Package ‘xegaGpGene’

April 17, 2025

Title Genetic Operations for Grammar-Based Genetic Programming

Version 1.0.0.2

Description An implementation of
the representation-dependent gene level operations of grammar-based
genetic programming with genes which are derivation trees
of a context-free grammar: Initialization of a gene with a
complete random derivation tree, decoding of a derivation tree.
Crossover is implemented by exchanging subtrees. Depth-bounds
for the minimal and the maximal depth of the roots of the subtrees
exchanged by crossover can be set.
Mutation is implemented by replacing a subtree by a random subtree.
The depth of the random subtree and the insertion node are
configurable. For details,
see Geyer-Schulz (1997, ISBN:978-3-7908-0830-X).

License MIT + file LICENSE

URL https://github.com/ageyerschulz/xegaGpGene

Encoding UTF-8

RoxygenNote 7.3.2

Suggests testthat

Imports stats, xegaBNF, xegaDerivationTrees, xegaSelectGene

NeedsCompilation no

Author Andreas Geyer-Schulz [aut, cre]
(<https://orcid.org/0009-0000-5237-3579>)

Maintainer Andreas Geyer-Schulz <Andreas.Geyer-Schulz@kit.edu>

Repository CRAN

Date/Publication 2025-04-17 07:30:02 UTC

Contents

`findCrossoverExample` *Prints a random example of crossover for a crossover method given a random number seed.*

Description

The function supports the search for examples for unit tests for crossover functions whose behavior depends on random numbers.

Usage

```
findCrossoverExample(FUN, s, verbose = TRUE)
```

Arguments

`FUN`	String. Specification of crossover method.
`s`	Integer. Seed of random number generator.
`verbose`	Boolean. If `TRUE` (default), print the example to the console.

Value

No return.

Examples

```
findCrossoverExample(FUN="AllCross2Gene", s=2)
```

`lFxegaGpGene` *Generate local functions and objects.*

Description

We enhance the configurability of our code by introducing a function factory. The function factory contains all the functions that are needed for defining local functions in genetic operators. The local function list keeps the signatures of functions (e.g. mutation functions) uniform and small. At the same time, variants of functions can use different local functions.

Usage

```
lFxegaGpGene
```

Format

An object of class `list` of length 25.

Details

We use the local function list for

1. replacing all constants by constant functions.

   Rationale: We need one formal argument (the local function list lF) and we can dispatch multiple functions. E.g. `lF$verbose()`
2. dynamically binding a local function with a definition from a proper function factory. E.g., the selection methods `lF$SelectGene()` and `lF$SelectMate()`.
3. gene representations which require special functions to handle them: For example, `lF$InitGene()`, `lF$DecodeGene()`, `lF$EvalGene()`, `lF$ReplicateGene()`, ...

`xegaGpAllCross2Gene` *Crossover of 2 derivation tree genes.*

Description

`xegaGpAllCross2Gene()` swaps two randomly extracted subtrees between 2 genes. Subtrees must have the same root in order to be compatible. The current implementation performs at most `lF$MaxTrials()` trials to find compatible subtrees. If this fails, the original genes are returned.

Usage

```
xegaGpAllCross2Gene(ng1, ng2, lF)
```

Arguments

`ng1`	Derivation tree.
`ng2`	Derivation tree.
`lF`	Local configuration of the genetic algorithm.

Details

Crossover is controlled by one local parameter:

- `lF$MaxTrials()`: Maximal number of trials to find compatible subtrees. If compatible subtrees are not found, the gene is returned unchanged.

Value

List of 2 derivation trees.

See Also

Other Crossover: `xegaGpAllCrossGene()`, `xegaGpFilterCross2Gene()`, `xegaGpFilterCrossGene()`

Examples

```
gene1<-xegaGpInitGene(lFxegaGpGene)
gene2<-xegaGpInitGene(lFxegaGpGene)
xegaGpDecodeGene(gene1, lFxegaGpGene)
xegaGpDecodeGene(gene2, lFxegaGpGene)
newgenes<-xegaGpAllCross2Gene(gene1, gene2,  lFxegaGpGene)
xegaGpDecodeGene(newgenes[[1]], lFxegaGpGene)
xegaGpDecodeGene(newgenes[[2]], lFxegaGpGene)
```

`xegaGpAllCrossGene` *Crossover of 2 derivation tree genes.*

Description

`xegaGpAllCrossGene()` swaps two randomly extracted subtrees between 2 genes. Subtrees must have the same root in order to be compatible. The current implementation performs at most `lF$MaxTrials()` attempts to find compatible subtrees. If this fails, the original gene is returned.

Usage

```
xegaGpAllCrossGene(ng1, ng2, lF)
```

Arguments

`ng1`	Derivation tree.
`ng2`	Derivation tree.
`lF`	Local configuration of the genetic algorithm.

Details

Crossover is controlled by one local parameter:

- `lF$MaxTrials()`: Maximal number of trials to find compatible subtrees. If compatible subtrees are not found, the gene is returned unchanged.

Value

List of 1 derivation tree.

See Also

Other Crossover: `xegaGpAllCross2Gene()`, `xegaGpFilterCross2Gene()`, `xegaGpFilterCrossGene()`

Examples

```
gene1<-xegaGpInitGene(lFxegaGpGene)
gene2<-xegaGpInitGene(lFxegaGpGene)
xegaGpDecodeGene(gene1, lFxegaGpGene)
xegaGpDecodeGene(gene2, lFxegaGpGene)
newgene<-xegaGpAllCrossGene(gene1, gene2,  lFxegaGpGene)
xegaGpDecodeGene(newgene[[1]], lFxegaGpGene)
```

`xegaGpCrossoverFactory` *Configure the crossover function of a grammar-based genetic algorithm.*

Description

`xegaGpCrossoverFactory()` implements the selection of one of the crossover functions in this package by specifying a text string. The selection fails ungracefully (produces a runtime error), if the label does not match. The functions are specified locally.

Current support:

1. Crossover functions with two kids:
   (a) "Cross2Gene" returns `xegaGpAllCross2Gene()`.
   (b) "AllCross2Gene" returns `xegaGpAllCross2Gene()`.
   (c) "FilterCross2Gene" returns `xegaGpFilterCross2Gene()`.
2. Crossover functions with one kid:
   (a) "AllCrossGene" returns `xegaGpAllCrossGene()`.
   (b) "FilterCrossGene" returns `xegaGpFilterCrossGene()`.

Usage

```
xegaGpCrossoverFactory(method = "Cross2Gene")
```

Arguments

`method`	String specifying the crossover function.

Value

Crossover function for genes.

See Also

Other Configuration: `xegaGpInitGeneFactory()`, `xegaGpMutationFactory()`

Examples

```
XGeneTwo<-xegaGpCrossoverFactory("Cross2Gene")
XGeneOne<-xegaGpCrossoverFactory("FilterCrossGene")
gene1<-xegaGpInitGene(lFxegaGpGene)
gene2<-xegaGpInitGene(lFxegaGpGene)
XGeneTwo(gene1, gene2, lFxegaGpGene)
XGeneOne(gene1, gene2, lFxegaGpGene)
```

`xegaGpDecodeGene` *Decode a derivation tree.*

Description

`xegaGpDecodeGene()` decodes a derivation tree.

Usage

```
xegaGpDecodeGene(gene, lF)
```

Arguments

`gene`	Derivation tree.
`lF`	Local configuration of the genetic algorithm.

Details

The recursive algorithm for the decoder is imported from the package `xegaDerivationTrees`.

Value

Decoded gene.

Examples

```
gene<-xegaGpInitGene(lFxegaGpGene)
xegaGpDecodeGene(gene, lFxegaGpGene)
```

xegaGpFilterCross2Gene

Crossover of 2 derivation tree genes with node filter.

Description

`xegaGpFilterCross2Gene()` swaps two randomly extracted subtrees between 2 genes. Subtrees must have the same root in order to be compatible. The current implementation performs at most `lF$MaxTrials()` trials to find compatible subtrees. If this fails, the original genes are returned. Only nodes with a depth between `lF$MinMutInsertionDepth()` and `lF$MaxMutInsertionDepth()` are considered as candidate roots of derivation trees to be swapped by crossover.

Usage

```
xegaGpFilterCross2Gene(ng1, ng2, lF)
```

Arguments

`ng1`	Derivation tree.
`ng2`	Derivation tree.
`lF`	Local configuration of the genetic algorithm.

Details

Crossover is controlled by three local parameters:

- `lF$MinCrossDepth()` and `lF$MaxCrossDepth()` control the possible exchange points for subtrees. The depth of the exchange node must be between `lF$MinMutInsertionDepth()` and `lF$MaxMutInsertionDepth()`.
- `lF$MaxTrials()`: Maximal number of trials to find compatible subtrees. If compatible subtrees are not found, the gene is returned unchanged.

Value

List of 2 derivation trees.

See Also

Other Crossover: `xegaGpAllCross2Gene()`, `xegaGpAllCrossGene()`, `xegaGpFilterCrossGene()`

Examples

```
gene1<-xegaGpInitGene(lFxegaGpGene)
gene2<-xegaGpInitGene(lFxegaGpGene)
xegaGpDecodeGene(gene1, lFxegaGpGene)
xegaGpDecodeGene(gene2, lFxegaGpGene)
newgenes<-xegaGpFilterCross2Gene(gene1, gene2,  lFxegaGpGene)
xegaGpDecodeGene(newgenes[[1]], lFxegaGpGene)
xegaGpDecodeGene(newgenes[[2]], lFxegaGpGene)
```

xegaGpFilterCrossGene *Crossover of 2 derivation tree genes with node filter.*

Description

`xegaGpFilterCrossGene()` swaps two randomly extracted subtrees between 2 genes. Subtrees must have the same root in order to be compatible. The current implementation performs at most `lF$MaxTrials()` attempts to find compatible subtrees. If this fails, the original gene is returned. Only nodes with a depth between `lF$MinMutInsertionDepth()` and `lF$MaxMutInsertionDepth()` are considered as candidate roots of derivation trees to be swapped by crossover.

Usage

```
xegaGpFilterCrossGene(ng1, ng2, lF)
```

Arguments

`ng1`	Derivation tree.
`ng2`	Derivation tree.
`lF`	Local configuration of the genetic algorithm.

Details

Crossover is controlled by three local parameters:

- `lF$MinCrossDepth()` and `lF$MaxCrossDepth()` control the possible exchange points for subtrees. The depth of the exchange node must be between `lF$MinMutInsertionDepth()` and `lF$MaxMutInsertionDepth()`.
- `lF$MaxTrials()`: Maximal number of trials to find compatible subtrees. If compatible subtrees are not found, the gene is returned unchanged.

Value

List of 1 derivation tree.

See Also

Other Crossover: `xegaGpAllCross2Gene()`, `xegaGpAllCrossGene()`, `xegaGpFilterCross2Gene()`

Examples

```
gene1<-xegaGpInitGene(lFxegaGpGene)
gene2<-xegaGpInitGene(lFxegaGpGene)
xegaGpDecodeGene(gene1, lFxegaGpGene)
xegaGpDecodeGene(gene2, lFxegaGpGene)
newgene<-xegaGpFilterCrossGene(gene1, gene2,  lFxegaGpGene)
xegaGpDecodeGene(newgene[[1]], lFxegaGpGene)
```

`xegaGpGene` *Package xegaGpGene.*

Description

Genetic operations for grammar-based genetic algorithms.

Details

For derivation tree genes, the `xegaGpGene` package provides

- Gene initiatilization.
- Decoding of parameters.
- Mutation functions as well as a function factory for configuration.
- Crossover functions as well as a function factory for configuration. Crossover functions can be restricted by depth or by the non-terminal symbols which are allowed as roots of the subtrees which are exchanged between 2 genes. We provide two families of crossover functions:
  1. Crossover functions with two kids: Crossover preserves the genetic information in the gene pool.
  2. Crossover functions with one kid: These functions allow the construction of evaluation pipelines for genes. One advantage of this is a simple control structure at the population level.

Derivation Tree Gene Representation

A derivation tree gene is a named list:

- `$gene1`: The gene must be a complete derivation tree.
- `$fit`: The fitness value of the gene (for EvalGeneDet() and EvalGeneU()) or the mean fitness (for stochastic functions evaluated with EvalGeneStoch()).
- `$evaluated`: Boolean. Has the gene been evaluated?
- `$evalFail`: Boolean. Has the evaluation of the gene failed?
- `$var`: The variance of the fitness of all evaluations of a gene is updated after each evaluation of a gene. (For stochastic functions.)
- `$sigma`: The standard deviation of the fitness of all evaluations of a gene. (For stochastic functions.)
- `$obs`: The number evaluations of a gene. (For stochastic functions.)

Abstract Interface of Problem Environment

A problem environment `penv` must provide:

- `$f(word, gene, lF)`: Function with a word of a language (a program) as first argument which computes the fitness of the gene.

Abstract Interface of Mutation Functions

Each mutation function has the following function signature:

`newGene<-Mutate(gene, lF)`

All local parameters of the mutation function configured are expected be available in the local function list lF.

Local Constants of Mutation Functions

The local constants of a mutation function determine the behavior of the function.

Constant	Default	Used in
`lF$MaxMutDepth()`	3	xegaGpMutateAllGene(),
	3	xegaGpMutateFilterGene()
`lF$MinMutInsertionDepth()`	1	xegaGpMutateFilterGene()
`lF$MaxMutInsertionDepth()`	7	xegaGpMutateFilterGene()

Abstract Interfaces of Crossover Functions

The signatures of the abstract interface to the 2 families of crossover functions are:

`ListOfTwoGenes<-Crossover2(gene1, gene2, lF)`

`ListOfOneGene<-Crossover(gene1, gene2, lF)`

All local parameters of the crossover function configured are expected to be available in the local function list lF.

Local Constants of Crossover Functions

Constant	Default	Used in
`lF$MinCrossDepth()`	1	xegaGpFilterCross2Gene(), xegaGpFilterCrossGene(),
`lF$MaxCrossDepth()`	7	xegaGpFilterCross2Gene(), xegaGpFilterCrossGene(),
`lF$MaxTrials()`	5	xegaGpAllCross2Gene() xegaGpAllCrossGene(), xegaGpFilter2CrossGene(), xegaGpFilterCrossGene(),

The Architecture of the xegaX-Packages

The xegaX-packages are a family of R-packages which implement eXtended Evolutionary and Genetic Algorithms (xega). The architecture has 3 layers, namely the user interface layer, the population layer, and the gene layer:

- The user interface layer (package `xega`) provides a function call interface and configuration support for several algorithms: genetic algorithms (sga), permutation-based genetic algorithms (sgPerm), derivation-free algorithms as e.g. differential evolution (sgde), grammar-based genetic programming (sgp) and grammatical evolution (sge).
- The population layer (package `xegaPopulation`) contains population-related functionality as well as support for population statistics dependent adaptive mechanisms and parallelization.
- The gene layer is split in a representation independent and a representation dependent part:
  1. The representation-indendent part (package `xegaSelectGene`) is responsible for variants of selection operators, evaluation strategies for genes, as well as profiling and timing capabilities.
  2. The representation-dependent part consists of the following packages:
     - `xegaGaGene` for binary coded genetic algorithms.
     - `xegaPermGene` for permutation-based genetic algorithms.
     - `xegaDfGene` for derivation-free algorithms as e.g. differential evolution.
     - `xegaGpGene` for grammar-based genetic algorithms.
     - `xegaGeGene` for grammatical evolution algorithms.

     The packages `xegaDerivationTrees` and `xegaBNF` support the last two packages: `xegaBNF` essentially provides a grammar compiler and `xegaDerivationTrees` is an abstract data type for derivation trees.

Copyright

(c) 2023 Andreas Geyer-Schulz

License

MIT

URL

<https://github.com/ageyerschulz/xegaGpGene>

Installation

From CRAN by `install.packages('xegaGpGene')`

Author(s)

Andreas Geyer-Schulz

References

Geyer-Schulz, Andreas (1997): *Fuzzy Rule-Based Expert Systems and Genetic Machine Learning*, Physica, Heidelberg. (ISBN:978-3-7908-0830-X)

See Also

Useful links:

- `https://github.com/ageyerschulz/xegaGpGene`

`xegaGpInitGene` *Generates a gene as a random derivation tree.*

Description

For a given grammar, `xegaGpInitGene()` generates a gene as a random derivation tree with a depth-bound.

Usage

```
xegaGpInitGene(lF)
```

Arguments

`lF`	Local configuration of the genetic algorithm.

Details

In the derivation tree representation of package `xegaGpGene`, a *gene* is a list with

1. `$gene1`: a derivation tree.
2. `$fit`: The fitness of the genotype of `$gene1`
3. `$evaluated`: Boolean: TRUE if the fitness is known.
4. `$evalFail`: Has the evaluation of the gene failed?
5. `$var`: The cumulative variance of the fitness of all evaluations of a gene. (For stochastic functions)
6. `$sigma`: The standard deviation of the fitness of all evaluations of a gene. (For stochastic functions)
7. `$obs`: The number of evaluations of a gene. (For stochastic functions)

The algorithm for generating a complete derivation tree with a depth-bound is imported from the package `xegaDerivationTrees`.

Value

Derivation tree.

See Also

Other Gene Generation: `xegaGpInitGeneGe()`

Examples

```
gene<-xegaGpInitGene(lFxegaGpGene)
xegaGpDecodeGene(gene, lFxegaGpGene)
```

`xegaGpInitGeneFactory` *Configure the initialization function of grammar-based genetic programming.*

Description

`xegaGpInitGeneFactory()` implements the creation of a complete derivation tree by one an algorithm selected by specifying a text string. The selection fails ungracefully (produces a runtime error), if the label does not match. The functions are specified locally.

Current support:

1. "InitGene" returns `xegaGpInitGene()`.
2. "InitGeneGe" returns `xegaGpInitGeneGe()`.

Usage

```
xegaGpInitGeneFactory(method = "InitGene")
```

Arguments

`method`	String specifying the mutation function.

Value

Initialization function for genes.

See Also

Other Configuration: `xegaGpCrossoverFactory()`, `xegaGpMutationFactory()`

Examples

```
InitGene<-xegaGpInitGeneFactory("InitGene")
gene1<-InitGene(lFxegaGpGene)
InitGene<-xegaGpInitGeneFactory("InitGeneGe")
gene2<-InitGene(lFxegaGpGene)
```

xegaGpInitGeneGe *Generates a gene as a random derivation tree from a random integer vector.*

Description

For a given grammar, `xegaGpInitGene()` generates a gene as a random derivation tree with a depth-bound. This function uses almost the same initialization algorithm as for grammatical evolution.

Usage

```
xegaGpInitGeneGe(lF)
```

Arguments

`lF`	Local configuration of the genetic algorithm.

Details

In the derivation tree representation of package `xegaGpGene`, a *gene* is a list with

1. `$gene1`: a derivation tree.
2. `$fit`: The fitness of the genotype of `$gene1`
3. `$evaluated`: Boolean: TRUE if the fitness is known.
4. `$evalFail`: Has the evaluation of the gene failed?
5. `$var`: The cumulative variance of the fitness of all evaluations of a gene. (For stochastic functions)
6. `$sigma`: The standard deviation of the fitness of all evaluations of a gene. (For stochastic functions)
7. `$obs`: The number of evaluations of a gene. (For stochastic functions)

The algorithm for generating a complete derivation tree with a depth-bound is imported from the package `xegaDerivationTrees`.

Value

Derivation tree.

See Also

Other Gene Generation: `xegaGpInitGene()`

Examples

```
gene<-xegaGpInitGeneGe(lFxegaGpGene)
xegaGpDecodeGene(gene, lFxegaGpGene)
```

`xegaGpMutateAllGene` *Mutate a gene.*

Description

`xegaGpMutateAllGene()` replaces a randomly selected subtree by a random derivation tree with the same root symbol with a small probability. All non-terminal nodes are considered as insertion points. Depth-bounds are respected.

Usage

```
xegaGpMutateAllGene(g, lF)
```

Arguments

`g`	Derivation tree.
`lF`	Local configuration of the genetic algorithm.

Details

Mutation is controlled by one local parameter:

1. `lF$MaxMutDepth()` controls the maximal depth of the new random generation tree.

This version of the genetic operator skips the filter loop.

Value

Derivation tree.

See Also

Other Mutation: `xegaGpMutateFilterGene()`

Examples

```
gene1<-xegaGpInitGene(lFxegaGpGene)
xegaGpDecodeGene(gene1, lFxegaGpGene)
gene<-xegaGpMutateAllGene(gene1, lFxegaGpGene)
xegaGpDecodeGene(gene, lFxegaGpGene)
```

xegaGpMutateFilterGene

Mutate a gene (with a node filter)

Description

`xegaGpMutateGeneFilter()` replaces a randomly selected subtree by a random derivation tree with the same root symbol with a small probability. Only non-terminal nodes with a depth between `lF$MinMutInsertionDepth()` and `lF$MaxMutInsertionDepth()` are considered as tree insertion points. Depth-bounds are respected.

Usage

```
xegaGpMutateFilterGene(g, lF)
```

Arguments

`g`	Derivation tree.
`lF`	Local configuration of the genetic algorithm.

Details

Mutation is controlled by three local parameters:

1. `lF$MaxMutDepth()` controls the maximal depth of the new random generation tree.
2. `lF$MinMutInsertionDepth()` and `lF$MaxMutInsertionDepth()` control the possible insertion points for the new random derivation tree. The depth of the insertion node must be between `lF$MinMutInsertionDepth()` and `lF$MaxMutInsertionDepth()`.

Value

Derivation tree.

See Also

Other Mutation: `xegaGpMutateAllGene()`

Examples

```
gene1<-xegaGpInitGene(lFxegaGpGene)
xegaGpDecodeGene(gene1, lFxegaGpGene)
gene<-xegaGpMutateFilterGene(gene1, lFxegaGpGene)
xegaGpDecodeGene(gene, lFxegaGpGene)
```

`xegaGpMutationFactory` *Configure the mutation function of a genetic algorithm.*

Description

`xegaGpMutationFactory()` implements the selection of one of the mutation functions in this package by specifying a text string. The selection fails ungracefully (produces a runtime error), if the label does not match. The functions are specified locally.

Current support:

1. "MutateGene" returns `xegaGpMutateAllGene()`.
2. "MutateAllGene" returns `xegaGpMutateAllGene()`.
3. "MutateFilterGene" returns `xegaGpMutateFilterGene()`.

Usage

```
xegaGpMutationFactory(method = "MutateGene")
```

Arguments

`method`	String specifying the mutation function.

Value

Mutation function for genes.

See Also

Other Configuration: `xegaGpCrossoverFactory()`, `xegaGpInitGeneFactory()`

Examples

```
Mutate<-xegaGpMutationFactory("MutateGene")
gene1<-xegaGpInitGene(lFxegaGpGene)
gene1
Mutate(gene1, lFxegaGpGene)
```

Index