Package 'xegaGaGene'

July 28, 2025

Title Binary Gene Operations for Genetic Algorithms

Version 1.0.0.4

Description Representation-dependent gene level operations of a genetic algorithm with binary coded genes:
Initialization of random binary genes, several gene maps for binary genes, several mutation operators, several crossover operators with 1 and 2 kids, replication
pipelines for 1 and 2 kids, and, last but not least, function factories for configuration.
See Goldberg, D. E. (1989, ISBN:0-201-15767-5).
For crossover operators, see
Syswerda, G. (1989, ISBN:1-55860-066-3),
Spears, W. and De Jong, K. (1991, ISBN:1-55860-208-9).
For mutation operators, see
Stanhope, S. A. and Daida, J. M. (1996, ISBN:0-18-201-031-7).

License MIT + file LICENSE

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

Encoding UTF-8

RoxygenNote 7.3.2

Suggests testthat (>= 3.0.0)

Imports xegaSelectGene

NeedsCompilation no

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

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

Repository CRAN

Date/Publication 2025-07-28 16:50:02 UTC

Contents

`Gray2Bin` *Map Gray code to binary.*

Description

Map Gray code to binary.

Usage

```
Gray2Bin(x)
```

Arguments

x	Gray code (boolean vector).

Details

Start with the highest order bit, and `r[k-i]<- xor(n[k], n[k-1])`.

Value

Binary code (boolean vector).

References

Gray, Frank (1953): Pulse Code Communication. US Patent 2 632 058.

Examples

```
Gray2Bin(c(1, 0, 0, 0))
Gray2Bin(c(1, 1, 1, 1))
```

---

lFxegaGaGene *The local function list lFxegaGaGene.*

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

```
lFxegaGaGene
```

Format

An object of class `list` of length 32.

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$SelectMate()`.
3. gene representations which require special functions to handle them: `lF$InitGene()`, `lF$DecodeGene()`, `lF$EvalGene()`, `lF$ReplicateGene()`, ...

See Also

Other Configuration: xegaGaCrossoverFactory(), xegaGaGeneMapFactory(), xegaGaMutationFactory(), xegaGaReplicationFactory()

`newCross2Mut1Pipeline` *Converts two genes into a pipeline with crossover (2 kids) and mutation for first kid.*

Description

The pipeline is `evaluate(accept((crossover o mutation), gene, gene1))`. Mutation is applied to the first kid.

Usage

```
newCross2Mut1Pipeline(g, g1, lF)
```

Arguments

g	A gene.
g1	A gene.
lF	The local function configuration.

Value

Closure of genetic operator pipeline with crossover with 2 kids and mutation on the first kid only. The argument of the closure `lF` configures the behavior of the pipeline.

See Also

Other Genetic Operator Pipelines: newCross2Mut2Pipeline(), newCross2Pipeline(), newCrossMut2Pipeline(), newCrossMutPipeline(), newCrossPipeline(), newMutPipeline(), newPipeline()

Examples

```
lFxegaGaGene$CrossGene<-xegaGaCross2Gene
lFxegaGaGene$MutationRate<-function(fit, lF) {0.5}
lFxegaGaGene$CrossRate<-function(fit, lF) {0.5}
lFxegaGaGene$Accept<-function(OpPipeline, gene, lF) {OpPipeline(gene, lF)}
g<-xegaGaInitGene(lFxegaGaGene)
g1<-xegaGaInitGene(lFxegaGaGene)
a<-newCross2Mut1Pipeline(g, g1, lFxegaGaGene)
print(a)
a(lFxegaGaGene)
```

`newCross2Mut2Pipeline` *Converts two genes into a pipeline with crossover (2 kids) and mutation for second kid.*

Description

The pipeline is `evaluate(accept((crossover o mutation), gene, gene1))`. Mutation is applied to the second kid.

Usage

```
newCross2Mut2Pipeline(g, g1, lF)
```

Arguments

`g`	A gene.
`g1`	A gene.
`lF`	The local function configuration.

Value

Closure of genetic operator pipeline with crossover with 2 kids and mutation on the second kid only. The argument of the closure `lF` configures the behavior of the pipeline.

See Also

Other Genetic Operator Pipelines: `newCross2Mut1Pipeline()`, `newCross2Pipeline()`, `newCrossMut2Pipeline()`, `newCrossMutPipeline()`, `newCrossPipeline()`, `newMutPipeline()`, `newPipeline()`

Examples

```
lFxegaGaGene$CrossGene<-xegaGaCross2Gene
lFxegaGaGene$MutationRate<-function(fit, lF) {0.5}
lFxegaGaGene$CrossRate<-function(fit, lF) {0.5}
lFxegaGaGene$Accept<-function(OpPipeline, gene, lF) {OpPipeline(gene, lF)}
g<-xegaGaInitGene(lFxegaGaGene)
g1<-xegaGaInitGene(lFxegaGaGene)
a<-newCross2Mut1Pipeline(g, g1, lFxegaGaGene)
print(a)
a(lFxegaGaGene)
```

`newCross2Pipeline` *Converts two genes into a genetic operator pipeline with crossover (2 kids).*

Description

The pipeline is `evaluate(accept(crossover, gene, gene1))`. The execution of this pipeline produces two genes.

Usage

```
newCross2Pipeline(g, g1, lF)
```

Arguments

`g`	A gene.
`g1`	A gene.
`lF`	The local function configuration.

Value

Closure of genetic operator pipeline with crossover only. The argument of the closure `lF` configures the behavior of the pipeline.

See Also

Other Genetic Operator Pipelines: `newCross2Mut1Pipeline()`, `newCross2Mut2Pipeline()`, `newCrossMut2Pipeline()`, `newCrossMutPipeline()`, `newCrossPipeline()`, `newMutPipeline()`, `newPipeline()`

Examples

```
lFxegaGaGene$CrossGene<-xegaGaCross2Gene
lFxegaGaGene$MutationRate<-function(fit, lF) {0.5}
lFxegaGaGene$CrossRate<-function(fit, lF) {0.5}
lFxegaGaGene$Accept<-function(OpPipeline, gene, lF) {OpPipeline(gene, lF)}
g<-xegaGaInitGene(lFxegaGaGene)
g1<-xegaGaInitGene(lFxegaGaGene)
a<-newCross2Pipeline(g, g1, lFxegaGaGene)
print(a)
a(lFxegaGaGene)
```

`newCrossMut2Pipeline` *Converts two genes into a pipeline with crossover and mutation for both kids.*

Description

The pipeline is `evaluate(accept((crossover o mutation), gene, gene1))`. Mutation is applied to both kids.

Usage

```
newCrossMut2Pipeline(g, g1, lF)
```

Arguments

`g`	A gene.
`g1`	A gene.
`lF`	The local function configuration.

Value

Closure of genetic operator pipeline with crossover with two kids and mutation on both kids. The argument of the closure `lF` configures the behavior of the pipeline.

See Also

Other Genetic Operator Pipelines: `newCross2Mut1Pipeline()`, `newCross2Mut2Pipeline()`, `newCross2Pipeline()`, `newCrossMutPipeline()`, `newCrossPipeline()`, `newMutPipeline()`, `newPipeline()`

Examples

```
lFxegaGaGene$CrossGene<-xegaGaCross2Gene
lFxegaGaGene$MutationRate<-function(fit, lF) {0.5}
lFxegaGaGene$CrossRate<-function(fit, lF) {0.5}
lFxegaGaGene$Accept<-function(OpPipeline, gene, lF) {OpPipeline(gene, lF)}
g<-xegaGaInitGene(lFxegaGaGene)
g1<-xegaGaInitGene(lFxegaGaGene)
a<-newCrossMut2Pipeline(g, g1, lFxegaGaGene)
print(a)
a(lFxegaGaGene)
```

`newCrossMutPipeline` *Converts a gene into a genetic operator pipeline with crossover and mutation (a function closure).*

Description

The pipeline is `evaluate(accept((crossover o mutation), gene, gene1))`. The symbol `o` is short for `mutation(crossover(gene, gene1))` in the accept function.

Usage

```
newCrossMutPipeline(g, g1, lF)
```

Arguments

`g`	A gene.
`g1`	A gene.
`lF`	The local function configuration.

Value

Closure of genetic operator pipeline with mutation and crossover. The argument of the closure `lF` configures the behavior of the pipeline.

See Also

Other Genetic Operator Pipelines: `newCross2Mut1Pipeline()`, `newCross2Mut2Pipeline()`, `newCross2Pipeline()`, `newCrossMut2Pipeline()`, `newCrossPipeline()`, `newMutPipeline()`, `newPipeline()`

Examples

```
lFxegaGaGene$CrossGene<-xegaGaCrossGene
lFxegaGaGene$MutationRate<-function(fit, lF) {0.5}
lFxegaGaGene$CrossRate<-function(fit, lF) {0.5}
lFxegaGaGene$Accept<-function(OperatorPipeline, gene, lF) {gene}
g<-xegaGaInitGene(lFxegaGaGene)
g1<-xegaGaInitGene(lFxegaGaGene)
a<-newCrossMutPipeline(g, g1, lFxegaGaGene)
print(a)
a(lFxegaGaGene)
```

newCrossPipeline *Converts two genes into a genetic operator pipeline with crossover (1 kid).*

Description

The pipeline is `evaluate(accept(crossover, gene, gene1))`.

Usage

```
newCrossPipeline(g, g1, lF)
```

Arguments

g	A gene.
g1	A gene.
lF	The local function configuration.

Value

Closure of genetic operator pipeline with crossover only. The argument of the closure `lF` configures the behavior of the pipeline.

See Also

Other Genetic Operator Pipelines: `newCross2Mut1Pipeline()`, `newCross2Mut2Pipeline()`, `newCross2Pipeline()`, `newCrossMut2Pipeline()`, `newCrossMutPipeline()`, `newMutPipeline()`, `newPipeline()`

Examples

```
lFxegaGaGene$CrossGene<-xegaGaCrossGene
lFxegaGaGene$MutationRate<-function(fit, lF) {0.5}
lFxegaGaGene$CrossRate<-function(fit, lF) {0.5}
lFxegaGaGene$Accept<-function(OperatorPipeline, gene, lF) {gene}
g<-xegaGaInitGene(lFxegaGaGene)
g1<-xegaGaInitGene(lFxegaGaGene)
a<-newCrossPipeline(g, g1, lFxegaGaGene)
print(a)
a(lFxegaGaGene)
```

`newMutPipeline` *Converts a gene into a genetic operator pipeline with mutation (a function closure).*

Description

The pipeline is `evaluate(accept(mutate, gene))`.

Usage

```
newMutPipeline(g, lF)
```

Arguments

`g`	A gene.
`lF`	The local function configuration.

Value

Closure of genetic operator pipeline with mutation only. The argument of the closure `lF` configures the behavior of the pipeline.

See Also

Other Genetic Operator Pipelines: `newCross2Mut1Pipeline()`, `newCross2Mut2Pipeline()`, `newCross2Pipeline()`, `newCrossMut2Pipeline()`, `newCrossMutPipeline()`, `newCrossPipeline()`, `newPipeline()`

Examples

```
lFxegaGaGene$CrossGene<-xegaGaCrossGene
lFxegaGaGene$MutationRate<-function(fit, lF) {0.5}
lFxegaGaGene$CrossRate<-function(fit, lF) {0.5}
lFxegaGaGene$Accept<-function(OperatorPipeline, gene, lF) {gene}
g<-xegaGaInitGene(lFxegaGaGene)
a<-newMutPipeline(g, lFxegaGaGene)
print(a)
a(lFxegaGaGene)
```

newPipeline *Converts a gene into a genetic operator pipeline (a function closure).*

Description

The pipeline is `evaluate(gene)`.

Usage

```
newPipeline(g, lF)
```

Arguments

`g`	A gene.
`lF`	The local function configuration.

Details

`newPipeline` is a constructor of a function closure which contains the evaluation of a gene.

Value

Closure of genetic operator pipeline without mutation and crossover. The argument of the closure `lF` configures the behavior of the pipeline.

See Also

Other Genetic Operator Pipelines: `newCross2Mut1Pipeline()`, `newCross2Mut2Pipeline()`, `newCross2Pipeline()`, `newCrossMut2Pipeline()`, `newCrossMutPipeline()`, `newCrossPipeline()`, `newMutPipeline()`

Examples

```
lFxegaGaGene$CrossGene<-xegaGaCrossGene
lFxegaGaGene$MutationRate<-function(fit, lF) {0.5}
lFxegaGaGene$CrossRate<-function(fit, lF) {0.5}
lFxegaGaGene$Accept<-function(OperatorPipeline, gene, lF) {gene}
g<-xegaGaInitGene(lFxegaGaGene)
a<-newPipeline(g, lFxegaGaGene)
print(a)
a(lFxegaGaGene)
```

without *Returns elements of vector* x *without elements in* y.

Description

Returns elements of vector x without elements in y.

Usage

```
without(x, y)
```

Arguments

x	A vector.
y	A vector.

Value

A vector.

Examples

```
a<-sample(1:15,15, replace=FALSE)
b<-c(1, 3, 5)
without(a, b)
```

xegaGaCross2Gene *One point crossover of 2 genes.*

Description

xegaGaCross2Gene() randomly determines a cut point. It combines the bits before the cut point of the first gene with the bits after the cut point from the second gene (kid 1). It combines the bits before the cut point of the second gene with the bits after the cut point from the first gene (kid 2). It returns 2 genes.

Usage

```
xegaGaCross2Gene(gg1, gg2, lF)
```

Arguments

gg1	A binary gene.
gg2	A binary gene.
lF	The local configuration of the genetic algorithm.

Value

A list of 2 binary genes.

See Also

Other Crossover (Returns 2 Kids): `xegaGaUCross2Gene()`, `xegaGaUPCross2Gene()`

Examples

```
gene1<-xegaGaInitGene(lFxegaGaGene)
gene2<-xegaGaInitGene(lFxegaGaGene)
xegaGaDecodeGene(gene1, lFxegaGaGene)
xegaGaDecodeGene(gene2, lFxegaGaGene)
newgenes<-xegaGaCross2Gene(gene1, gene2, lFxegaGaGene)
xegaGaDecodeGene(newgenes[[1]], lFxegaGaGene)
xegaGaDecodeGene(newgenes[[2]], lFxegaGaGene)
```

---

xegaGaCrossGene *One point crossover of 2 genes.*

Description

`xegaGaCrossGene()` randomly determines a cut point. It combines the bits before the cut point of the first gene with the bits after the cut point from the second gene (kid 1).

Usage

```
xegaGaCrossGene(gg1, gg2, lF)
```

Arguments

`gg1`	A binary gene.
`gg2`	A binary gene.
`lF`	The local configuration of the genetic algorithm.

Value

A list of one binary gene.

See Also

Other Crossover (Returns 1 Kid): `xegaGaUCrossGene()`, `xegaGaUPCrossGene()`

Examples

```
gene1<-xegaGaInitGene(lFxegaGaGene)
gene2<-xegaGaInitGene(lFxegaGaGene)
xegaGaDecodeGene(gene1, lFxegaGaGene)
xegaGaDecodeGene(gene2, lFxegaGaGene)
gene3<-xegaGaCrossGene(gene1, gene2, lFxegaGaGene)
xegaGaDecodeGene(gene3[[1]], lFxegaGaGene)
```

---

xegaGaCrossoverFactory

Configure the crossover function of a genetic algorithm.

Description

`xegaGaCrossoverFactory()` 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 `xegaGaCross2Gene()`.
   (b) "UCross2Gene" returns `xegaGaUCross2Gene()`.
   (c) "UPCross2Gene" returns `xegaGaUPCross2Gene()`.
2. Crossover functions with one kid:
   (a) "CrossGene" returns `xegaGaCrossGene()`.
   (b) "UCrossGene" returns `xegaGaUCrossGene()`.
   (c) "UPCrossGene" returns `xegaGaUPCrossGene()`.

Usage

```
xegaGaCrossoverFactory(method = "Cross2Gene")
```

Arguments

`method`	A string specifying the crossover function.

Details

Crossover operations which return 2 kids preserve the genetic material in the population. However, because we work with fixed size populations, genes with 2 offsprings fill two slots in the new population with their genetic material.

Value

A crossover function for genes.

See Also

Other Configuration: lFxegaGaGene, xegaGaGeneMapFactory(), xegaGaMutationFactory(), xegaGaReplicationFacto

Examples

```
XGene<-xegaGaCrossoverFactory("Cross2Gene")
gene1<-xegaGaInitGene(lFxegaGaGene)
gene2<-xegaGaInitGene(lFxegaGaGene)
XGene(gene1, gene2, lFxegaGaGene)
```

---

xegaGaDecodeGene *Decode a gene.*

Description

xegaGaDecodeGene() decodes a binary gene.

Usage

```
xegaGaDecodeGene(gene, lF)
```

Arguments

gene	A binary gene (the genotype).
lF	The local configuration of the genetic algorithm.

Value

The decoded gene (the phenotype).

See Also

Other Decoder: xegaGaGeneMap(), xegaGaGeneMapGray(), xegaGaGeneMapIdentity(), xegaGaGeneMapPerm()

Examples

```
gene<-xegaGaInitGene(lFxegaGaGene)
xegaGaDecodeGene(gene, lFxegaGaGene)
```

xegaGaGene *Package xegaGaGene.*

Description

Genetic operations for binary coded genetic algorithms.

Details

For an introduction to this class of algorithms, see Goldberg, D. (1989).

For binary-coded genes, the xegaGaGene package provides

- Gene initiatilization.
- Decoding of parameters as well as a function factory for configuration.
- Mutation functions as well as a function factory for configuration.
- Crossover functions as well as a function factory for configuration. 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 gene evaluation pipelines. One advantage of this is a simple control structure at the population level.
- Constructors for abstract genetic operator pipelines embedded in function closures.
- Gene replication functions as well as a function factory for configuration. The replication functions implement control flows for sequences of gene operations. For xegaReplicateGene, an acceptance step has been added. Simulated annealing algorithms can be configured e.g. by configuring uniform random selection combined with a Metropolis Acceptance Rule and a suitable cooling schedule.
- Constructors for abstract genetic operator pipelines embedded in function closures.
- Gene replication functions which compile function closures with genetic operator pipelines.

Binary Gene Representation

A binary gene is a named list:

- $gene1 the gene must be a binary vector.
- $fit the fitness value of the gene (for EvalGeneDet and EvalGeneU) or the mean fitness (for stochastic functions evaluated with EvalGeneStoch).
- $evaluated has the gene been evaluated?
- $evalFail has the evaluation of the gene failed?
- $var the cumulative variance of the fitness of all evaluations 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 of evaluations of a gene. (For stochastic functions)

Abstract Interface of Problem Environment

A problem environment penv must provide:

- `$f(parameters, gene, lF)`: Function with a real parameter vector as first argument which returns a gene with evaluated fitness.
- $genelength(): The number of bits of the binary-coded real parameter vector. Used in `InitGene`.
- $bitlength(): A vector specifying the number of bits used for coding each real parameter. If `penv$bitlength()[1]` is `20`, then `parameters[1]` is coded by 20 bits. Used in `GeneMap`.
- $lb(): The lower bound vector of each parameter. Used in `GeneMap`.
- $ub(): The upper bound vector of each parameter. Used in `GeneMap`.

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 in the local function list lF.

Local Constants of Mutation Functions

The local constants of a mutation function determine the behavior of the function. The default values in the table below are set in `lFxegaGaGene`.

Constant	Default	Used in
lF$BitMutationRate1()	0.01	xegaGaMutateGene()
		xegaGaIVAdaptiveMutateGene()
lF$BitMutationRate2()	0.20	xegaGaIVAdaptiveMutateGene()
lF$CutoffFit()	0.5	xegaGaIVADaptiveMutateGene()

Abstract Interface 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 in the local function list lF.

Local Constants of Crossover Functions

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

Constant	Default	Used in
lF$UCrossSwap()	0.2	UPCross2Gene()
		UPCrossGene()

Abstract Gene Operator Pipelines

Compiling abstract gene operator pipelines based on random experiments generates function closures which when evaluated produce an evaluated gene. With pipelines, the gene life cycle looks like this:

evaluated gene -> replicate -> replicated gene -> evaluate -> evaluated gene

where evaluated genes are data structures (named lists) and replicated genes are function closures (genetic operator pipelines bound with selected genes).

Pipelines lead to the following separation of work between the replication and the evaluation phase of the genetic algorithm

1. The replication phase of the genetic algorithm consists of compiling function closures.
2. The evaluation phase consists of the actual execution of the complete genetic operator pipeline: `evaluate(accept(mutate(crossover(gene1, gene2))))`.

For sequential execution models, pipelines may lead to small performance gains, because the compilation phase produces minimal genetic operator pipelines.

For parallel or distributed execution models, the replication phase is still executed sequentially whereas the evaluation of the population of function closures is done in parallel. The effect is that most of the computational work of the genetic algorithm is shifted to the evaluation phase and thus executed in parallel.

The pipeline mechanism implemented is completely abstract and independent of the actual gene representation. However, algorithms like e.g. differential evolution which use more than two genes may need replication functions which compile appropriate function closures.

Abstract Interface of Gene Replication Functions

The signatures of the abstract interface to the 4 gene replication functions are:

ListOfTwoGenes<-Replicate2Gene(pop, fit, lF)

ListOfOneGene<-ReplicateGene(pop, fit, lF)

ListOfFunctionClosures<-Replicate2GenePipeline(pop, fit, lF)

ListOfFunctionClosures<-ReplicateGenePipeline(pop, fit, lF)

Configuration and Constants of Replication Functions

Configuration for ReplicateGene (1 Kid, Default).

Function	Default	Configured By
lF$SelectGene()	SelectSUS()	SelectGeneFactory()
lF$SelectMate()	SelectSUS()	SelectGeneFactory()
lF$CrossGene()	CrossGene()	xegaGaCrossoverFactory()
lF$MutateGene()	MutateGene()	xegaGaMutationFactory()
lF$Accept()	AcceptNewGene()	AcceptFactory()

Configuration for Replicate2Gene (2 Kids).

Function	Default	Configured By
lF$SelectGene()	SelectSUS()	SelectGeneFactory()
lF$SelectMate()	SelectSUS()	SelectGeneFactory()
lF$CrossGene()	Cross2Gene()	xegaGaCrossoverFactory()
lF$MutateGene()	MutateGene()	xegaGaMutationFactory()

Global Constants.

Global constants specify the probability that a mutation or crossover operator is applied to a gene. In the xega-architecture, these rates can be configured to be adaptive.

Constant	Default	Used in
lF$MutationRate()	1.0 (static)	xegaGaReplicateGene()
		xegaGaReplicate2Gene()
lF$CrossRate()	0.2 (static)	xegaGaReplicateGene()
		xegaGaReplicate2Gene()

Local Constants.

Constant	Default	Used in
lF$BitMutationRate1()	0.01	xegaGaMutateGene()
		xegaGaIVAdaptiveMutateGene()
lF$BitMutationRate2()	0.20	xegaGaIVAdaptiveMutateGene()
lF$CutoffFit()	0.5	xegaGaIVADaptiveMutateGene()
lF$UCrossSwap()	0.2	xegaGaUPCross2Gene()
		xegaGaUPCrossGene()

In the xega-architecture, these rates can be configured to be adaptive.

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 into 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

License

MIT

URL

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

Installation

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

Author(s)

Andreas Geyer-Schulz

References

Goldberg, David E. (1989) Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading. (ISBN:0-201-15767-5)

See Also

Useful links:

- https://github.com/ageyerschulz/xegaGaGene

xegaGaGeneMap *Map the bit strings of a binary gene to parameters in an interval.*

Description

`xegaGaGeneMap()` maps the bit strings of a binary string to parameters in an interval. Bit vectors are mapped into equispaced numbers in the interval. Examples: Optimization of problems with real-valued parameter vectors.

Usage

```
xegaGaGeneMap(gene, penv)
```

Arguments

gene	A binary gene (the genotype).
penv	A problem environment.

Value

The decoded gene (the phenotype).

See Also

Other Decoder: `xegaGaDecodeGene()`, `xegaGaGeneMapGray()`, `xegaGaGeneMapIdentity()`, `xegaGaGeneMapPerm()`

Examples

```
gene<-xegaGaInitGene(lFxegaGaGene)
xegaGaGeneMap(gene$gene1, lFxegaGaGene$penv)
```

xegaGaGeneMapFactory *Configure the gene map function of a genetic algorithm.*

Description

`xegaGaGeneMapFactory()` implements the selection of one of the GeneMap 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. "Bin2Dec" returns `xegaGaGeneMap()`. (Default).
2. "Gray2Dec" returns `xegaGaGeneMapGray()`.
3. "Identity" returns `xegaGaGeneMapIdentity()`.
4. "Permutation" returns `xegaGaGeneMapPerm()`.

Usage

```
xegaGaGeneMapFactory(method = "Bin2Dec")
```

Arguments

method	A string specifying the GeneMap function.

Value

A gene map function for genes.

See Also

Other Configuration: lFxegaGaGene, xegaGaCrossoverFactory(), xegaGaMutationFactory(), xegaGaReplicationFactory()

Examples

```
XGene<-xegaGaGeneMapFactory("Identity")
gene<-xegaGaInitGene(lFxegaGaGene)
XGene(gene$gene1, lFxegaGaGene$penv)
```

---

xegaGaGeneMapGray *Map the bit strings of a gray-coded gene to parameters in an interval.*

Description

xegaGaGeneMapGray() maps the bit strings of a binary string interpreted as Gray codes to parameters in an interval. Bit vectors are mapped into equispaced numbers in the interval. Examples: Optimization of problems with real-valued parameter vectors.

Usage

```
xegaGaGeneMapGray(gene, penv)
```

Arguments

gene	A binary gene (the genotype).
penv	A problem environment.

Value

The decoded gene (the phenotype).

See Also

Other Decoder: xegaGaDecodeGene(), xegaGaGeneMap(), xegaGaGeneMapIdentity(), xegaGaGeneMapPerm()

Examples

```
gene<-xegaGaInitGene(lFxegaGaGene)
xegaGaGeneMapGray(gene$gene1, lFxegaGaGene$penv)
```

`xegaGaGeneMapIdentity` *Map the bit strings of a binary gene to an identical bit vector.*

Description

`xegaGaGeneMapIdentity()` maps the bit strings of a binary vector to an identical binary vector. Faster for all problems with single-bit coding. Examples: Knapsack, Number Partitioning into 2 partitions.

Usage

```
xegaGaGeneMapIdentity(gene, penv)
```

Arguments

`gene`	A binary gene (the genotype).
`penv`	A problem environment.

Value

A binary gene (the phenotype).

See Also

Other Decoder: `xegaGaDecodeGene()`, `xegaGaGeneMap()`, `xegaGaGeneMapGray()`, `xegaGaGeneMapPerm()`

Examples

```
gene<-xegaGaInitGene(lFxegaGaGene)
xegaGaGeneMapIdentity(gene$gene1, lFxegaGaGene$penv)
```

xegaGaGeneMapPerm *Map the bit strings of a binary gene to a permutation.*

Description

`xegaGaGeneMapPerm()` maps the bit strings of a binary string to a permutation of integers. Example: Traveling Salesman Problem (TSP).

Usage

```
xegaGaGeneMapPerm(gene, penv)
```

Arguments

gene	A binary gene (the genotype).
penv	A problem environment.

Value

A permutation (the decoded gene (the phenotype))

See Also

Other Decoder: `xegaGaDecodeGene()`, `xegaGaGeneMap()`, `xegaGaGeneMapGray()`, `xegaGaGeneMapIdentity()`

Examples

```
gene<-xegaGaInitGene(lFxegaGaGene)
xegaGaGeneMapPerm(gene$gene1, lFxegaGaGene$penv)
```

xegaGaInitGene *Generate a random binary gene.*

Description

`xegaGaInitGene()` generates a random binary gene with a given length.

Usage

```
xegaGaInitGene(lF)
```

Arguments

lF	The local configuration of the genetic algorithm.

Value

A binary gene (a named list):

- `$evaluated`: FALSE. See package `xegaSelectGene`.
- `$evalFail`: FALSE. Set by the error handler(s) of the evaluation functions in package `xegaSelectGene` in the case of failure.
- `$fit`: Fitness.
- `$gene1`: Binary gene.

Examples

```
xegaGaInitGene(lF$xegaGaGene)
```

---

xegaGaIVAdaptiveMutateGene

Individually variable adaptive mutation of a gene.

Description

`xegaGaIVAdaptiveMutateGene()` mutates a binary gene. Two mutation rates (`lF$BitMutationRate1()` and `lF$BitMutationRate2()` which is higher than the first) are used depending on the relative fitness of the gene. `lF$CutoffFit()` and `lF$CBestFitness()` are used to determine the relative fitness of the gene. The rationale is that mutating genes having a low fitness with a higher probability rate improves the performance of a genetic algorithm, because the gene gets a higher chance to improve.

Usage

```
xegaGaIVAdaptiveMutateGene(gene, lF)
```

Arguments

`gene`	A binary gene.
`lF`	The local configuration of the genetic algorithm.

Details

This principle is a candidate for a more abstract implementation, because it applies to all variants of evolutionary algorithms.

The goal is to separate the threshold code and the representation-dependent part and to combine them in the factory properly.

Value

A binary gene

References

Stanhope, Stephen A. and Daida, Jason M. (1996) An Individually Variable Mutation-rate Strategy for Genetic Algorithms. In: Koza, John (Ed.) Late Breaking Papers at the Genetic Programming 1996 Conference. Stanford University Bookstore, Stanford, pp. 177-185. (ISBN:0-18-201-031-7)

See Also

Other Mutation: `xegaGaMutateGene()`

Examples

```
parm<-function(x) {function() {return(x)}}
lFxegaGaGene$BitMutationRate1<-parm(1.0)
lFxegaGaGene$BitMutationRate2<-parm(0.5)
gene1<-xegaGaInitGene(lFxegaGaGene)
xegaGaDecodeGene(gene1, lFxegaGaGene)
gene<-xegaGaIVAdaptiveMutateGene(gene1, lFxegaGaGene)
xegaGaDecodeGene(gene, lFxegaGaGene)
```

---

xegaGaMutateGene *Mutate a gene.*

Description

`xegaGaMutateGene()` mutates a binary gene. The per-bit mutation rate is given by `lF$BitMutationRate1()`.

Usage

```
xegaGaMutateGene(gene, lF)
```

Arguments

`gene`	A binary gene.
`lF`	The local configuration of the genetic algorithm.

Value

A binary gene.

See Also

Other Mutation: `xegaGaIVAdaptiveMutateGene()`

Examples

```
parm<-function(x) {function() {return(x)}}
lFxegaGaGene$BitMutationRate1<-parm(1.0)
gene1<-xegaGaInitGene(lFxegaGaGene)
xegaGaDecodeGene(gene1, lFxegaGaGene)
lFxegaGaGene$BitMutationRate1()
gene<-xegaGaMutateGene(gene1, lFxegaGaGene)
xegaGaDecodeGene(gene, lFxegaGaGene)
```

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

Description

`xegaGaMutationFactory()` 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 `xegaGaMutateGene()`.
2. "IVM" returns `xegaGaIVAdaptiveMutateGene()`.

Usage

```
xegaGaMutationFactory(method = "MutateGene")
```

Arguments

`method`	A string specifying the mutation function.

Value

A mutation function for genes.

See Also

Other Configuration: `lFxegaGaGene`, `xegaGaCrossoverFactory()`, `xegaGaGeneMapFactory()`, `xegaGaReplicationFactory()`

Examples

```
parm<-function(x) {function() {return(x)}}
lFxegaGaGene$BitMutationRate1<-parm(1.0)
Mutate<-xegaGaMutationFactory("MutateGene")
gene1<-xegaGaInitGene(lFxegaGaGene)
gene1
Mutate(gene1, lFxegaGaGene)
```

xegaGaReplicate2Gene *Replicates a gene with a crossover operator with 2 kids.*

Description

xegaGaReplicate2Gene() replicates a gene by 2 random experiments which determine if a mutation operator (boolean variable mut) and/or a crossover operator (boolean variable cross should be applied. For each of the 4 cases, the appropriate code is executed and the genes are generated.

Usage

```
xegaGaReplicate2Gene(pop, fit, lF)
```

Arguments

pop	A population of binary genes.
fit	Fitness vector.
lF	The local configuration of the genetic algorithm.

Details

xegaGaReplicate2Gene() implements the control flow by case distinction which depends on the random choices for mutation and crossover:

1. A gene g is selected and the boolean variables mut and cross are set to runif(1)<rate. rate is given by lF$MutationRate() or lF$CrossRate().
2. The truth values of cross and mut determine the code that is executed:
   (a) (cross==TRUE) & (mut==TRUE): Mate selection, crossover, mutation.
   (b) (cross==TRUE) & (mut==FALSE): Mate selection, crossover.
   (c) (cross==FALSE) & (mut==TRUE): Mutation.
   (d) (cross==FALSE) & (mut==FALSE) is implicit: Returns a gene list.

Value

A list of either 1 or 2 binary genes.

See Also

Other Replication: xegaGaReplicate2GenePipeline(), xegaGaReplicateGene(), xegaGaReplicateGenePipeline()

Examples

```
lFxegaGaGene$CrossGene<-xegaGaCross2Gene
lFxegaGaGene$MutationRate<-function(fit, lF) {0.001}
names(lFxegaGaGene)
pop10<-lapply(rep(0,10), function(x) xegaGaInitGene(lFxegaGaGene))
epop10<-lapply(pop10, lFxegaGaGene$EvalGene, lF=lFxegaGaGene)
fit10<-unlist(lapply(epop10, function(x) {x$fit}))
newgenes<-xegaGaReplicate2Gene(pop10, fit10, lFxegaGaGene)
```

---

xegaGaReplicate2GenePipeline

Replicates a gene with a crossover operator with 2 kids by generating a list of function closures.

Description

`xegaGaReplicate2GenePipeline()` replicates a gene by 3 random experiments which determine if a mutation operator (boolean variable `mut1` and `mut2`) and/or a crossover operator (boolean variable `cross` should be applied. For each of the 8 cases, the appropriate pipeline constructor is executed and the genetic operator pipeline(s) is (are) returned.

Usage

```
xegaGaReplicate2GenePipeline(pop, fit, lF)
```

Arguments

`pop`	A population of binary genes.
`fit`	Fitness vector.
`lF`	The local configuration of the genetic algorithm.

Details

`xegaGaReplicate2GenePipeline()` implements the control flow by case distinction which depends on the random choices for mutation and crossover. The pipeline constructor chosen returns the function closure with the appropriate genetic operator pipeline.

1. A gene g is selected and the boolean variables `mut1`, `mut2`, and `cross` are set to `runif(1)<rate`. `rate` is given by `lF$MutationRate()` or `lF$CrossRate()`.
2. The truth values of `cross`, `mut1`, and `mut2` determine the genetic operator pipeline constructor that is executed:
   (a) `(cross==FALSE) & (mut1==FALSE)` is implicit: Executes the pipeline constructor `newPipeline`.
   (b) `(cross==TRUE) & (mut1==TRUE) & (mut2==TRUE)`: Crossover, mutation on both kids. Executes the genetic operator pipeline constructor `newCrossMut2Pipeline`.
   (c) `(cross==TRUE) & (mut1==TRUE) & (mut2==FALSE)`: Crossover, mutation on first kid. Executes the genetic operator pipeline constructor `newCross2Mut1Pipeline`.

(d) `(cross==TRUE) & (mut1==FALSE) & (mut2==TRUE)`: Crossover, mutation on second kid. Executes the genetic operator pipeline constructor `newCross2Mut2Pipeline`.

(e) `(cross==TRUE) & (mut1==FALSE) & (mut2==FALSE)`: Crossover (2 kids). Executes the genetic operator pipeline constructor `newCross2Pipeline`.

(f) `(cross==FALSE) & (mut1==TRUE)`: Mutation. Executes the genetic operator pipeline constructor `newMutPipeline`.

Value

A list of either 1 or 2 function closures with the operator pipeline.

See Also

Other Replication: `xegaGaReplicate2Gene()`, `xegaGaReplicateGene()`, `xegaGaReplicateGenePipeline()`

Examples

```
lFxegaGaGene$CrossGene<-xegaGaCross2Gene
lFxegaGaGene$MutationRate<-function(fit, lF) {0.001}
names(lFxegaGaGene)
pop10<-lapply(rep(0,10), function(x) xegaGaInitGene(lFxegaGaGene))
epop10<-lapply(pop10, lFxegaGaGene$EvalGene, lF=lFxegaGaGene)
fit10<-unlist(lapply(epop10, function(x) {x$fit}))
newgenes<-xegaGaReplicate2GenePipeline(pop10, fit10, lFxegaGaGene)
```

---

xegaGaReplicateGene *Replicates a gene with a crossover operator which returns a single gene.*

Description

`xegaGaReplicateGene()` replicates a gene by applying a gene reproduction pipeline which uses crossover and mutation and finishes with an acceptance rule. The control flow starts by selecting a gene from the population followed by the case distinction:

- Check if the mutation operation should be applied. (`mut` is `TRUE` with a probability of `lF$MutationRate()`).
- Check if the crossover operation should be applied. (`cross` is `TRUE` with a probability of `lF$CrossRate()`).

The state distinction determines which genetic operations are performed.

Usage

```
xegaGaReplicateGene(pop, fit, lF)
```

Arguments

pop	Population of binary genes.
fit	Fitness vector.
lF	Local configuration of the genetic algorithm.

Details

xegaGaReplicateGene() implements the control flow by a dynamic definition of the operator pipeline depending on the random choices for mutation and crossover:

1. A gene g is selected and the boolean variables mut and cross are set to runif(1)<rate.
2. The local function for the operator pipeline OPpip(g, lF) is defined by the truth values of cross and mut:
   (a) (cross==FALSE) & (mut==FALSE): Identity function.
   (b) (cross==TRUE) & (mut==TRUE): Mate selection, crossover, mutation.
   (c) (cross==TRUE) & (mut==FALSE): Mate selection, crossover.
   (d) (cross==FALSE) & (mut==TRUE): Mutation.
3. Perform the operator pipeline and accept the result. The acceptance step allows the combination of a genetic algorithm with other heuristic algorithms like simulated annealing by executing an acceptance rule. For the genetic algorithm, the identity function is used.

Value

A list of one gene.

See Also

Other Replication: xegaGaReplicate2Gene(), xegaGaReplicate2GenePipeline(), xegaGaReplicateGenePipeline()

Examples

```
lFxegaGaGene$CrossGene<-xegaGaCrossGene
lFxegaGaGene$MutationRate<-function(fit, lF) {0.001}
lFxegaGaGene$Accept<-function(OperatorPipeline, gene, lF) {gene}
pop10<-lapply(rep(0,10), function(x) xegaGaInitGene(lFxegaGaGene))
epop10<-lapply(pop10, lFxegaGaGene$EvalGene, lF=lFxegaGaGene)
fit10<-unlist(lapply(epop10, function(x) {x$fit}))
newgenes<-xegaGaReplicateGene(pop10, fit10, lFxegaGaGene)
```

`xegaGaReplicateGenePipeline` *Replicates a gene by generating a pipeline with a crossover operator returning a single kid.*

Description

`xegaGaReplicateGenePipeline()` returns a gene reproduction pipeline which is represented as a closure with crossover and mutation and an acceptance rule together with the necessary genes. The control flow starts by selecting a gene from the population followed by the case distinction:

- Check if the mutation operation should be applied. (`mut` is `TRUE` with a probability of `lF$MutationRate()`).
- Check if the crossover operation should be applied. (`cross` is `TRUE` with a probability of `lF$CrossRate()`).

The state distinction determines which genetic operator pipeline is returned.

Usage

```
xegaGaReplicateGenePipeline(pop, fit, lF)
```

Arguments

`pop`	Population of binary genes.
`fit`	Fitness vector.
`lF`	Local configuration of the genetic algorithm.

Details

`xegaGaReplicateGenePipeline()` returns an operator pipeline with the steps crossover, mutate, accept, and evaluate. generated by a pipeline constructor depending on the random choices for mutation and crossover:

1. The genes `g`, `g1` are selected and the boolean variables `mut` and `cross` are set to `runif(1)<rate`.
2. The local function for the operator pipeline `OPpip(g, lF)` is generated by the pipeline constructor selected by the truth values of `cross` and `mut`:
   (a) `(cross==FALSE) & (mut==FALSE)`: Pipeline constructor `newPipeline(g, lF)`.
   (b) `(cross==TRUE) & (mut==TRUE)`: Pipeline constructor `newCrossMutPipeline(g, g1, lF)`.
   (c) `(cross==TRUE) & (mut==FALSE)`: Pipeline constructor `newCrossPipeline(g, g1, lF)`.
   (d) `(cross==FALSE) & (mut==TRUE)`: Pipeline constructor `newMutPipeline(g, lF)`.

Value

A list of a function closure with the operator pipeline.

See Also

Other Replication: xegaGaReplicate2Gene(), xegaGaReplicate2GenePipeline(), xegaGaReplicateGene()

Examples

```
lFxegaGaGene$CrossGene<-xegaGaCrossGene
lFxegaGaGene$MutationRate<-function(fit, lF) {0.5}
lFxegaGaGene$CrossRate<-function(fit, lF) {0.5}
lFxegaGaGene$Accept<-function(OperatorPipeline, gene, lF) {gene}
pop10<-lapply(rep(0,10), function(x) xegaGaInitGene(lFxegaGaGene))
epop10<-lapply(pop10, lFxegaGaGene$EvalGene, lF=lFxegaGaGene)
fit10<-unlist(lapply(epop10, function(x) {x$fit}))
newgenes<-xegaGaReplicateGenePipeline(pop10, fit10, lFxegaGaGene)
```

---

xegaGaReplicationFactory

Configure the replication function of a genetic algorithm.

Description

xegaGaReplicationFactory() implements the selection of a replication method.

Current support:

1. "Kid1" returns xegaGaReplicateGene().
2. "Kid1Pipeline" returns xegaGaReplicateGenePipeline().
3. "Kid2" returns xegaGaReplicate2Gene().
4. "Kid2Pipeline" returns xegaGaReplicate2GenePipeline().

Usage

```
xegaGaReplicationFactory(method = "Kid1")
```

Arguments

method	A string specifying the replication function.

Value

A replication function for genes.

See Also

Other Configuration: lFxegaGaGene, xegaGaCrossoverFactory(), xegaGaGeneMapFactory(), xegaGaMutationFactory()

Examples

```
lFxegaGaGene$CrossGene<-xegaGaCrossGene
lFxegaGaGene$MutationRate<-function(fit, lF) {0.001}
lFxegaGaGene$Accept<-function(OperatorPipeline, gene, lF) {gene}
Replicate<-xegaGaReplicationFactory("Kid1")
pop10<-lapply(rep(0,10), function(x) xegaGaInitGene(lFxegaGaGene))
epop10<-lapply(pop10, lFxegaGaGene$EvalGene, lF=lFxegaGaGene)
fit10<-unlist(lapply(epop10, function(x) {x$fit}))
newgenes1<-Replicate(pop10, fit10, lFxegaGaGene)
lFxegaGaGene$CrossGene<-xegaGaCross2Gene
Replicate<-xegaGaReplicationFactory("Kid2")
newgenes2<-Replicate(pop10, fit10, lFxegaGaGene)
```

---

xegaGaUCross2Gene *Uniform crossover of 2 genes.*

Description

`xegaGaUCross2Gene()` swaps alleles of both genes with a probability of 0.5. It generates a random mask which is used to build the new genes. It returns 2 genes.

Usage

```
xegaGaUCross2Gene(gg1, gg2, lF)
```

Arguments

`gg1`	A binary gene.
`gg2`	A binary gene.
`lF`	The local configuration of the genetic algorithm.

Value

A list of 2 binary genes.

References

Syswerda, Gilbert (1989): Uniform Crossover in Genetic Algorithms. In: Schaffer, J. David (Ed.) Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Los Altos, California, pp. 2-9. (ISBN:1-55860-066-3)

See Also

Other Crossover (Returns 2 Kids): `xegaGaCross2Gene()`, `xegaGaUPCross2Gene()`

Examples

```
gene1<-xegaGaInitGene(lFxegaGaGene)
gene2<-xegaGaInitGene(lFxegaGaGene)
xegaGaDecodeGene(gene1, lFxegaGaGene)
xegaGaDecodeGene(gene2, lFxegaGaGene)
newgenes<-xegaGaUCross2Gene(gene1, gene2, lFxegaGaGene)
xegaGaDecodeGene(newgenes[[1]], lFxegaGaGene)
xegaGaDecodeGene(newgenes[[2]], lFxegaGaGene)
```

---

xegaGaUCrossGene *Uniform crossover of 2 genes.*

Description

`xegaGaUCrossGene()` swaps alleles of both genes with a probability of 0.5. It generates a random mask which is used to build the new gene.

Usage

```
xegaGaUCrossGene(gg1, gg2, lF)
```

Arguments

`gg1`	A binary gene.
`gg2`	A binary gene.
`lF`	The local configuration of the genetic algorithm.

Value

A list of one binary gene.

References

Syswerda, Gilbert (1989): Uniform Crossover in Genetic Algorithms. In: Schaffer, J. David (Ed.) Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Los Altos, California, pp. 2-9. (ISBN:1-55860-066-3)

See Also

Other Crossover (Returns 1 Kid): `xegaGaCrossGene()`, `xegaGaUPCrossGene()`

Examples

```
gene1<-xegaGaInitGene(lFxegaGaGene)
gene2<-xegaGaInitGene(lFxegaGaGene)
xegaGaDecodeGene(gene1, lFxegaGaGene)
xegaGaDecodeGene(gene2, lFxegaGaGene)
gene3<-xegaGaUCrossGene(gene1, gene2, lFxegaGaGene)
xegaGaDecodeGene(gene3[[1]], lFxegaGaGene)
```

xegaGaUPCross2Gene *Parameterized uniform crossover of 2 genes.*

Description

xegaGaUP2CrossGene() swaps alleles of both genes with a probability of lF$UCrossSwap(). It generates a random mask which is used to build the new gene. It returns 2 genes.

Usage

```
xegaGaUPCross2Gene(gg1, gg2, lF)
```

Arguments

gg1	A binary gene.
gg2	A binary gene.
lF	The local configuration of the genetic algorithm.

Value

A list of 2 binary genes.

References

Spears William and De Jong, Kenneth (1991): On the Virtues of Parametrized Uniform Crossover. In: Belew, Richar K. and Booker, Lashon B. (Ed.) Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Los Altos, California, pp. 230-236. (ISBN:1-55860-208-9)

See Also

Other Crossover (Returns 2 Kids): xegaGaCross2Gene(), xegaGaUCross2Gene()

Examples

```
gene1<-xegaGaInitGene(lFxegaGaGene)
gene2<-xegaGaInitGene(lFxegaGaGene)
xegaGaDecodeGene(gene1, lFxegaGaGene)
xegaGaDecodeGene(gene2, lFxegaGaGene)
newgenes<-xegaGaUPCross2Gene(gene1, gene2, lFxegaGaGene)
xegaGaDecodeGene(newgenes[[1]], lFxegaGaGene)
xegaGaDecodeGene(newgenes[[2]], lFxegaGaGene)
```

xegaGaUPCrossGene *Parameterized uniform crossover of 2 genes.*

Description

xegaGaUPCrossGene() swaps alleles of both genes with a probability of lF$UCrossSwap(). It generates a random mask which is used to build the new gene.

Usage

```
xegaGaUPCrossGene(gg1, gg2, lF)
```

Arguments

gg1	A binary gene.
gg2	A binary gene.
lF	The local configuration of the genetic algorithm.

Value

A list of one binary gene.

References

Spears William and De Jong, Kenneth (1991): On the Virtues of Parametrized Uniform Crossover. In: Belew, Richar K. and Booker, Lashon B. (Ed.) Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Los Altos, California, pp. 230-236. (ISBN:1-55860-208-9)

See Also

Other Crossover (Returns 1 Kid): xegaGaCrossGene(), xegaGaUCrossGene()

Examples

```
gene1<-xegaGaInitGene(lFxegaGaGene)
gene2<-xegaGaInitGene(lFxegaGaGene)
xegaGaDecodeGene(gene1, lFxegaGaGene)
xegaGaDecodeGene(gene2, lFxegaGaGene)
gene3<-xegaGaUPCrossGene(gene1, gene2, lFxegaGaGene)
xegaGaDecodeGene(gene3[[1]], lFxegaGaGene)
```

Index