammaticalEvolution` automatically changes cross-over parameters depending on the grammar to improve optimisation results. A user can turn this off by manually setting the `optimizer`.
- The first generation is made from the `suggestions` in form of integer chromosomes, and randomly generated individuals.
- Each integer chromosome is mapped using the grammar, and its fitness is assessed by calling `evalFunc`.
- For each generation, the top n scoring chromosomes where $n =$ `elitism` are directly added to the next generation's population. The rest of the population is created using cross-over of chromosomes selected with roulette selection operator.
- Each chromosome may mutate by a probability of `mutationChance`.
- After reaching a termination criteria, e.g., the maximum number of `iterations` or the desired `terminationCost`, the algorithm stops and returns the best expression found so far.
- `GrammaticalEvolution` supports multi-gene operations, generating more than one expression per chromosome using the `numExpr` parameter.
- The number of integer codons in the chromosome is determined by `seqLen` times `numExpr` (i.e., the sequence length per expression, times the number of expressions).
- `monitorFunc` is then called with information and statistics about the current status of the population.
- `plapply` is used for parallel processing.
- `GrammaticalEvolution` automatically filters non-terminal expressions (i.e., expressions that don't yield a terminal expression even after times of `wrappings`). Therefore the end-user does not need to worry about them while using **gramEvol**.

4.4 Parallel processing option

Processing expressions and computing their fitness is often computationally expensive. The **gramEvol** package can utilise parallel processing facilities in R to improve its performance. This is done through the `plapply` argument of `GrammaticalEvolution` function. By default, `lapply` function is used to evaluate all individuals in the population.

Multi-core systems simply benefit from using `mclapply` from package **parallel**, which is a drop-in replacement for `lapply` on POSIX compatible systems. The following code optimises `evalFunc` on 4 cores:

```
library("parallel")
options(mc.cores = 4)
ge <- GrammaticalEvolution(grammarDef, evalFunc,
                           plapply = mclapply)
```

To run **gramEvol** on a cluster, `clusterapply` functions can be used instead. The **gramEvol** package must be first installed on all machines and the fitness function and its data dependencies exported before GE is called. The following example demonstrates a four-process cluster running on the local machine:

```
library("parallel")
cl <- makeCluster(type = "PSOCK", c("127.0.0.1",
                                    "127.0.0.1",
                                    "127.0.0.1",
                                    "127.0.0.1"))
clusterEvalQ(cl, library("gramEvol"))
clusterExport(cl, c("evalFunc"))
ge <- GrammaticalEvolution(grammarDef, evalFunc,
                           plapply = function(...) parLapply(cl, ...))
stopCluster(cl)
```

It must be noticed that in any problem, the speed-up achieved depends on the overhead of communication compared with the fitness functions' computational complexity.

4.5 Generating more than one expression

gramEvol supports generation and evaluation of multiple expressions:

- `numExpr` in `GrammaticalEvolution` is used to pass a list of more than one R `expression` to the fitness function.
- `EvalExpressions` offers a simpler interface for evaluating multiple expressions.

The following example show cases `EvalExpressions`: It uses a dataset for variables defined in the grammar, and evaluates a GE expression object along with a string:

```
df <- data.frame(c1 = c(1, 2),
                 c2 = c(2, 3),
                 v1 = c(3, 4),
                 v2 = c(4, 5))

quad.expr <- expression(c1 * v1, c1 * v2, c2 * v1, c2 * v2)
EvalExpressions(quad.expr, envir = df)

##   expr1 expr2 expr3 expr4
## 1     3     4     6     8
## 2     8    10    12    15
```

This is useful in applications when more than one expression is required, or the collective power of several simple expressions outperform a single complex program. For example in [10], the authors have used GE for electricity load forecasting; instead of using a complex machine learning algorithm, pools of string expressions were generated in a guided manner and were used as features in a simpler machine learning algorithm to obtain better results.

The idea of generating *features* using GE is further explored in [11].

4.6 Alternative optimisation algorithms

gramEvol also provides a random search and an exhaustive search. Their syntax is similar to the `GrammaticalEvolution`:

```
result1 <- GrammaticalExhaustiveSearch(grammarDef, evalFunc)
result2 <- GrammaticalRandomSearch(grammarDef, evalFunc)
```

4.7 Using vectors as rules

`gvrule` allows members of a vector to be used as individual rules. For example,

```
gvrule(1:5)

## Rule 0: 1L
## Rule 1: 2L
## Rule 2: 3L
## Rule 3: 4L
## Rule 4: 5L
```

which is equal to

```
grule(1,2,3,4,5)

## Rule 0: 1
## Rule 1: 2
## Rule 2: 3
## Rule 3: 4
## Rule 4: 5
```

Without `gvrule`, `1:5` would have been interpreted as a single rule:

```
grule(1:5)

## Rule 0: 1:5
```

4.8 Using commas and assignments in rules

There are two ways to use commas and assignments in **gramEvol** rules:

1. Rules are defined in character string form using `gsrule`.
2. Rules are wrapped in `.()` and defined using `grule`.

For example, consider the following rules:
<*assignment*> ::= A = B — A = C
<*comma*> ::= A, B — B, C

Their definition using **gramEvol** is as follows:

```
CreateGrammar(list(assignment = gsrule("A = B", "A = C"),
                   comma      = gsrule("A, B", "B, C")))

## <assignment> ::= A = B | A = C
## <comma>      ::= A, B | B, C
```

or

```
CreateGrammar(list(assignment = grule(.(A = B), .(A = C)),
                   comma      = grule(.(A, B), .(B, C))))

## <assignment> ::= A = B | A = C
## <comma>      ::= A, B | B, C
```

5 Conclusion

GE offers a flexible yet powerful framework for automatic program generation. The syntax and the structure of the programs are described using a context-free grammar, and their objective is determined by a cost function. An evolutionary search is performed on the grammar to find the program that minimises the cost function.

gramEvol implements GE in R. It allows a grammar to be defined using R expressions, as well as in custom string formats. A GE program generator using **gramEvol** only requires a grammar definition and a cost function, and other steps including the evolutionary search and selecting its optimal parameters are handled automatically by the package. It also supports parallel computing, and includes facilities for exhaustive and random search.

In this vignette, some of the functionalities of **gramEvol** were explored. Furthermore, two examples were used to demonstrate the flexibility of GE and **gramEvol**.

References

[1] M. O'Neill and C. Ryan, "Grammatical Evolution," *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 4, pp. 349–358, 2001.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[3] J. R. Koza, "Symbolic Regression - Error-Driven Evolution," in *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT press, 1992, vol. 1, ch. 10, pp. 237–288.

[4] C. Ferreira, "Kepler's Third Law," in *Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence*, 2nd ed. Springer, May 2006, ch. 6.3, pp. 253–257.

[5] P. Langley, H. A. Simon, and G. L. Bradshaw, "Heuristics for empirical discovery," in *Computational Models of Learning*. Springer-Verlag Berlin, 1987, pp. 21–54.

[6] A. Bartoli, G. Davanzo, A. De Lorenzo, M. Mauri, E. Medvet, and E. Sorio, "Automatic Generation of Regular Expressions from Examples with Genetic Programming," in *Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference Companion*. ACM, 2012, pp. 1477–1478. [Online]. Available: https://www.lri.fr/~hansen/proceedings/2012/GECCO/companion/p1477.pdf

[7] K. Ushey and J. Hester, **rex***: Friendly Regular Expressions*, 2014, R package version 0.2.0.99. [Online]. Available: https://github.com/kevinushey/rex

[8] K. Deb and S. Agrawal, "Understanding Interactions among Genetic Algorithm Parameters," *Foundations of Genetic Algorithms*, pp. 265–286, 1999.

[9] M. O'Neill, C. Ryan, M. Keijzer, and M. Cattolico, "Crossover in Grammatical Evolution," *Genetic Programming and Evolvable Machines*, vol. 4, no. 1, pp. 67–93, 2003.

[10] A. M. de Silva, F. Noorian, R. I. A. Davis, and P. H. W. Leong, "A Hybrid Feature Selection and Generation Algorithm for Electricity Load Prediction using Grammatical Evolution," in *IEEE 12th International Conference on Machine Learning and Applications ICMLA 2013, special session on Machine Learning in Energy Applications*, 2013, pp. 211–217. [Online]. Available: https://doi.org/10.1109/ICMLA.2013.125

[11] F. Noorian, A. M. de Silva, and P. H. W. Leong, "gramEvol: Grammatical evolution in R," *Journal of Statistical Software*, vol. 71, no. 1, pp. 1–26, 2016.