Package: gramEvol (via r-universe)

August 20, 2024

Type Package

Title Grammatical Evolution for R

Version 2.1-4

Date 2020-07-18

Author Farzad Noorian, Anthony Mihirana de Silva

Maintainer Farzad Noorian <`farzad.noorian@gmail.com`>

Description A native R implementation of grammatical evolution (GE).
GE facilitates the discovery of programs that can achieve a
desired goal. This is done by performing an evolutionary
optimisation over a population of R expressions generated via a
user-defined context-free grammar (CFG) and cost function.

URL `https://github.com/fnoorian/gramEvol/`

BugReports `https://github.com/fnoorian/gramEvol/issues`

Suggests rex, knitr

License GPL (>= 2)

VignetteBuilder knitr

RoxygenNote 5.0.1

Repository https://fnoorian.r-universe.dev

RemoteUrl https://github.com/fnoorian/gramevol

RemoteRef HEAD

RemoteSha 302bcc95f1f3d6951dcd1735bf3a9d241bb34e04

Contents

c 2
CreateGrammar 3
EvalExpressions 5
EvolutionStrategy.int 6
GeneticAlg.int 9
GrammarGetNextSequence 12

GrammarIsTerminal 14
GrammarMap 15
GrammarRandomExpression 16
GrammaticalEvolution 17
GrammaticalExhaustiveSearch 20
GrammaticalRandomSearch 22
ReplaceInExpression 24
summary 25

Index **28**

`c` *Grammar Rule Concatenation*

Description

Concatenates two or more grammar rule objects.

Usage

```
  ## S3 method for class 'GERule'
c(..., recursive=FALSE)
```

Arguments

`...`	Grammar rule objects to be concatenated.
`recursive`	Not used.

Value

A new grammar rule object.

See Also

`CreateGrammar`

Examples

```
rule1 <- grule(Func1, Func2)
rule2 <- grule(`*`, `/`)

rule.all <- c(rule1, rule2)

print(rule.all)
```

`CreateGrammar` *Context-free Grammar Object*

Description

Creates a context-free grammar object.

Usage

```
grule(...)

gsrule(...)

gvrule(vec)

CreateGrammar(ruleDef, startSymb)
```

Arguments

`...`	A series of comma separated strings or expressions, for `gsrule` and `grule` respectively. Expressions can be wrapped in `.()` to preserve their commas or assignment operators.
`vec`	An iterable vector or list.
`ruleDef`	Grammatical rule definition. Either a list of grammar rule objects (`GERule`) created using `grule` and `gsrule` with a syntax similar to Backus-Naur form, or a list of character strings representing symbols and sequences in Backus-Naur form, or a filename or `connection` to a .bnf file. See details.
`startSymb`	The symbol where the generation of a new expression should start. If not given, the first rule in `ruleDef` is used.

Details

The rule definition is the grammar described in Backus-Naur context-free grammatical format. The preferred way of defining a grammar is to create a `list` simulating BNF format, which collects several named grammar rule objects (`GERule`). Each name defines the *non-terminal symbol*, and each rule in the collection determines the *production rule*, i.e., possible *sequences* that will replace the symbol.

Defining a grammar rule object (`GERule`) can take three forms:

1. The first form uses `grule` (Grammar Rule), where R expressions are accepted. In the mapping process, variables are looked up and replaced using the production rules.

2. The second form uses `gsrule` (Grammar String Rule) and uses character strings. The input to `gsrule` are character string values, where any value surrounded by '<' or '>' is considered as *non-terminal symbols* and will be replaced using the rule with the same name in the mapping process.

Other symbols are considered terminals. This form allows generation of sequences that are not syntactically valid in R (such as `` `var op var` ``).

3. The third form uses `gvrule` (Grammar Vector Rule), where objects within an iterable (vector or list) containing all of the expressions are used as individual rules.

Alternatively, `CreateGrammar` can read and parse .bnf text files.

Value

`CreateGrammar` returns a `grammar` object.

`grule` and `gsrule` return a `GERule` object.

See Also

`c`, `GrammarMap`, `GrammaticalEvolution`

Examples

```
# Define a simple grammar in BNF format
# <expr> ::= <var><op><var>
# <op>   ::= + | - | *
# <var>  ::= A | B
ruleDef <- list(expr = gsrule("<var><op><var>"),
                op   = gsrule("+", "-", "*"),
                var  = gsrule("A", "B"))

# print rules
print(ruleDef)

# create and display a vector rule
vectorRule = gvrule(1:5)
print(vectorRule)

# Create a grammar object
grammarDef <- CreateGrammar(ruleDef)

# print grammar object
print(grammarDef)

# Creating the same grammar using R expressions
ruleDef <- list(expr = grule(op(var, var)),
                op   = grule(`+`, `-`, `*`),
                var  = grule(A, B))

grammarDef <- CreateGrammar(ruleDef)

print(grammarDef)

# Two rules with commas and assignments, preserved using .()
ruleDef <- list(expr = grule(data.frame(dat)),
                dat  = grule(.(x = 1, y = 2), .(x = 5, y = 6)))
grammarDef <- CreateGrammar(ruleDef)
```

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

`EvalExpressions` *Evaluate a collection of Expressions*

Description

`EvalExpressions` evaluates one or more expressions, either in string format or as `expression` objects.

Usage

```
EvalExpressions(expressions, envir = parent.frame())
```

Arguments

`expressions`	an expression, or a collection of expressions.
`envir`	the `environment` in which expressions are to be evaluated. May also be `NULL`, a list, a data frame, a pair-list or an integer as specified in `sys.call`.

Details

`EvalExpressions` is a wrapper around `eval` and `parse` functions in R base package. It can handle a single, a vector or a list of expressions, character strings or `GEPhenotype` objects.

The `envir` argument is directly passed to `eval` function. If it is not specified, the parent frame (i.e., the environment where the call to `eval` was made) is used instead.

`EvalExpressions` only evaluates terminal expressions and character strings. Evaluating non-terminal expressions will result in a warning and `NA` is returned.

Value

If one expression is evaluated, a vector of numeric values is returned. Otherwise a data frame with result of each expression in a separate column is returned.

See Also

`GrammarMap`

Examples

```
A <- 1:6
B <- 1

EvalExpressions("A - B")

# a vector of text strings
exprs <- c("A + B", "A - B")
```

```
EvalExpressions(exprs, data.frame(A = A, B = B))

# a vector of expressions
exprs <- expression(A + B, A - B)
EvalExpressions(exprs, data.frame(A = A, B = B))
```

`EvolutionStrategy.int` *Evolution Strategy with Integer Chromosomes*

Description

Uses evolution strategy to find the minima of a given cost function. It evolves chromosomes with limited-range integers as codons.

Usage

```
EvolutionStrategy.int(genomeLen, codonMin, codonMax,
    genomeMin = rep.int(codonMin, genomeLen),
    genomeMax = rep.int(codonMax, genomeLen),
    suggestion = NULL, popSize=4, newPerGen = 4,
    iterations = 500, terminationCost = NA,
    mutationChance = 1/(genomeLen+1),
    monitorFunc = NULL, evalFunc, allowrepeat = TRUE,
    showSettings = FALSE, verbose = FALSE, plapply = lapply)
```

Arguments

`genomeLen`	Number of integers (i.e, codons) in chromosome.
`codonMin`	Minimum integer value range for all codons.
`codonMax`	Maximum integer value range for all codons.
`genomeMin`	A vector of length `genomeLen` containing fine-grained control over each codon's minimum. Overrides `codonMin`.
`genomeMax`	A vector of length `genomeLen` containing fine-grained control over each codon's maximum. Overrides `codonMax`.
`suggestion`	A list of suggested chromosomes to be used in the initial population.
`popSize`	Size of the population generated by mutating the parent.
`newPerGen`	Number of the new randomly generated chromosome in each generation.
`iterations`	Number of generations to evolve the population.
`terminationCost`	Target cost. If the best chromosome's cost reaches this value, the algorithm terminates.
`mutationChance`	The chance of a codon being mutated. It must be between 0 and 1.
`monitorFunc`	A function that is called at each generation. Can be used to monitor evolution of population.

`evalFunc`	The cost function.
`allowrepeat`	Allows or forbids repeated integers in the chromosome.
`showSettings`	Enables printing GA settings.
`verbose`	Enables verbose debugging info.
`plapply`	`lapply` function used for mapping chromosomes to the cost function. See details for parallelization tips.

Details

`EvolutionStrategy.int` implements evolutionary strategy search algorithm with chromosomes created from integer values in the range of `codonMin` to `codonMax`. `genomeMin` and `genomeMax` allow fine-grained control of range for individual codons. It first creates an initial population, using suggested input `suggestion` or a randomly generated chromosome. Score of each chromosome is evaluated using the cost function `costFunc`. If the best chromosome reaches `terminationCost`, the algorithm terminates; otherwise only the best candidate is selected and mutated to create a new generation, and the cycle is repeated. This iteration continues until the required cost is reached or the number of generations exceeds `iterations`.

At each generation, the supplied `monitorFunc` is called with a list similar to `EvolutionStrategy.int` returning value as its argument.

The `evalFunc` receives integer sequences and must return a numeric value. The goal of optimization would be to find a chromosome which minimizes this value.

To parallelize cost function evaluation, set `plapply` to a parallelized `lapply`, such as `mclapply` from package `parallel`. In functions that do not handle data dependencies such as `parLapply`, variables and functions required for correct execution of `evalFunc` must be exported to worker nodes before invoking `EvolutionStrategy.int`.

Value

A list containing information about `settings`, `population`, and the `best` chromosome.

`settings$genomeMin`	Minimum of each codon.
`Settings$genomeMax`	Maximum of each codon.
`settings$popSize`	Size of the population created using mutation.
`settings$newPerGen`	Number of the new randomly generated chromosome in each generation.
`settings$totalPopulation`	Size of the total population.
`settings$iterations`	Number of maximum generations.
`settings$suggestion`	Suggested chromosomes.
`settings$mutationChance`	Mutation chance.

`population$population`	The genomic data of the current population.
`population$evaluations`	Cost of the latest generation.
`population$best`	Historical cost of the best chromosomes.
`population$mean`	Historical mean cost of population.
`population$currentIteration`	Number of generations evolved until now.
`best$genome`	The best chromosome in integer sequence format.
`best$cost`	The cost of the best chromosome.

See Also

`GrammaticalEvolution`, `GeneticAlg.int`

Examples

```
# define the evaluate function
evalfunc <- function(l) {
    # maximize the odd indices and minimize the even indices
    # no repeated values are allowed
    odd <- seq(1, 20, 2)
    even <- seq(2, 20, 2)
    err <- sum(l[even]) - sum(l[odd]);

    stopifnot(!any(duplicated(l))) # no duplication allowed

    return (err)
}

monitorFunc <- function(result) {
    cat("Best of gen: ", min(result$best$cost), "\n")
}

x <- EvolutionStrategy.int(genomeLen = 20, codonMin = 0, codonMax = 20,
                allowrepeat = FALSE, terminationCost = -110,
                monitorFunc = monitorFunc, evalFunc = evalfunc)

print(x)

best.result <- x$best$genome
print("Odds:")
print(sort(best.result[seq(1, 20, 2)]))
print("Evens:")
print(sort(best.result[seq(2, 20, 2)]))
```

`GeneticAlg.int` *Genetic Algorithm with Integer Chromosomes*

Description

Uses genetic algorithm to find the minima of a given cost function. It evolves chromosomes with limited-range integers as codons.

Usage

```
GeneticAlg.int(genomeLen, codonMin, codonMax,
    genomeMin = rep.int(codonMin, genomeLen),
    genomeMax = rep.int(codonMax, genomeLen),
    suggestions = NULL, popSize = 50,
    iterations = 100, terminationCost = NA,
    mutationChance = 1/(genomeLen+1), elitism = floor(popSize/10),
    geneCrossoverPoints = NULL,
    monitorFunc = NULL, evalFunc, allowrepeat = TRUE,
    showSettings = FALSE, verbose = FALSE, plapply = lapply)
```

Arguments

`genomeLen`	Number of integers (i.e, codons) in chromosome.
`codonMin`	Minimum integer value range for all codons.
`codonMax`	Maximum integer value range for all codons.
`genomeMin`	A vector of length `genomeLen` containing fine-grained control over each codon's minimum. Overrides `codonMin`.
`genomeMax`	A vector of length `genomeLen` containing fine-grained control over each codon's maximum. Overrides `codonMax`.
`suggestions`	A list of suggested chromosomes to be used in the initial population. Alternatively, an m-by-n matrix, where m is the number of suggestions and n is the chromosome length.
`popSize`	Size of the population.
`iterations`	Number of generations to evolve the population.
`terminationCost`	Target cost. If the best chromosome's cost reaches this value the algorithm terminates.
`mutationChance`	The chance of a codon being mutated. It must be between 0 and 1.
`geneCrossoverPoints`	Codon groupings (genes) to be considered while crossover occurs. If given, odd and even codon groups are exchanged between parents. Otherwise random points are selected and a classic single-point crossover is performed.
`elitism`	Number of top ranking chromosomes that are directly transfered to next generation without going through evolutionary operations.

`monitorFunc`	A function that is called at each generation. Can be used to monitor evolution of population.
`evalFunc`	The cost function.
`allowrepeat`	Allows or forbids repeated integers in the chromosome.
`showSettings`	Enables printing GA settings.
`verbose`	Enables verbose debugging info.
`plapply`	`lapply` function used for mapping chromosomes to cost function. See details for parallelization tips.

Details

`GeneticAlg.int` implements evolutionary algorithms with chromosomes created from integer values in the range of `codonMin` to `codonMax`. `genomeMin` and `genomeMax` allow fine-grained control of range for individual codons. It first creates an initial population, using suggested inputs `suggestions` and randomly generated chromosomes. Cost of each chromosome is evaluated using the cost function `evalFunc`. If one of the chromosomes reaches `terminationCost`, the algorithm terminates; Otherwise evolutionary operators including selection, cross-over and mutation are applied to the population to create a new generation. This iteration is continued until the required cost is reached or the number of generations exceeds `iterations`.

At each generation, the supplied `monitorFunc` is called with a list similar to `GeneticAlg.int` returning value as its argument.

The `evalFunc` receives integer sequences and must return a numeric value. The goal of optimization would be to find a chromosome which minimizes this value.

To parallelize cost function evaluation, set `plapply` to a parallelized `lapply`, such as `mclapply` from package `parallel`. In functions that do not handle data dependencies such as `parLapply`, variables and functions required for correct execution of `evalFunc` must be exported to worker nodes before invoking `GeneticAlg.int`.

Value

A list containing information about `settings`, `population`, and the `best` chromosome.

`settings$genomeMin`	Minimum of each codon.
`Settings$genomeMax`	Maximum of each codon.
`settings$popSize`	Size of the population.
`settings$elitism`	Number of elite individuals.
`settings$iterations`	Number of maximum generations.
`settings$suggestions`	Suggested chromosomes.
`settings$mutationChance`	Mutation chance.

`settings$geneCrossoverPoints`	Cross-over points.
`population$population`	The genomic data of the current population.
`population$evaluations`	Cost of the latest generation.
`population$best`	Historical cost of the best chromosomes.
`population$mean`	Historical mean cost of population.
`population$currentIteration`	Number of generations evolved until now.
`best$genome`	The best chromosome.
`best$cost`	The cost of the best chromosome.

References

This function is partially inspired by `genalg` package by Egon Willighagen. See `https://cran.r-project.org/package=genalg`.

See Also

`GrammaticalEvolution`, `EvolutionStrategy.int`

Examples

```
# define the evaluate function
evalfunc <- function(l) {
    # maximize the odd indices and minimize the even indices
    # no repeated values are allowed
    odd <- seq(1, 20, 2)
    even <- seq(2, 20, 2)
    err <- sum(l[even]) - sum(l[odd]);

    stopifnot(!any(duplicated(l))) # no duplication allowed

    return (err)
}

monitorFunc <- function(result) {
    cat("Best of gen: ", min(result$best$cost), "\n")
}

x <- GeneticAlg.int(genomeLen = 20, codonMin = 0, codonMax = 20,
                allowrepeat = FALSE, terminationCost = -110,
                monitorFunc = monitorFunc, evalFunc = evalfunc)

print(x)

best.result <- x$best$genome
```

```
print("Odds:")
print(sort(best.result[seq(1, 20, 2)]))
print("Evens:")
print(sort(best.result[seq(2, 20, 2)]))
```

`GrammarGetNextSequence` *Grammar Iterator*

Description

Iterates through grammar's valid sequences.

Usage

```
GrammarGetFirstSequence(grammar,
            seqStart = NULL,
            startSymb = GrammarStartSymbol(grammar),
            max.depth = GrammarGetDepth(grammar),
            max.len = GrammarMaxSequenceLen(grammar, max.depth, startSymb))

GrammarGetNextSequence(grammar,
            seqStart = NULL,
            startSymb = GrammarStartSymbol(grammar),
            max.depth = GrammarGetDepth(grammar),
            max.len = GrammarMaxSequenceLen(grammar, max.depth, startSymb))

is.GrammarOverflow(object)
```

Arguments

`grammar`	A `grammar` object.
`seqStart`	The sequence to be incremented. For a value of `NULL`, the first sequence is returned. Partial sequences are completed and returned.
`startSymb`	The non-terminal symbol where the generation of a new expression should start.
`max.depth`	Maximum depth of recursion, in case of a cyclic grammar. By default it is limited to the number of production rules in the grammar.
`max.len`	Maximum length of sequence to return. Used to avoid recursion.
`object`	An object to be tested.

Details

`GrammarGetFirstSequence` returns the first sequence that creates a valid expression with the given `grammar` object. `GrammarGetNextSequence` allows iterating through all valid sequences in a grammar. If a `seqStart = NULL` is used, `GrammarGetFirstSequence` is called to and the first sequence in the grammar is returned. Calling `GrammarGetNextSequence` or `GrammarGetFirstSequence` with an incomplete sequence returns a full-length sequence starting with the given `seqStart`.

When `GrammarGetNextSequence` reaches the last of all valid sequences, it returns a `GrammarOverflow` object. This object can be identified using `is.GrammarOverflow`.

Value

`GrammarGetFirstSequence` returns a numeric vector representing the first sequence of the grammar.

`GrammarGetNextSequence` returns a numeric vector or a `GrammarOverflow` object.

`is.GrammarOverflow` returns TRUE if `object` is a `GrammarOverflow`, otherwise FALSE.

See Also

`GrammaticalExhaustiveSearch`

Examples

```
# Define a simple grammar
# <expr> ::= <var><op><var>
# <op>   ::= + | - | *
# <var>  ::= A | B
ruleDef <- list(expr = gsrule("<var><op><var>"),
                op   = gsrule("+", "-", "*"),
                var  = gsrule("A", "B"))

# Create a grammar object
grammarDef <- CreateGrammar(ruleDef)

# Iterate and print all valid sequence and expressions
string <- NULL
while (TRUE) {
  string <- GrammarGetNextSequence(grammarDef, string)

  if (is.GrammarOverflow(string)) {
    break
  }

  expr <- GrammarMap(string, grammarDef)
  cat(string, " -> ", as.character(expr), "\n")
}

# test a partial string
GrammarGetNextSequence(grammarDef, c(0, 0, 2))
```

`GrammarIsTerminal` *Non-terminal Phenotype test.*

Description

Checks a phenotype object for containing non-terminal symbols.

Usage

```
GrammarIsTerminal(x)
```

Arguments

`x`	A `GEPhenotype` object.

Value

`TRUE` if phenotype is terminal, `FALSE` otherwise.

See Also

`GrammarMap`

Examples

```
# Define a recursive grammar
# <expr> ::= <expr>+<expr> | var
# <var>  ::= A | B | C
ruleDef <- list(expr = grule(expr+expr, var),
                var =  grule(A, B, C))

# Create a grammar object
grammarDef <- CreateGrammar(ruleDef)

# a short sequence leading to infinite recursion
sq <- c(0)
expr <- GrammarMap(sq, grammarDef)

print(expr)

# check the phenotype for being non-terminal
print(GrammarIsTerminal(expr))


# a terminal sequence
sq <- c(0, 1, 0, 1, 2)
expr <- GrammarMap(sq, grammarDef)

print(expr)
print(GrammarIsTerminal(expr))
```

`GrammarMap` *Sequence to Expression Mapping using Context-free Grammar*

Description

Converts a sequence of integer numbers to an expression using a `grammar` object.

Usage

```
GrammarMap(inputString, grammar, wrappings = 3, verbose = FALSE)
```

Arguments

`inputString`	A vector of integers to define the path of symbol selection in grammar tree. It uses zero-based indexing to address production rules in the grammar.
`grammar`	A `grammar` object.
`wrappings`	The number of times the function is allowed to wrap around `inputString` if non-terminal symbols are still remaining.
`verbose`	Prints out each steps of grammar mapping.

Details

`GrammarMap` starts from the `startExpr` defined in the `grammar` object; then it iterates through `inputString`, replacing symbols in the expression with associated replacements in the grammar using the current value of `inputString`.

If the function exhausts all non-terminal symbols in the expression, it terminates. If the end of `inputString` is reached and still non-terminal symbols exist, the algorithm will restart from the beginning of the current `inputString`. To avoid unlimited recursions in case of a cyclic grammar, `wrappings` variable limits the number of this restart.

If `verbose = TRUE`, step-by-step replacement of symbols with production rules are displayed.

`GrammarMap` returns a `GEPhenotype` object, which can be converted to a character string using `as.character`, or an R expression with `as.expression`.

Value

A `GrammarMap` returns a `GEPhenotype` object.

`expr`	The generated expression as a character string.
`parsed`	The generated expression. NULL if the expression still contains non-terminal symbols.
`type`	"T" if the expression is valid, "NT" if the expression still contains non-terminal symbols.

See Also

`GrammarIsTerminal` `CreateGrammar`, `GrammarRandomExpression`

Examples

```
# Define a simple grammar
# <expr> ::= <var><op><var>
# <op>   ::= + | - | *
# <var>  ::= A | B | C
ruleDef <- list(expr = gsrule("<var><op><var>"),
                op =   gsrule("+", "-", "*"),
                var =  grule(A, B, C))

# Create a grammar object
grammarDef <- CreateGrammar(ruleDef)

# this should create the expression "A - C"
# <expr>         -> 0 -> <var><op><var>
# <var><op><var> -> 0 -> A<op><var>
# A<op><var>     -> 1 -> A - <var>
# A - <var>      -> 2 -> A - C
sq <- c(0, 0, 1, 2)
expr <- GrammarMap(sq, grammarDef, verbose = TRUE)

print(expr)

# check the expression as a character string
stopifnot(as.character(expr) == "A - C")

# evaluate the expression
A = 5; C = 1
eval(as.expression(expr))
```

`GrammarRandomExpression`

Random Expression Generation

Description

Creates random expressions from context-free grammar.

Usage

```
GrammarRandomExpression(grammar,
           numExpr = 1,
           max.depth = length(grammar$def),
           startSymb = GrammarStartSymbol(grammar),
           max.string = GrammarMaxSequenceRange(grammar, max.depth, startSymb,
                                                approximate = TRUE),
           wrappings = 3,
           retries = 100)
```

Arguments

`grammar`	A `grammar` object.
`numExpr`	Number of random expressions to generate.
`max.depth`	Maximum depth of recursion, in case of a cyclic grammar. By default it is limited to the number of production rules in the grammar.
`startSymb`	The symbol where the generation of a new expression should start.
`max.string`	Maximum value for each element of the sequence.
`wrappings`	The number of times the function is allowed to wrap around `inputString` if non-terminal symbols are still remaining.
`retries`	Number of retries until a terminal and valid expressions is found.

Details

`GrammarRandomExpression` creates `num.expr` random expressions from the given `grammar`. It can be used to quickly examine the expressibility of the grammar, or as a form of random search over the grammar.

Value

An `expressions`, or a `list` of `expressions`.

Examples

```
# Define a simple grammar
# <expr> ::= <var><op><var>
# <op>   ::= + | - | *
# <var>  ::= A | B | C
ruleDef <- list(expr = gsrule("<var><op><var>"),
                op =   gsrule("+", "-", "*"),
                var =  grule(A, B, C))

# Create a grammar object
grammarDef <- CreateGrammar(ruleDef)

# Generate 5 random expressions
exprs <- GrammarRandomExpression(grammarDef, 5)
print(exprs)
```

`GrammaticalEvolution` *Grammatical Evolution*

Description

Evolves an expression using a context-free grammar to minimize a given cost function.

Usage

```
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 = "auto", newPerGen = "auto", elitism = 2,
              mutationChance = NA,
              iterations = "auto",
              terminationCost = NA,
              monitorFunc = NULL,
              disable.warnings=FALSE,
              plapply = lapply, ...)

## S3 method for class 'GrammaticalEvolution'
print(x, ..., show.genome = FALSE)
```

Arguments

`grammarDef`	A `grammar` object.
`evalFunc`	The cost function, taking a string or a collection of strings containing the expression(s) as its input and returning the cost of the expression(s).
`numExpr`	Number of expressions generated and given to `evalFunc`.
`max.depth`	Maximum depth of search in case of a cyclic grammar. By default it is limited to the number of production rules in the grammar.
`startSymb`	The symbol where the generation of a new expression should start.
`seqLen`	Length of integer vector used to create the expression.
`wrappings`	Number of wrappings in case the length of chromosome is not enough for conversion to an expression.
`suggestions`	Suggested chromosomes to be added to the initial population pool. if `optimizer` parameter is set to `"es"`, only a single chromosome (as a numeric vector) is acceptable. For `"ga"` mode, a list of numeric vectors.
`optimizer`	The evolutionary optimizer. `"es"` uses evolution strategy as in `EvolutionStrategy.int` and `"ga"` uses genetic algorithm as in `GeneticAlg.int`. `"auto"` chooses evolution strategy when `numExpr = 1`, and genetic algorithm otherwise. If `"auto"` is used, `popSize` and `iterations` are tweaked based on the grammar as well.
`popSize`	Population size in the evolutionary optimizer. By default, 8 for ES and 48 for GA.
`newPerGen`	Number of randomly generated individuals in evolution strategy. If “auto", it is set to 25% of population of grammar if it is not recursive, otherwise to all of it.
`elitism`	Number of top ranking chromosomes that are directly transfered to the next generation without going through evolutionary operations, used in genetic algorithm optimizer.

`iterations`	Number of maximum iterations in the evolutionary optimizer. By default, 1000 for `"es"` optimizer and 200 for `"ga"`.
`terminationCost`	Target cost. If a sequence with this cost or less is found, the algorithm terminates.
`mutationChance`	Mutation chance in the evolutionary optimizer. It must be between 0 and 1. By default it is set to `1/(1+chromosome size))` for genetic algorithm and `10/(1+chromosome size))` for evolution strategy.
`monitorFunc`	A function that is called at each generation. It can be used to monitor evolution of population.
`disable.warnings`	If `TRUE`, suppresses any warnings generated while evaulating `evalFuncs`.
`plapply`	`lapply` function used for mapping chromosomes to the cost function. See details for parallelization tips.
`...`	Additional parameters are passed to `GeneticAlg.int` or `EvolutionStrategy.int`.
`x`	Grammatical Evolution results.
`show.genome`	Prints the numeric value of genome if TRUE.

Details

This function performs an evolutionary search over the grammar, better known as Grammatical Evolution. It evolves integer sequences and converts them to a collection containing `numExpr` expression. These expressions can be evaluated using `eval` function. The `evalFunc` receives these expressions and must return a numeric value. The goal of optimization would be to find a chromosome which minimizes this function.

Two evolutionary optimizers are supported: Genetic algorithm and evolution strategy, which are set by the `optimizer` parameter.

Only valid (i.e., terminal) expressions are passed to `evalFunc`, and it is guaranteed that `evalFunc` receives at least one expression.

If the grammar contains recursive elements, it is advisable that `chromosomeLen` is defined manually, as in such cases the possible search space grows explosively with the recursion. The evolutionary algorithm automatically removes the recursive chromosomes from the population by imposing a penalty for chromosomes creating expressions with non-terminal elements.

`monitorFunc` receives a list similar to the `GrammaticalEvolution`'s return value.

Value

The results of `GeneticAlg.int` or `EvolutionStrategy.int` with an additional item:

`best$expressions`	Expression(s) with the best cost.

See Also

`CreateGrammar`, `GeneticAlg.int`, `EvolutionStrategy.int`, `EvalExpressions`

Examples

```
# Grammar Definition
ruleDef <- list(expr     = gsrule("<der.expr><op><der.expr>"),
                der.expr = grule(func(var), var),
                func     = grule(log, exp, sin, cos),
                op       = gsrule("+", "-", "*"),
                var      = grule(A, B, n),
                n        = grule(1, 2, 3, 4))

# Creating the grammar object
grammarDef <- CreateGrammar(ruleDef)

# cost function
evalFunc <- function(expr) {
  # expr: a string containing a symbolic expression
  # returns: Symbolic regression Error
  A <- 1:6
  B <- c(2, 5, 8, 3, 4, 1)

  result <- eval(as.expression(expr))

  X <- log(A) * B
  err <- sum((result - X)^2)

  return(err)
}

# invoke grammatical evolution (with default parameters)
ge <- GrammaticalEvolution(grammarDef, evalFunc, terminationCost = 0.001)

# print results
print(ge, sequence = TRUE)
```

`GrammaticalExhaustiveSearch`

Exhaustive Search

Description

Exhaustive Search within context-free grammar.

Usage

```
GrammaticalExhaustiveSearch(grammar, evalFunc,
                  max.depth = GrammarGetDepth(grammar),
                  startSymb = GrammarStartSymbol(grammar),
                  max.len = GrammarMaxSequenceLen(grammar, max.depth, startSymb),
                  wrappings = 3,
                  terminationCost = NA,
                  monitorFunc = NULL)
```

Arguments

`grammar`	A `grammar` object.
`evalFunc`	The evaluation function, taking an expression as its input and returning the cost (i.e., the score) of the expression.
`max.depth`	Maximum depth of recursion, in case of a cyclic grammar. By default it is limited to the number of production rules in the grammar.
`startSymb`	The symbol where the generation of a new expression should start.
`max.len`	Maximum length of the sequences to search. By default it is determined by `max.depth`.
`wrappings`	The number of times the function is allowed to wrap around `inputString` if non-terminal symbols are still remaining.
`terminationCost`	Target cost. If an expression with this cost or less is found, the algorithm terminates.
`monitorFunc`	A function that is called at each iteration. It can be used to monitor the search.

Details

`GrammaticalExhaustiveSearch` performs an exhaustive search, iterating through all possible expressions that can be generated by the `grammar`, to find the expression that minimises `evalFunc`.

The search terminates when all possible expressions are exhausted, or when an expression with a cost less than `terminationCost` is discovered.

If a `monitorFunc` is given, it is called for each expression, and it receives a list similar to the `GrammaticalExhaustiveSearch`'s return value with the information availabe for that expression.

Value

`bestExpression`	The Best expresssion.
`bestSequence`	Best expresssion's generating sequence.
`bestCost`	Best expresssion's cost.
`numExpr`	Number of evaluated expressions.

In addition, the `monitorFunc` receives the following additional slots:

`currentExpression`	The current expresssion.
`currentSequence`	Current expresssion's generating sequence.
`currentCost`	Current expresssion's cost.

See Also

`GrammarGetNextSequence`, `GrammaticalEvolution`

Examples

```
library("gramEvol")

ruleDef <- list(expr = gsrule("<var><op><var>"),
                op   = gsrule("+", "-", "*"),
                var  = gsrule("A", "B"))

# Create a grammar object
grammarDef <- CreateGrammar(ruleDef)


# use exhaustive search to find the sequence for creating "B - A"
evalFunc <- function(expr) {
  if (as.character(expr) == "B - A") {
    return(0) # Minimum error
  } else {
    return(1) # maximum error
  }
}

res <- GrammaticalExhaustiveSearch(grammarDef, evalFunc, terminationCost = 0)

print(res)
```

`GrammaticalRandomSearch` *Random Search*

Description

Random Search within context-free grammar.

Usage

```
GrammaticalRandomSearch(grammar, evalFunc,
                           max.depth = GrammarGetDepth(grammar),
                           startSymb = GrammarStartSymbol(grammar),
                           wrappings = 3,
                           iterations = 1000,
                           terminationCost = NA,
                           monitorFunc = NULL)
```

Arguments

`grammar`	A `grammar` object.
`evalFunc`	The evaluation function, taking an expression as its input and returning the cost (i.e., the score) of the expression.

`max.depth`	Maximum depth of recursion, in case of a cyclic grammar. By default it is limited to the number of production rules in the grammar.
`startSymb`	The symbol where the generation of a new expression should start.
`wrappings`	The number of times the function is allowed to wrap around `inputString` if non-terminal symbols are still remaining.
`iterations`	Number of random expressions to test.
`terminationCost`	Target cost. If an expression with this cost or less is found, the algorithm terminates.
`monitorFunc`	A function that is called at each generation. It can be used to monitor evolution of population.

Details

`GrammaticalRandomSearch` performs a random search within expressions that can be generated by the `grammar`, to find the expression that minimises `evalFunc`.

The search terminates when either the predetermined number of `iterations` are reached, or when an expression with a cost less than `terminationCost` is discovered.

If a `monitorFunc` is given, it is called for each expression, and it receives a list similar to the `GrammaticalExhaustiveSearch`'s return value with the information availabe for that expression.

Value

`bestExpression`	The Best expresssion.
`bestSequence`	Best expresssion's generating sequence.
`bestCost`	Best expresssion's cost.
`numExpr`	Number of evaluated expressions.
`population`	A matrix of sequences that were tested.
`populationCost`	Numeric value of cost of sequences that were tested.

In addition, the `monitorFunc` receives the following additional slots:

`currentExpression`	The current expresssion.
`currentSequence`	Current expresssion's generating sequence.
`currentCost`	Current expresssion's cost.

See Also

`GrammarGetNextSequence`, `GrammaticalEvolution`

Examples

```
library("gramEvol")

ruleDef <- list(expr = gsrule("<var><op><var>"),
                op   = gsrule("+", "-", "*"),
                var  = gsrule("A", "B"))

# Create a grammar object
grammarDef <- CreateGrammar(ruleDef)


# use exhaustive search to find the sequence for creating "B - A"
evalFunc <- function(expr) {
  if (as.character(expr) == "B - A") {
    return(0) # Minimum error
  } else {
    return(1) # maximum error
  }
}

# search and terminate after getting to cost = 0
res <- GrammaticalRandomSearch(grammarDef, evalFunc, terminationCost = 0)

print(res)
```

`ReplaceInExpression` *Replace as sub-expression isnide an expression*

Description

Replace every subexpression equal to or starting with `what` in `expr`. Replacement is performed by passing the whole subexpression to `replacer.func`, which should be a `function(x, ...)`, where `x` is the expression and return the desirable expression.

Usage

```
ReplaceInExpression(expr, what, replacer.func, ...)
```

Arguments

`expr`	An `expression`.
`what`	A backquoted expression to find in `expr`.
`replacer.func`	A `function(x, ...)` to process the subexpression.
`...`	Other parameters passed to `replacer.func`.

Details

This function was designed to be used as a runtime processing tool for grammar generated expression. This allows the user to modify the resulting expression on the fly based on runtime variables, without including them in the grammar. See examples section.

Value

An expression

References

See http://adv-r.had.co.nz/Expressions.html by Hadley Wickham.

Examples

```
expr = expression(function(x) {
    cbind(f1(x),
          f2(x),
          g3(y))
})
expr
ReplaceInExpression(expr, bquote(f2), function(x) {NULL})
ReplaceInExpression(expr, bquote(f2), function(x) {bquote(f2(y))})
ReplaceInExpression(expr, bquote(g3), function(x) {bquote(f3(x))})
ReplaceInExpression(expr, bquote(g3), function(x, b) {if (b > 1) x else NULL}, b = 0)
ReplaceInExpression(expr, bquote(g3), function(x, b) {if (b > 1) x else NULL}, b = 2)
```

`summary` *Context-free Grammar Object Information*

Description

Examines a context-free grammar object.

Usage

```
## S3 method for class 'grammar'
summary(object, ...)

GrammarStartSymbol(grammar)

GrammarIsRecursive(grammar, startSymb = GrammarStartSymbol(grammar), ...)

GrammarGetDepth(grammar, max.depth = max(length(grammar$def), 4),
    startSymb = GrammarStartSymbol(grammar), ...)

GrammarMaxSequenceLen(grammar, max.depth = GetGrammarDepth(grammar),
    startSymb = GrammarStartSymbol(grammar), ...)
```

```
GrammarMaxRuleSize(grammar)

GrammarMaxSequenceRange(grammar, max.depth = GrammarGetDepth(grammar),
    startSymb = GrammarStartSymbol(grammar), approximate = FALSE, ...)

GrammarNumOfExpressions(grammar, max.depth = GrammarGetDepth(grammar),
    startSymb = GrammarStartSymbol(grammar), ...)
```

Arguments

`grammar, object`	A `grammar` object.
`max.depth`	Maximum depth of search in case of a cyclic grammar. By default it is limited to the maximum of 4 or the number of production rules in the grammar.
`startSymb`	The symbol where the generation of a new expression should start.
`approximate`	If True, results are approximated. Useful for recursive grammars, where number of valid expressions prohibits an accurate measurement.
`...`	unused inputs.

Value

`summary` returns a `summary.grammar` object, with the following slots which are obtained from the other functions:

`Start.Symbol`	`GrammarStartSymbol` returns the grammar's starting symbol.
`Is.Recursive`	`GrammarIsRecursive` returns TRUE if grammar contains a recursive element.
`Tree.Depth`	`GrammarGetDepth` returns the depth of the grammar. It is limited to `max.depth` for a recursive grammar.
`Maximum.Sequence.Length`	`GrammarMaxSequenceLen` returns the maximum length of a sequence needed to generate an expression without wrapping.
`Maximum.Rule.Size`	`GrammarMaxRuleSize` returns the largest rule size in the grammar.
`Maximum.Sequence.Variation`	`GrammarMaxSequenceRange` returns a numeric sequence, with each of its elements holding the highest range that the same position in all sequences can hold.
`No.of.Unique.Expressions`	`GrammarNumOfExpressions` returns the number of expressions a grammar can generate.

See Also

`CreateGrammar`, `GrammarMap`

Examples

```
# Define a simple grammar
# <expr> ::= <var><op><var>
# <op>   ::= + | - | *
# <var>  ::= A | B
ruleDef <- list(expr = gsrule("<var><op><var>"),
                op   = gsrule("+", "-", "*"),
                var  = gsrule("A", "B"))

# Create a grammar object
grammarDef <- CreateGrammar(ruleDef)

# summarize grammar object
summary(grammarDef)
```

Index

as.character.GEPhenotype *(GrammarMap)*, 15
as.expression.GEPhenotype *(GrammarMap)*, 15

c, 2, *4*
connection, *3*
CreateGrammar, 2, 3, *15*, *19*, *26*

EvalExpressions, 5, *19*
EvolutionStrategy.int, 6, *11*, *18*, *19*
expression, *24*

GeneticAlg.int, *8*, 9, *18*, *19*
GEPhenotype, *14*
GetGrammarDepth *(summary)*, 25
GetGrammarMaxRuleSize *(summary)*, 25
GetGrammarMaxSequenceLen *(summary)*, 25
GetGrammarNumOfExpressions *(summary)*, 25
grammar, *12*, *15*, *17*, *18*, *21*, *22*, *26*
GrammarGenotypeToPhenotype *(GrammarMap)*, 15
GrammarGetDepth *(summary)*, 25
GrammarGetFirstSequence *(GrammarGetNextSequence)*, 12
GrammarGetNextSequence, 12, *21*, *23*
GrammarIsRecursive *(summary)*, 25
GrammarIsTerminal, 14, *15*
GrammarMap, *4*, *5*, *14*, 15, *26*
GrammarMaxRuleSize *(summary)*, 25
GrammarMaxSequenceLen *(summary)*, 25
GrammarMaxSequenceRange *(summary)*, 25
GrammarNumOfExpressions *(summary)*, 25
GrammarRandomExpression, *15*, 16
GrammarStartSymbol *(summary)*, 25
GrammaticalEvolution, *4*, *8*, *11*, 17, *21*, *23*
GrammaticalExhaustiveSearch, *13*, 20
GrammaticalRandomSearch, 22
grule *(CreateGrammar)*, 3
gsrule *(CreateGrammar)*, 3
gvrule *(CreateGrammar)*, 3

is.GrammarOverflow *(GrammarGetNextSequence)*, 12

print.EvolutionStrategy.int *(EvolutionStrategy.int)*, 6
print.GeneticAlg.int *(GeneticAlg.int)*, 9
print.GEPhenotype *(GrammarMap)*, 15
print.GERule *(CreateGrammar)*, 3
print.GESearch *(GrammaticalExhaustiveSearch)*, 20
print.grammar *(CreateGrammar)*, 3
print.GrammarOverflow *(GrammarGetNextSequence)*, 12
print.GrammaticalEvolution *(GrammaticalEvolution)*, 17
print.summary.grammar *(summary)*, 25

ReplaceInExpression, 24

summary, 25
summary.grammar *(summary)*, 25