Package 'xega'

March 20, 2024

Title Extended Evolutionary and Genetic Algorithms

Version 0.9.0.0

Description Implementation of a scalable, highly configurable, and e(x)tended architecture for (e)volutionary and (g)enetic (a)lgorithms. Multiple representations (binary, real-coded, permutation, and derivation-tree), a rich collection of genetic operators, as well as an extended processing pipeline are provided for genetic algorithms (Goldberg, D. E. (1989, ISBN:0-201-15767-5)), differential evolution (Price, Kenneth V., Storn, Rainer M. and Lampinen, Jouni A. (2005) <doi:10.1007/3-540-31306-0>), simulated annealing (Aarts, E., and Korst, J. (1989, ISBN:0-471-92146-7)), grammar-based genetic programming (Geyer-Schulz (1997, ISBN:978-3-7908-0830-X)), and grammatical evolution (Ryan, C., O'Neill, M., and Collins, J. J. (2018) <doi:10.1007/978-3-319-78717-6>). All algorithms reuse basic adaptive mechanisms for performance optimization. Sequential or parallel execution (on multi-core machines, local clusters, and high performance computing environments) is available for all algorithms. See <https://github.com/ageyerschulz/xega/tree/main/examples/executionModel>.

License MIT + file LICENSE

URL <https://github.com/ageyerschulz/xega>

Encoding UTF-8

RoxygenNote 7.2.3

Depends R (>= 3.5.0), parallelly

Imports xegaSelectGene, xegaBNF, xegaDerivationTrees, xegaGaGene, xegaGpGene, xegaGeGene, xegaDfGene, xegaPermGene, xegaPopulation

Suggests testthat (>= 3.0.0)

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 2024-03-20 09:00:02 UTC

R topics documented:

`booleanGrammar` *A constant function with a boolean grammar.*

Description

For the distribution of examples of BNF in grammars.

Usage

```
booleanGrammar()
```

Details

Imported from package xegaBNF for use in examples.

Value

A named list with $filename and $BNF, the grammar of a boolean grammar with two variables.

See Also

Other Grammar: `compileBNF()`

Examples

```
booleanGrammar()
```

compileBNF *Compile a BNF.*

Description

compileBNF() produces a context-free grammar from its specification of in Backus-Naur form (BNF). Warning: No error checking implemented.

Usage

```
compileBNF(g, verbose = FALSE)
```

Arguments

g	A character string with a BNF.
verbose	Boolean. TRUE: Show progress. Default: FALSE.

Details

A grammar consists of the symbol table ST, the production table PT, the start symbol Start, and the short production table SPT. An example BNF is provided by booleanGrammar().

The function performs the following steps:

1. Make the symbol table.
2. Make the production table.
3. Extract the start symbol.
4. Compile a short production table.
5. Return the grammar.

For a full documentation, see <https://CRAN.R-project.org/package=xegaBNF>

Value

A grammar object (list) with the attributes name (the filename of the grammar), ST (symbol table), PT (production table), Start (the start symbol of the grammar), and SPT (the short production table).

See Also

Other Grammar: booleanGrammar()

Examples

```
g<-compileBNF(booleanGrammar())
g$ST
g$PT
g$Start
g$SPT
```

lau15 *The problem environment lau15*

Description

15 abstract cities for which a traveling salesman solution is sought. Solution: A path with a length of 291.

Usage

```
lau15
```

Format

An object of class `list` of length 12.

References

Lau, H. T. (1986): *Combinatorial Heuristic Algorithms in FORTRAN*. Springer, 1986. <doi:10.1007/978-3-642-61649-5>

See Also

Other Problem Environment: `NewEnvXOR()`, `Parabola2DEarly`, `Parabola2D`

Examples

```
names(lau15)
lau15$genelength()
```

NewEnvXOR *Generate the problem environment EnvXOR*

Description

`NewEnvXOR()` generates the problem environment for the XOR-Problem.

The problem environment provides an abstract interface to the simple genetic programming algorithm. `ProblemEnv$f(parm)` defines the function we want to optimize.

A problem environment is a function factory with the following elements:

1. `name()` a string with the name of the environment
2. `ProblemEnv$f(word)` function with word a word of the language (as text string).

Should be provided by the user as standard R-file.

Usage

```
NewEnvXOR()
```

Value

The problem environment:

- `$name`: The name of the problem environment.
- `$f`: The fitness function. For this environment number of correct test cases (correct function) and the inverse of the number of terminal symbols (boolean function with small number of elements).

See Also

Other Problem Environment: `Parabola2DEarly`, `Parabola2D`, `lau15`

Examples

```
EnvXOR<-NewEnvXOR()
EnvXOR$name()
a2<-"OR(OR(D1, D2), (AND(NOT(D1), NOT(D2))))"
a3<-"OR(OR(D1, D2), AND(D1, D2))"
a4<-"AND(OR(D1,D2),NOT(AND(D1,D2)))"
gp4<-"(AND(AND(OR(D2,D1),NOT(AND(D1,D2))),(OR(D2,D1))))"
EnvXOR$f(a2)
EnvXOR$f(a3)
EnvXOR$f(a4)
EnvXOR$f(gp4)
```

Parabola2D *Problem environment for a 2-dimensional quadratic parabola*

Description

Problem environment for finding maxima and minima of a 2-dimensional quadratic parabola.

Usage

```
Parabola2D
```

Format

An object of class `list` of length 8.

Value

A named list

- `$name()`: Returns the name of the problem environment.
- `$bitlength()`: The vector of the bitlengths of the parameters.
- `$genelength()`: The number of bits of a gene.
- `$lb()`: The vector of lower bounds of the parameters.
- `$ub()`: The vector of upper bounds of the parameters.
- `$f(parm)`: The implementation of the function of the quadratic parabola.
  - `parm`: A 2-element vector of reals.
  - Returns the value of the function.
- `$describe()`: Returns the description of the problem environment.
- `$solution()`: The solutions (maxima/minima) of the problem environment (if known).

See Also

Other Problem Environment: `NewEnvXOR()`, `Parabola2DEarly`, `lau15`

Examples

```
names(Parabola2D)
Parabola2D$name()
Parabola2D$describe()
Parabola2D$bitlength()
Parabola2D$genelength()
Parabola2D$lb()
Parabola2D$ub()
Parabola2D$f
Parabola2D$f(c(2.2, -1.37))
Parabola2D$solution()
Parabola2D$solution()$minimum
Parabola2D$solution()$minpoints
Parabola2D$solution()$maximum
Parabola2D$solution()$maxpoints
```

Parabola2DEarly *Problem environment for a 2-dimensional quadratic parabola.*

Description

An example of a problem environment with an early termination condition.

Usage

```
Parabola2DEarly
```

Format

An object of class `list` of length 9.

Value

A problem environment (see Parabola2D). `Parabola2DEarly$terminate(solution, lF)` is a test function which returns true if the `solution` is in an epsilon environment of a known solution. To invoke this function, use `Run( ..., early=TRUE, ...)`. The epsilon which determines the length of the interval as a fraction of the known optimal solution is configured by e.g. `Run( ..., terminationEps=0.001, ...)`.

See Also

Other Problem Environment: `NewEnvXOR()`, `Parabola2D`, `lau15`

`sgXCrossoverFactory` *Factory for configuring a gene dependent Crossover function.*

Description

`sgXCrossoverFactory()` selects

1. the algorithm specific crossover factory and
2. the method in this factory.

Usage

```
sgXCrossoverFactory(algorithm = "sga", method = "CrossGene")
```

Arguments

`algorithm`	Specifies algorithm. Available: "sga", "sgde", "sgperm", "sge", sgp". Default: "sga".
`method`	Crossover method. Algorithm (gene representation) dependent. Default: `Crossgene()`. Must be available in the gene specific crossover factories.

Details

The available methods for each algorithm are:

- "sga": "Cross2Gene", "UCross2Gene", "UPCross2Gene", "CrossGene", "UCrossGene", "UP-CrossGene".
- "sge": "Cross2Gene", "UCross2Gene", "UPCross2Gene", "CrossGene", "UCrossGene", "UP-CrossGene".
- "sgp": "CrossGene", "Cross2Gene".
- "sgde": "CrossGene", "UCrossGene", "UPCrossGene".
- "sgperm": "CrossGene", "Cross2Gene".

Value

Crossover Crossover function from the crossover factory of the selected package.

See Also

Other Configuration: sgXDecodeGeneFactory(), sgXGeneMapFactory(), sgXInitGeneFactory(), sgXMutationFactory(), sgXReplicationFactory()

Examples

```
sgXCrossoverFactory(algorithm="sga", method="CrossGene")
```

---

sgXDecodeGeneFactory *Factory for configuring a gene dependent DecodeGene function.*

Description

A gene specific decoder must be provided.

Usage

```
sgXDecodeGeneFactory(algorithm = "sga", method = "DecodeGene")
```

Arguments

algorithm	"sga", "sgde", "sgperm", "sge", sgp". Default: "sga".
method	Method. Default: "DecodeGene".

Value

Decode function for the selected algorithm from the correct package.

See Also

Other Configuration: sgXCrossoverFactory(), sgXGeneMapFactory(), sgXInitGeneFactory(), sgXMutationFactory(), sgXReplicationFactory()

Examples

```
sgXDecodeGeneFactory(algorithm="sgperm", method="DecodeGene")
```

sgXGeneMapFactory *Factory for configuring a gene dependent geneMap function.*

Description

The geneMap function depends on the gene representation and the algorithm selected.

Usage

```
sgXGeneMapFactory(algorithm = "sga", method = "Bin2Dec")
```

Arguments

algorithm	Algorithm. Available: "sga", "sgde", "sgperm", "sge", sgp". Default: "sga".
method	The GeneMap method. The choices depend on the algorithm.

Details

Methods available for the different algorithms are:

- "sga": "Bin2Dec", "Gray2Dec", "Identity", "Permutation".
- "sgde": "Identity".
- "sgperm": "Identity". The gene map function is not used in the decoder.
- "sgp": "Identity". The gene map function is not used in the decoder.
- "sge": "Mod" or "Bucket".

Value

GeneMap function for the selected algorithm from the correct package.

See Also

Other Configuration: sgXCrossoverFactory(), sgXDecodeGeneFactory(), sgXInitGeneFactory(), sgXMutationFactory(), sgXReplicationFactory()

Examples

```
sgXGeneMapFactory(algorithm="sga", method="Bin2Dec")
```

sgXInitGeneFactory *Factory for configuring a gene dependent InitGene function.*

Description

Factory for configuring a gene dependent InitGene function.

Usage

```
sgXInitGeneFactory(algorithm = "sga", method = "InitGene")
```

Arguments

algorithm	Algorithm. Available: "sga", "sgde", "sgperm", "sge", sgp". Default: "sga".
method	Method. Default: "InitGene".

Value

InitGene function from the correct package.

See Also

Other Configuration: sgXCrossoverFactory(), sgXDecodeGeneFactory(), sgXGeneMapFactory(), sgXMutationFactory(), sgXReplicationFactory()

Examples

```
sgXInitGeneFactory(algorithm="sgperm")
```

sgXMutationFactory *Factory for configuring a gene dependent Mutation function.*

Description

sgXMutationFactory() selects

1. the algorithm specific crossover factory and
2. the method in this factory.

Usage

```
sgXMutationFactory(algorithm = "sga", method = "MutateGene")
```

Arguments

algorithm	Algorithm. Available: "sga", "sgde", "sgperm", "sge", sgp". Default: "sga".
method	Method. Available methods are package-dependent.

Details

The available methods for each factory are:

- "sga": "MutateGene", "IVM".
- "sge": "MutateGene", "IVM".
- "sgp": "MutateGene".
- "sgde": "MutateGeneDE".
- "sgperm": "MutateGeneOrderBased", "MutateGenekInversion", "MutateGene2Opt", "MutateGenekOptLK", "MutateGeneGreedy", "MutateGeneBestGreedy", "MutateGeneMix".

Value

MutateGene Function for the selected algorithm from the correct package.

See Also

Other Configuration: sgXCrossoverFactory(), sgXDecodeGeneFactory(), sgXGeneMapFactory(), sgXInitGeneFactory(), sgXReplicationFactory()

Examples

```
sgXMutationFactory(algorithm="sga", method="MutateGene")
```

---

sgXReplicationFactory *Factory for configuring a gene dependent Replication function.*

Description

Factory for configuring a gene dependent Replication function.

Usage

```
sgXReplicationFactory(algorithm = "sga", method = "Kid1")
```

Arguments

algorithm	Algorithm. Available: "sga", "sgde", "sgperm", "sge", sgp". Default: "sga".
method	Method. Options are package-dependent: • "sga", "sgperm", "sge", sgp": "Kid1", "Kid2". • "sgde": "DE".

Value

A replication function for the algorithm from the correct package.

See Also

Other Configuration: `sgXCrossoverFactory()`, `sgXDecodeGeneFactory()`, `sgXGeneMapFactory()`, `sgXInitGeneFactory()`, `sgXMutationFactory()`

Examples

```
sgXReplicationFactory(algorithm="sgp", method="Kid1")
```

`xega` *Package xega*

Description

The main program of the e(x)tended (e)volutionary and (g)enetic (a)lgorithm (xega) package.

Layers (in top-down direction)

1. **Top-level main programs** (Package `xega`): `RunGA()`, `ReRun()`
2. **Population level operations - independent of representation** (Package `xegaPopulation`): The population layer consists of functions for initializing, logging, observing, evaluating a population of genes, as well as of computing the next population.
3. **Gene level operations - representation-dependent**.
   (a) **Binary representation** (Package `xegaGaGene`): 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.
   (b) **Real-coded genes** (Package `xegaDfGene`).
   (c) **Permutation genes** (Package `xegaPermGene`).
   (d) **Derivation-tree genes** (Package `xegaGpGene`).
   (e) **Binary genes with a grammar-driven decoder** (Package `xegaGeGene`).
4. **Gene level operations - independent of representation** (Package `selectGene`): Functions for static and adaptive fitness scaling, gene selection, and gene evaluation as well as for the measurement of performance and for configuration.

Early Termination

A problem environment may implement a function `terminate(solution)` which returns TRUE if the `solution` meets a condition for early termination.

Parallel and Distributed Execution

Parallel and distributed execution is supported for several combinations of hard- and software architectures by overloading the `lapply()`-function used in the evaluation of a fitness function for a population of genes with a parallel version with the abstract interface:

```
parallelApply(pop, EvalGene, lF)
```

where `pop` is a list of genes, `EvalGene` the evaluation function for the fitness of a gene, and `lF` the local function configuration of the algorithm.

The several implementations of a `parallelApply()` function are provided. The implementations use

- the function `parallel::mclapply()` for multicore parallelization by the fork mechanism of Unix-based operating systems on a single machine.
- the function `parallel::parLapply()` for socket connections on a single or multiple machines on the Internet.
- the function `future.apply::future_lapply()` for asynchronous parallelization based on future packages.

In addition, user-defined parallel apply functions can be provided. Example scripts for using the `Rmpi::mpi.parLapply()` function of the `Rmpi` package are provided for a HPC environment with Slurm as well as on a notebook.

The Architecture of the xegaX-Packages

The xegaX-packages are a family of R-packages which implement e(x)tended (e)volutionary and (g)enetic (a)lgorithms (xega). The architecture has 3 layers, namely the user interface layer, the population layer, and the gene layer:

- The user interface layer (package `xega` <https://CRAN.R-project.org/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` <https://CRAN.R-project.org/package=xegaPopulation>) contains population related functionality as well as support for population statistics dependent adaptive mechanisms and for parallelization.
- The gene layer is split in a representation independent and a representation dependent part:
  1. The representation indendent part (package `xegaSelectGene` <https://CRAN.R-project.org/package=xegaSelectGe) 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` <https://CRAN.R-project.org/package=xegaGaGene> for binary coded genetic algorithms.
     - `xegaPermGene` <https://CRAN.R-project.org/package=xegaPermGene> for permutation-based genetic algorithms.
     - `xegaDfGene` <https://CRAN.R-project.org/package=xegaDfGene> for derivation free algorithms as e.g. differential evolution.

- xegaGpGene <https://CRAN.R-project.org/package=xegaGpGene> for grammar-based genetic algorithms.
- xegaGeGene <https://CRAN.R-project.org/package=xegaGaGene> for grammatical evolution algorithms.

The packages xegaDerivationTrees and xegaBNF support the packages xegaGpGene and xegaGeGene:

- xegaBNF <https://CRAN.R-project.org/package=xegaBNF> essentially provides a grammar compiler and
- xegaDerivationTrees <https://CRAN.R-project.org/package=xegaDerivationTrees> an abstract data type for derivation trees.

Copyright

(c) 2023 Andreas Geyer-Schulz

License

MIT

URL

https://github.com/ageyerschulz/xega

Installation

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

Author(s)

Andreas Geyer-Schulz

---

xegaReRun *Run an evolutionary or genetic algorithm with the same configuration as in the previous run.*

Description

`xegaReRun()` runs a simple genetic algorithm with the same configuration as in the run specified by the list element `$GAconfig` of the solution of a simple genetic algorithm.

Usage

```
xegaReRun(solution)
```

Arguments

`solution`	The solution of a previous run of `xegaRun()`.

Details

xegaReRun() does not capture the configuration for parallel/distributed processing for the execution model "FutureApply", because the user defines the configuration before calling xegaRun().

If executionModel matches neither "Sequential" nor "MultiCore" or !is.null(uParApply)==TRUE, a warning is printed, and the previous solution is returned.

Value

A list of

1. $popStat: A matrix with mean, min, Q1, median, Q3, max, var, mad of population fitness as columns: i-th row for i-th each generation.
2. $solution: With fields $solution$fitness, $solution$value, $solution$genotype, and
3. $GAconfig: The configuration of the GA used by xegaReRun().
4. $GAenv: Attribute value list of GAconfig.
5. $timer: An attribute value list with the time used (in seconds) in the main blocks of the GA: tUsed, tInit, tNext, tEval, tObserve, and tSummary.

See Also

Other Main Program: xegaRun()

Examples

```
a<-xegaRun(Parabola2D, max=FALSE, algorithm="sga", generations=10, popsize=20, verbose=1)
b<-xegaReRun(a)
seqApply<-function(pop, EvalGene, lF) {lapply(pop, EvalGene, lF)}
c<-xegaRun(Parabola2D, max=FALSE, algorithm="sga", uParApply=seqApply)
b<-xegaReRun(c)
```

---

xegaRun — *Run an evolutionary or genetic algorithm for a problem environment which contains a function to optimize.*

Description

Run runs an evolutionary or genetic algorithm whose type is selected by algorithm. Available algorithms are:

1. "sga": Genetic algorithm with binary genes.
2. "sgde": Differential evolution with real genes.
3. "sgperm": Genetic algorithm with permutation genes.
4. "sgp": Grammar-based genetic programming with derivation-tree genes.
5. "sge": Grammatical evolution (genetic algorithm with binary genes and a grammar-driven decoder.

The choice of the algorithm determines the gene-dependent configuration options.

Usage

```
xegaRun(
  penv,
  grammar = NULL,
  max = TRUE,
  algorithm = "sga",
  popsize = 100,
  generations = 20,
  crossrate = 0.2,
  mutrate = 1,
  elitist = TRUE,
  replay = 0,
  maxdepth = 7,
  maxtrials = 5,
  codons = 25,
  codonBits = 0,
  codonPrecision = "LCM",
  maxPBias = 0.01,
  evalmethod = "EvalGeneU",
  reportEvalErrors = TRUE,
  genemap = "Bin2Dec",
  crossrate2 = 0.3,
  ivcrossrate = "Const",
  crossover = "Cross2Gene",
  uCrossSwap = 0.2,
  mincrossdepth = 1,
  maxcrossdepth = 7,
  ivmutrate = "Const",
  mutrate2 = 1,
  bitmutrate = 0.005,
  bitmutrate2 = 0.01,
  maxmutdepth = 3,
  minmutinsertiondepth = 1,
  maxmutinsertiondepth = 7,
  lambda = 0.05,
  max2opt = 100,
  scalefactor1 = 0.9,
  scalefactor2 = 0.3,
  scalefactor = "Const",
  cutoffFit = 0.5,
  mutation = "MutateGene",
  replication = "Kid2",
  offset = 1,
  eps = 0.01,
  tournamentSize = 2,
  selectionBias = 1.5,
  maxTSR = 1.5,
  selection = "SUS",
```

```
  mateselection = "SUS",
  selectionContinuation = TRUE,
  scaling = "NoScaling",
  scalingThreshold = 0,
  scalingExp = 1,
  scalingExp2 = 1,
  rdmWeight = 1,
  drMax = 2,
  drMin = 0.5,
  dispersionMeasure = "var",
  scalingDelay = 1,
  accept = "All",
  alpha = 0.99,
  beta = 2,
  cooling = "ExponentialMultiplicative",
  coolingPower = 1,
  temp0 = 40,
  tempN = 0.01,
  verbose = 1,
  logevals = FALSE,
  allsolutions = FALSE,
  early = FALSE,
  terminationEps = 0.01,
  cores = NA,
  executionModel = "Sequential",
  uParApply = NULL,
  Cluster = NULL,
  profile = FALSE,
  batch = FALSE,
  path = ""
)
```

Arguments

penv	Problem environment.
grammar	A compiled grammar object. Default: NULL. Example: `compileBNF(booleanGrammar())`
max	If `TRUE` then Maximize! Default: TRUE. Used in functions `EvalGeneDet`, `EvalGeneStoch`, `EvalGeneU`, and `EvalGeneR` of package `xegaSelectGene`.
algorithm	Specifies the algorithm class dependend on gene representation: • "sga": Binary representation (Default). • "sgde": Real representation. E.g. Differential evolution. • "sgperm": Permutation representation. • "sge": Binary representation. Grammatical evolution. (Not yet variable length.) • "sgp": Derivation tree representation. Grammar Based Genetic Programming.
popsize	Population size. Default: 100.

`generations` Number of generations. Default: 20.

`crossrate` Probability of applying crossover operator. Default: 0.20. (Global parameter)

`mutrate` Probability of applying mutation operator. Default: 1.0. (Global parameter)

`elitist` Boolean. If `TRUE`, then keep best solution in population. Default: `TRUE`.

`replay` Integer. If `replay>0` then use `replay` as seed of random number generator and store it for exact repetition of run. Default: 0.

`maxdepth` The maximal depth of a derivation tree. Default: 7. (`"sgp"`).

`maxtrials` Maximal number of trials of finding subtrees with same root symbol. Default: 5. (`sgp`).

`codons` The maximal number of codons of derivations on a gene. Default: 25. (`"sge"`).

`codonBits` The number of bits of a codon. Default: 0. (`"sge"`).

`codonPrecision` Specify the method to set the number of bits of a codon (`"sge"`):

- "Min": Sufficient to code the maximal number of choices of production rules for a non-terminal.
- "LCM": Contains the least common multiple of the prime factors of the number of choices of production rules for all non-terminals.
- "MaxPBias": The computed precision guarantees that the choice rule bias for a non-terminal is below `maxPBias`.

Argument of function factory `xegaGePrecisionFactory` in package `xegaGeGene`.

`maxPBias` The threshold of the choice rule bias. Default: `0.01`. (`sge`").

`evalmethod` Specifies the method of function evaluation:

- "EvalGeneU": The function is always evaluated. (Default)
- "EvalGeneR": The function is always evaluated. Repairs of the gene by the decoder are possible.
- "Deterministic": The function is evaluated only once.
- "Stochastic": The expected function value and its variance are incrementally updated.

Argument of function factory `EvalGeneFactory` in package xegaSelectGene.

`reportEvalErrors`

Report errors in the evaluation of fitness functions. Default: TRUE.

`genemap` Gene map for decoding. Default: "Bin2Dec". The default value works only for algorithm "sga". Used as `method` argument of the function factory `sgXGeneMapFactory` of package `xega`.

Available available options determined by `algorithm`:

- "sga": Binary representation (Default).
  - "Bin2Dec": For real parameter vectors.
  - "Gray2Dec": For real parameter vectors.
  - "Identity": For 0/1 parameter vectors.
  - "Permutation": For permutations.

  See the function factory `xegaGaGeneMapFactory` in package `xegaGaGene`.
- "sgp": Derivation tree. Gene map is not used, but must be specified. We use `xegaGaGene::xegaGaGeneMapFactory` with `method="Identity"`.

- "sge": Binary representation (Default). How are genes decoded?
  - "Mod": The modulo rule.
  - "Bucket": The bucket rule (with the mLCM). Problem: Mapping `1:2^k` to `1:mLCMG`.

  See the function factory `xegaGeGeneMapFactory` in package `xegaGeGene`.
- "sgde": Real coded gene. We use `xegaDfGene::xegaDfGeneMapFactory` with `method="Identity"`. Function used: `xegaDfGene::xegaDfGeneMapIdentity`
- "sgperm": Permutation gene. Gene map is not used, but must be specified. We use `xegaDfGene::xegaDfGeneMapFactory` with `method="Identity"`. Function used: `xegaDfGene::xegaDfGeneMapIdentity`

end of genemap

`crossrate2` Crossover rate for genes with below "average" fitness. Probability of applying crossover operator for genes with a "below average" fitness. Default: 0.30. (Global parameter)

`ivcrossrate` Specifies the method of determining the crossover rate.

- "Const" Constant crossover rate. The probability of applying the crossover operator is constant for the whole run of the algorithm. Default: "Const".
- "IV" Individually variable crossover rate. The crossrate of a gene is determined by the following threshold rule: If the fitness of the gene is higher than `lF$CutoffFit()*` `lF$CBestFitness()` then `lF$CrossRate1()` else `lF$CrossRate2()` is used.

Argument of function factory `CrossRateFactory` in package `xegaPopulation`.

`crossover` Crossover method. Default: "CrossGene". The choice of crossover methods depends on the setting of the argument `algorithm`. Used as the `method` argument in function factory `sgXCrossoverFactory` of package `xega`.

- `algorithm="sga"`: `crossover` is argument of function factory `xegaGaCrossoverFactory` in package `xegaGaGene`.
  - Crossover operators with 1 kid:
    - "CrossGene" one-point crossover.
    - "UCrossGene" uniform crossover.
    - "UPCrossgene" parameterized uniform crossover. Local parameter: `uCrossSwap`.
  - Crossover operators with 2 kids:
    - "Cross2Gene" one-point crossover.
    - "UCross2Gene" uniform crossover.
    - "UPCross2gene" parameterized uniform crossover. Local parameter: `uCrossSwap`.
- `algorithm="sgp"`: `crossover` is argument of function factory `xegaGpCrossoverFactory` in package `xegaGpGene`.
  - Crossover operators with 1 kid:
    - "CrossGene" position based one-point crossover.
  - Crossover operators with 2 kids:
    - "Cross2Gene" position based one-point crossover.

- algorithm="sge": We use the factory xegaGaCrossoverFactory. (Adpatation needed for variable-length binary representation.)
- algorithm="sgde": crossover is argument of function factory xegaDfCrossoverFactory in package xegaDfGene.
  - Crossover operators with 1 kid:
    - "CrossGene" one-point crossover (of reals)
    - "UCrossGene" uniform crossover (of reals)
    - "UPCrossGene" parametrized uniform crossover (of reals). Local parameter: uCrossSwap.
  - Crossover operators with 2 kids: Not implemented.
- algorithm="sgperm": crossover is argument of function factory xegaPermCrossoverFactory in package xegaPermGene.
  - Crossover operators with 1 kid:
    - "CrossGene" position based one-point crossover.
  - Crossover operators with 2 kids:
    - "Cross2Gene" position based one-point crossover.

uCrossSwap
: The fraction of positions swapped in the parametrized uniform crossover operator. A local crossover parameter. Default: 0.2. ("sga" and "sgde"). Used in packages xegaGaGene and xegaDfGene for functions xegaGaUPCross2Gene, xegaDfUPCross2Gene, xegaGaUPCrossGene, and xegaDfUPCrossGene.

mincrossdepth
: minimal depth of exchange nodes (roots of subtrees swapped by crossover). ("sgp").

maxcrossdepth
: Maximal depth of exchange nodes (roots of subtrees swapped by crossover). ("sgp"). Used in package xegaGpGene functions xegaGpCrossGene and xegaGpCross2Gene in package xegaGpGene.

ivmutrate
: "Const" or "IV" (individually variable). Default: "Const".

mutrate2
: Mutation rate. Default: 1.0. (Global parameter).

bitmutrate
: Bit mutation rate. Default: 0.005. A local mutation parameter. ("sga" and "sge"). Used in package xegaGaGene functions MutateGene IVAdaptiveMutateGene

bitmutrate2
: Bit mutation rate for genes with "below average" fitness. Default: 0.01. A local mutation parameter. ("sga" and "sge"). Used in package xegaGaGene functions IVAdaptiveMutateGene

maxmutdepth
: Maximal depth of a derivation tree inserted by mutation. Default: 3. ("sgp").

minmutinsertiondepth
: Minimal depth at which an insertion tree is inserted. Default: 1. ("sgp").

maxmutinsertiondepth
: Maximal depth at which an insertion tree is inserted. Default: 7. ("sgp"). Used in package xegaGpGene function xegaGpMutateGene.

lambda
: Decay rate. Default: 0.05. A local mutation parameter. ("sgperm"). Used in package xegaPermGene function xegaPermMutateGenekInversion.

max2opt
: Maximal number of trials to find an improvement by a random edge exchange in a permutation. Default: 100. ("sgperm"). Used in package xegaPermGene function xegaPermMutateGene2Opt. and xegaPermMutateGeneOptLK.

scalefactor1 Scale factor for differential mutation operator (Default: 0.9). ("sgde").

scalefactor2 Scale factor for differential mutation operator (Default: 0.2). ("sgde").

scalefactor Method for setting scale factor ("sgde"):

- "Const": constant scale factor.
- "Uniform": a random scale factor in 0.000001 to 1.0.

cutoffFit Cutoff for fitness. Default: 0.5. ("sga" and "sge"). Used in package xegaGaGene function IVAdaptiveMutateGene.

mutation Label specifies mutation method dependend on algorithm. Default: "MutateGene". The (global) probability of calling a mutation method is specified by mutrate and mutrate2. Used as method argument of function factory sgXMutationFactory package xega.

- algorithm="sga": mutation is argument of function factory xegaGaMutationFactory in package xegaGaGene.
  - "MutateGene": Bitwise mutation. Local parameter: bitmutrate. Function used: xegaGaGene::xegaGaMutateGene.
  - "IVM": Invividually variable mutation. Intuitively we know that bad genes need higher mutation rates. Good genes have a fitness which is above a threshold fitness. The threshold is determined as a percentage of the current best fitness in the population. The percentage is set by the parameter cutoffFit. Local parameters: bitmutrate for good genes. bitmutrate2 for bad genes. bitmutrate2 should be higher then bitmutrate.
- algorithm="sgp": mutation is argument of function factory xegaGpMutationFactory in package xegaGpGene.
  - "MutateGene": Random insertion of a random derivation tree. Local parameter: maxmutdepth. Function used: xegaGpGene::xegaGpMutateGene.
- algorithm="sge": mutation is argument of function factory xegaGaMutationFactory. Nothing specific to grammatical evolution implemented.
- algorithm="sgde": mutation is argument of function factory xegaDfMutationFactory in package xegaDfGene.
  - "MutateGene": Add the scaled difference of the parameters of two randomly selected to a gene. Local parameters: Choice of function for scalefactor as well as scalefactor1 and scalefactor2. Function used: xegaDfGene::xegaDfMutateGeneDE.
- algorithm="sgperm": mutation is argument of function factory xegaPermMutationFactory in package xegaPermGene.
  - "MutateGene": Function used: xegaPermGene::xegaPermMutateGeneOrderBased.
  - "MutateGeneOrderBased": See "MutateGene".
  - "MutateGenekInversion": Function used: xegaPermGene::xegaPermMutateGenekInversion.
  - "MutateGene2Opt": Function used: xegaPermGene::xegaPermMutateGene2Opt.
  - "MutateGenekOptLK": Function used: xegaPermGene::xegaPermMutateGenekOptLK.
  - "MutateGeneGreedy": Function used: xegaPermGene::xegaPermMutateGeneGreedy.
  - "MutateGeneBestGreedy": Function used: xegaPermGene::xegaPermMutateGeneBestGreedy
  - "MutateGeneMix": Function used: xegaPermGene::xegaPermMutateMix.

`replication` "Kid1" or "Kid2". Default: "Kid1". For algorithms "sga", "sgPerm", "sgp", and "sge": "Kid1" means a crossover operator with one kid, "Kid2" means a crossover operator with two kids.

For algorithm "sgde", `replication` must be set to "DE".

Used as the `method` argument of the function factory `sgXReplicationFactory` of package `xega`.

`offset` Offset used in proportional selection. Default: 1. Used in the following functions of package `xegaSelectGene`: `ScaleFitness`, `PropFitOnLn`, `PropFit`, `PropFitM`, `PropFitDiffOnLn`, `PropFitDiff`, `SUS`.

`eps` Epsilon in proportional fitness difference selection. Default: 0.01. Used in package `xegaSelectGene` function `PropFitDiffM`.

`tournamentSize` Tournament size. Default: 2. Used in package `xegaSelectGene` functions `SelectTournament`, `SelectSTournament`.

`selectionBias` (> 1.0). Controls selection pressure for Whitley's linear rank selection with selective pressure. Default: 1.5. Near 1.0: almost uniform selection. Used in package `xegaSelectGene` function `SelectLRSelective`,

`maxTSR` Controls selection pressure for Grefenstette and Baker's linear rank selection method. Should be higher than 1.0 and lower equal 2.0. Default: 1.5. Used in package `xegaSelectGene` function `SelectLinearRankTSR`.

`selection` Selection method for first parent of crossover. Default: "SUS".

`mateselection` Selection method for second parent of crossover. Default: "SUS".

Available selection methods for selection method of a parent:

- Uniform random selection: "Uniform".
- Uniform random selection without replacement: "UniformP".
- Proportional to fitness: "ProportionalOnln" (fastest), "Proportional", "ProportionalM",
- Proportional to fitness differences: "PropFitDiffOnln" (fastest), "PropfitDiff", "PropfitDiffM",
- Stochastic universal sampling: "SUS",
- Tournament selection: "Duel" (fastest), "Tournament", "STournament",
- Rank selection: "LRSelective" (fastest), "LRTSR".

Argument of function factory `SelectGeneFactory` in package `xegaSelectGene`.

`selectionContinuation`

Boolean. If `TRUE`, precomputes selection indices for next generation once and transforms selection function to index lookup continuation. Default: `TRUE`. Used in package `xegaPopulation` function `xegaNextPopulation`.

`scaling` Scaling method. Default: "NoScaling". Available scaling methods:

- "NoScaling",
- "ConstantScaling" (Static),
- "ThresholdScaling" (Dynamic),
- "ContinuousScaling" (Dynamic).

Argument of function factory `ScalingFactory` in package `xegaSelectGene`.

scalingThreshold
: Numerical constant. Default: 0.0. If ratio of dispersion measures is in [(1-scalingThreshold), 1+scalingThreshold)], fitness is not scaled. Used in package xegaSelectGene function ThresholdScaleFitness.

scalingExp
: Scaling exponent k in fit^k. With "ConstantScaling": 0 =< k. With "ThresholdScaling": 1 < k. (Default: 1) Used in package xegaSelectGene, functions ScalingFitness, ThresholdScaleFitness.

scalingExp2
: Scaling exponent for "ThresholdScaling": 0 <= k <1. (Default:1) Used in package xegaSelectGene function ThresholdScaleFitness.

rdmWeight
: Numerical constant. Default: 1.0. Weight of ratio of dispersion measures in continuous scaling. Used in package xegaSelectGene function ContinuousScaleFitness.

drMax
: Maximal allowable dispersion ratio. Default: 2.0. Used in package xegaSelectGene function DispersionRatio.

drMin
: Minimal allowable dispersion ratio. Default: 0.5. Used in package xegaSelectGene function DispersionRatio.

dispersionMeasure
: Dispersion measure used for computation of the ratio of dispersion measures for dynamic scaling methods. Default: "var". Available dispersion measures: "var, "std", "mad", "cv", "range", "iqr". Argument of function factory DispersionMeasureFactory in package xegaSelectGene.

scalingDelay
: The ratio of dispersion measures compares the current population dispersion at t with the population dispersion at t-scalingdelay. Default: 1. Used in package xegaSelectGene function DispersionRatio.

accept
: Acceptance rule for new gene. Default: "All".

  - "All" function AcceptNewGene
  - "Best" function AcceptBest
  - "Metropolis" function AcceptMetropolis. The behavior of this acceptance rule depends on:
    1. The distance between the fitness values. The larger the distance the larger the drop in acceptance probability.
    2. alpha is 1 minus the discount rate of the cooling schedule. alpha is in [0, 1]. The smaller alpha the faster the drop in temperatur and thus acceptance probability.
    3. beta a constant. The larger beta the faster the drop in acceptance probability.
    4. temperature the starting value of the temperature. Must be higher than the number of generations.
  - "IVMetropolis" function AcceptIVMetropolis. The behavior of this acceptance rule is qualitatively the same as that of the Metropolis acceptance rule above. The acceptance rule is adaptive by a correction of the temperature in proportion to the difference between the fitness of the current best and the fitness of the gene considered.

  Argument of function factory AcceptFactory in package xegaPopulation.

alpha
: 1 minus the discount rate for temperature. (Default: 0.99). (Used in cooling schedule at the end of main GA-loop.)

`beta` Constant in Metropolis acceptance rule. (Default: 2.0). (Used in Metroplis acceptance rule.)

`cooling` Cooling schedule for temperature. (Default: "ExponentialMultiplicative")

- "ExponentialMultiplicative" calls `ExponentialMultiplicativeCooling`
- "LogarithmicMultiplicative" calls `LogarithmicMultiplicativeCooling`
- "PowerMultiplicative" calls `PowerMultiplicativeCooling`
- "PowerAdditive" calls `PowerAdditiveCooling`
- "ExponentialAdditive" calls `ExponentialAdditiveCooling`
- "TrigonometricAdditive" calls `TrigonometricAdditiveCooling`

Argument of function factory `CoolingFactory` in package `xegaPopulation`.

`coolingPower` Exponent for PowerMultiplicative cooling schedule. (Default: 1. This is called linear multiplicative cooling.)

`temp0` Starting value of temperature (Default: 40). (Used in Metroplis acceptance rule. Updated in cooling schedule.)

`tempN` Final value of temperature (Default: 0.01). (Used in Metroplis acceptance rule. Updated in cooling schedule.)

`verbose` The value of `verbose` (Default: 1) controls the information displayed:

1. `== 0`: Nothing is displayed.
2. `== 1`: 1 point per generation.
3. `> 1`: Max(fit), number of solutions, indices.
4. `> 2`: and population fitness statistics.
5. `> 3`: and fitness, value of phenotype, and phenotype.
6. `> 4`: and str(genotype).

`logevals` Boolean. If `TRUE` then log all evaluations and their parameters in the file `xegaEvalLog<time stamp>.rds`. Default: FALSE.

`log<-readRDS(xegaEvalLog<time stamp>.rds)` reads the log. The format of a row of `log` is <fitness> <parameters>.

`allsolutions` Boolean. If `TRUE`, then return all best solutions. Default: `FALSE`.

`early` Boolean. If FALSE (Default), ignore code for early termination. See Parabola2DEarly.

`terminationEps` Fraction of known optimal solution for computing termination interval. Default: 0.01 See Parabola2DEarly.

`cores` Number of cores used for multi core parallel execution. (Default: NA. NA means that the number of cores is set by `parallelly:availableCores()` if the execution model is "MultiCore".

`executionModel` Execution model of fitness function evaluation. Available:

- "Sequential": `base::lapply` is used.
- "MultiCore": `parallel::mclapply` is used.
- "FutureApply": `future.apply::future_lapply` is used.
- "Cluster": Requires a proper configuration of the cluster.

Default: "Sequential".

`uParApply` A user defined parallel apply function (e.g. for Rmpi). If specified, overrides settings for `executionModel`. Default: `NULL`.

Cluster	A cluster object generated by parallel::makeCluster() or parallelly::makeCluster(). Default: NULL.
profile	Boolean. If TRUE measures execution time and counts number of executions to main components of genetic algorithms. Default: FALSE.
batch	Boolean. If TRUE then save result in file xegaResult<time stamp>.rds. Default: FALSE
path	Path. Default: "".

Details

The algorithm expects a problem environment penv which is a named list with at least the following functions:

- $name(): The name of the problem environment.
- $f(parm, gene=0, lF=0): The function to optimize. The parameters gene and lF are provided for future extensions.

Additional parameters needed depend on the algorithm and the problem environment. For example, for binary genes for function optimization, additional elements must be provided:

- $bitlength(): The vector of the bitlengths of the parameters.
- $genelength(): The number of bits of a gene.
- $lb(): The vector of lower bounds of the parameters.
- $ub(): The vector of upper bounds of the parameters.

Value

Result object. A named list of

1. $popStat: A matrix with mean, min, Q1, median, Q3, max, variance, and median absolute deviation of population fitness as columns: i-th row for the measures of the i-th generation.
2. $fit: Fitness vector if generations<=1 else: NULL.
3. $solution: Named list with fields
   - $solution$name: Name of problem environment.
   - $solution$fitness: Fitness value of the best solution.
   - $solution$value: The evaluated best gene.
   - $solution$numberofsolutions: Number of solutions with the same fitness.
   - $solution$genotype: The gene a genetic code.
   - $solution$phenotype: The decoded gene.
   - $solution$phenotypeValue: The value of the function of the parameters of the solution.
   - $solution$evalFail: Number of failures or fitness evaluations
   - and, if configured, $solution$allgenotypes, as well as $solution$allphenotypes.
4. $GAconfig: For rerun with xegaReRun().
5. $GAenv: Attribute value list of GAconfig.
6. $timer: An attribute value list with the time used (in seconds) in the main blocks of the GA: tUsed, tInit, tNext, tEval, tObserve, and tSummary.

Problem Specification

The problem specification consists of

- `penv`: The problem environment.
- `max`: Maximize? Boolean. Default: `TRUE`.
- `grammar`: A grammar object. For the algorithms `"sgp"` and `"sge"`.

Basic Parameters

The main parameters of a "standard" genetic algorithm are:

- `popsize`: Population size.
- `generations`: Number of generations.
- `crossrate`: Constant probability of one-point crossover.
- `mutrate`: Constant probability of mutation.

`crossrate` and `mutrate` specify the probability of applying the genetic operators crossover and mutation to a gene.

Two more parameters are important:

- `elitist`: Boolean. If `TRUE` (default), the fittest gene always survives.
- `replay`: Integer. If `0` (default), a random seed of the random number generator is chosen. For exact replications of a run of a genetic algorithm, set replay to a positive integer.

Global and Local Parameters

However, when using uniform crossover instead of one-point crossover, an additional parameter which specifies the probability of taking a bit from the first parent becomes necessary. Therefore, we distinguish between global and local operator parameters:

1. Global operator parameters: The probabilities of applying a crossover (`crossrate`) or a mutation operator (`mutrate`) to a gene.
2. Local operator parameters: E.g. the per bit probability of mutation or the probability of taking a bit from parent 1 for the uniform crossover operator. Local operator parameters affect only the genetic operator which needs them.

There exist several advantages of this classification of parameters:

- For the formal analysis of the behavior of the algorithms, we achieve a division in two parts: The equations of the global parameters with operator specific expressions as plug-ins.
- For empirically finding parameterizations for problem classes, we propose to fix local parameters at reasonable values (e.g. based on biological evidence) and and conditional on this optimize the (few) remaining global parameters.
- For parallelization specialized gene processing pipelines can be built and more efficiently executed, because the global parameters `crossrate` and `mutrate` decide which genes survive
  1. unchanged,
  2. mutated,

3. crossed, and
4. crossed as well as mutated.

To mimic a classic genetic algorithm with crossover and bit mutation rate, the probability of applying the mutation operator to a gene should be set to 1.

Global Adaptive Mechanisms

The adaptive mechanisms described in the following are based on threshold rules which determine how a parameter of the genetic operator is adapted. The threshold conditions are based on population statistics:

Adaptive Scaling. For adaptive scaling, select a dynamic scaling method, e.g. `scaling="ThresholdScaling"`. A high selection pressure decreases the dispersion in the population. The parameter `scalingThreshold` is a numerical parameter which defines an interval from `1-scalingThreshold` to `1+scalingThreshold`:

1. If the RDM is in this interval, the fitness function is not scaled.
2. If the RDM is larger than the upper bound of the interval, the constant `scalingExp` which is higher than 1 is chosen for the scaling function. This implements the rule: If the dispersion has increased, increase the selection pressure.
3. If the RDM is smaller than the lower bound of the interval, the constant `scalingExp2` which is smaller than 1 is chosen for the scaling function. This implements the rule: If the dispersion has decreased, increase the selection pressure.

The dispersion measure is computed as ratio of the dispersion measure at `t` relative to the dispersion measure at `t-scalingDelay`. The default dispersion measure is the variance of the population fitness (`dispersionMeasure="var"`). However, other dispersion measures ("std", "mad", "cv", "range", "iqr") can be configured.

Another adaptive scaling method is continuous scaling (`scaling="ContinuousScaling"`). The scaling exponent is adapted by a weighted ratio of dispersion measures. The weight of the exponent is set by `rdmWeight=1.1`, its default is `1.0`. Since the ratio of dispersion measures may be quite unstable, the default limits for the ratio are `drMin=0.5` and `drMax=2.0`.

Individually Variable Mutation and Crossover Probabilities

The rationale of individually variable mutation and crossover rates is that selected genes with a low fitness should be changed by a genetic operator with a higher probability. This increases the chance of survival of the gene because of the chance of a fitness increase through crossover or mutation.

Select an adaptive genetic operator rate: For the crossover rate, `ivcrossrate="IV"`. For the mutation rate, `ivmutrate="IV"`.

If the fitness of a gene is higher than `cutoffFit` times the current best fitness, the crossover rate is `crossrate` else the crossover rate is `crossrate2`.

If the fitness of a gene is higher than `cutoffFit` times the current best fitness, the mutation rate is `mutrate` else the mutation rate is `mutrate2`.

The Initialization of a Population

For the algorithms "sga", "sgde", and "sgperm" the information needed for initialization is the length of the gene in bits, in parameters, and in the number of symbols of a permutation. For "sgp", the depth bound gives an upper limit for the program which can be represented by a derivation tree.

For "sge", a codon is an integer for selecting a production rule. The number of bits of a genes is `codons*codonBits`.

Algorithm		Parameters
"sga"	Number of bits.	`penv$genelength()`
"sgde"	Number of parameters.	`length(penv$bitlength()`, `penv$lb()`, `penv$ub()`
"sgperm"	Number of symbols.	`penv$genelength()`
"sgp"	Depth bound of derivation tree.	`maxdepth`
"sge"	Number of codons and number of bits of a codon.	`codons`, `codonBits`, `codonPrecision`, `maxPBias`

The Pipeline of Genetic Operators

The pipeline of genetic operators merges the pipeline of a genetic algorithm with the pipeline of evolutionary algorithms and simulated annealing by adding an acceptance step:

- For evolutionary algorithms, the acceptance rule `accept="Best"` means that the fitter gene out of a parent and its kid survives (is copied into the next generation).
- For genetic algorithms the acceptance rule `accept="All"` means that always the kid survives.
- For simulated annealing the acceptance rule `accept="Metropolis"` means that the survival probability of a kid with a fitness worse than its parent decreases as the number of generations executed increases.

Proper configuration of the pipeline allows the configuration of new algorithm variants which mix elements of genetic, evolutionary, and simulated annealing algorithms.

The following table gives a working standard configuration of the pipeline of the genetic operators for each of the five algorithms:

Step/Algorithm	"sga"	"sgde"	"sgperm"	"sgp"	"sge"
(next) Scaling	NoScaling	NoScaling	NoScaling	NoScaling	NoScaling
(next) Selection	SUS	UniformP	SUS	SUS	SUS
(next) Replication	Kid2	DE	Kid2	Kid2	Kid2
(next) Crossover	Cross2Gene	UCrossGene	Cross2Gene	Cross2Gene	Cross2Gene
(next) Mutation	MutateGene	MutateGeneDE	MutateGene	MutateGene	MutateGene
(next) Acceptance	All	Best	All	All	All
(eval) Decoder	Bin2Dec	Identity	Identity	-	Mod
(eval) Evaluation	EvalGeneU	EvalGeneU	EvalGeneU	EvalGeneU	EvalGeneU

Scaling

In genetic algorithms scaling of the fitness functions has the purpose of increasing or decreasing the selection pressure. Two classes of scaling methods are available:

- Constant scaling methods.
  - No scaling (configured by `scaling="NoScaling"`).
  - Constant scaling (configured by `scaling="ConstantScaling"`). Depends on scaling exponent `scalingExp`.
- Adaptive scaling methods.

  - Threshold scaling (configured by `scaling="ThresholdScaling"`). It is configured with the scaling exponents `scalingExp` and `scalingExp2`, and the scaling threshold `scalingThreshold`. It uses a threshold rule about the change of a dispersion measure of the population fitness `lF$RDM()` to choose the scaling exponent:
    * `lF$RDM()>1+scalingThreshold`: The scaling exponent is `scalingExp` which should be greater than 1. Rationale: Increase selection pressure to reduce dispersion of fitness.
    * `lF$RDM()<1-scalingThreshold`: The scaling exponent is `scalingExp2` which should be lower than 1. Rationale: Decrease selection pressure to increase dispersion of fitness.
    * Else: Scaling exponent is 1. Fitness is not scaled.
  - Continuous scaling (configured by `scaling="ContinuousScaling"`). The ratio of the dispersion measures `lF$RDM()` is greater than 1 if the dispersion increased in the last generation and less than 1 if the dispersion decreased in the last generation. The scaling exponent is the product of the ratio of the dispersion measures `lF$RDM()` with the weight `rdmWeight`.

The change of the dispersion measure of the population fitness is measured by the function `lF$RDM()` (RDM means (R)atio of (D)ispersion (M)easure). This function depends on

- the choice of a dispersion measure of the population fitness `dispersionMeasure`. The variance is the default (`dispersionMeasure="var"`). The following dispersion measure of the population fitness are avalaible: Variance (`"var"`), standard deviation (`"std"`), median absolute deviation (`"mad"`), coefficient of variation (`"cv"`), range (`"range"`), inter quartile range (`"iqr"`).
- the scaling delay `scalingDelay`. The default is `scalingDelay=1`. This means the ratio of the variance of the fitness of the population at time t and the variance of the fitness of the population at time t-1 is computed.
- the upper and lower bounds of the ratio of dispersion measures.
- Dispersion ratios may have extreme fluctuations: The parameters `drMax` and `drMin` define upper and lower bounds of the ratio of dispersion measures. The defaults are `drMax=2` and `drMin=1`.

See package `xegaSelectGene` <https://CRAN.R-project.org/package=xegaSelectGene>

Selection

Selection operators determine which genes are chosen for the replication process for the next generation. Selection operators are configured by `selection` and `mateselection` (the 2nd parent for crossover). The default operator is stochastic universal selection for both parents (configured by `selection="SUS"` and `mateselection="SUS"`). The following operators are implemented:

- Uniform random selection with replacement (configured by `"Uniform"`). Needed for simulating uniform random mating behavior, computer experiments without selection pressure, for computing random search solutions as naive benchmarks.
- Uniform random selection without replacement (configured by `"UniformP"`). Needed for differential evolution.

- Selection proportional to fitness (in O(n) by "SelectPropFit", in O(n*log(n)) by "SelectPropFitOnln", and in O(n^2) by "SelectPropFitM"). offset configures the shift of the fitness vector if min(fit)=<0.
- Selection proportional to fitness differences (in O(n) by "SelectPropFitDiff", in O(n*log(n)) by "SelectPropFitDiffOnln", and in O(n^2) by "SelectPropFitDiffM"). Even the worst gene should have a minimal chance of survival: eps is added to the fitness difference vector. This also guarantees numerical stability for populations in which all genes have the same fitness.
- Deterministic tournament selection of k genes (configured by "Tournament"). The tournament size is configured by tournamentSize. Selection pressure increases with tournament size. The worst k-1 genes of a population never survive.
- Deterministic tournament selection of 2 genes (configured by "Duel").
- Stochastic tournament selection of k genes (configured by "STournament"). The tournament size is configured by tournamentSize.
- Linear rank selection with selective pressure (configured by "LRSelective"). The selection bias which regulates the selection pressure is configured by selectionBias (should be between 1.0 (uniform selection) and 2.0).
- Linear rank selection with interpolated target sampling rates (configured by "LRTSR"). The maximal target sampling rate is configured by maxTSR (should be between 1 and 2).
- Stochastic universal sampling (configured by "SUS").

If selectionContinuation=TRUE then selection functions are computed exactly once per generation. They are transformed into lookup-functions which deliver the index of selected genes by indexing a vector of integers.

See package xegaSelectGene <https://CRAN.R-project.org/package=xegaSelectGene>

Replication

For genetic algorithms ("sga", "sgp", sgperm", and "sge") in the replication process of a gene the crossover operator may by configured to produce one new gene (replication="Kid1") or two new genes (replication="Kid2"). The first version looses genetic information in the crossover operation, whereas the second version retains the genetic material in the population. There is a dependency between replication and crossover: "Kid2" requires a crossover operator which produces two kids. The replication method is configured by the function xegaGaReplicationFactory() of package xegaGaGene.

Note that only the function xegaGaReplicateGene of xegaGaGene (configured with replication="Kid1") implements a genetic operator pipeline with an acceptance rule.

For differential evolution (algorithm "sgde"), replication="DE" must be configured. The replication method for differential evolution is configured by the function xegaDfReplicationFactory() of package xegaDfGene. It implements a configurable acceptance rule. For classic differential evolution, use accept="Best".

Crossover

The table below summarizes the available crossover operators of the current version.

Algorithm:	**"sga"** and **"sge"**	**Package:**	**xegaGaGene**

Kids	Name	Function	crossover=	Influenced by
(2 kids)	1-Point	xegaGaCross2Gene()	"Cross2Gene"	
	Uniform	xegaGaUCross2Gene()	"UCross2Gene"	
	Parametrized Uniform	xegaGaUPCross2Gene()	"UPCross2Gene"	ucrossSwap
(1 kid)	1-Point	xegaGaCrossGene()	"CrossGene"	
	Uniform	xegaGaUCrossGene()	"UCrossGene"	
	Parametrized Uniform	xegaGaUPCrossGene()	"UPCrossGene"	ucrossSwap
Algorithm:	**"sgde"**	**Package:**	**xegaDfGene**	
(1 kid)	1-Point	xegaDfCrossGene()	"CrossGene"	
	Uniform	xegaDfCrossGene()	"UCrossGene"	
	Parametrized Uniform	xegaDfUPCrossGene()	"UPCrossGene"	ucrossSwap
Algorithm:	**"sgperm"**	**Package:**	**xegaPermGene**	
(2 kids)	Position-Based	xegaPermCross2Gene()	"Cross2Gene"	
(1 kid)	Position-Based	xegaPermCrossGene()	"CrossGene"	
Algorithm:	**"sgp"**	**Package:**	**xegaGpGene**	
(2 kids)	of Derivation Trees	xegaGpAllCross2Gene()	"Cross2Gene" or "All2Cross2Gene"	maxcrossdepth, maxdepth, and maxtrials
	of Depth-Filtered Derivation Trees	xegaGpFilterCross2Gene()	"FilterCross2Gene"	maxcrossdepth, mincrossdepth, maxdepth, and maxtrials
(1 kid)	of Derivation Trees	xegaGpAllCrossGene()	"CrossGene"	maxcrossdepth, maxdepth, and maxtrials
	of Depth-Filtered Derivation Trees	xegaGpFilterCrossGene()	"FilterCrossGene"	maxcrossdepth, mincrossdepth, maxdepth, and maxtrials

Mutation

The table below summarizes the available mutation operators of the current version.

Algorithm:	**"sga"** and **"sge"**	**Package:**	**xegaGaGene**
Name	Function	mutation=	Influenced by
Bit Mutation	xegaGaMutateGene()	"MutateGene"	bitmutrate
Individually Variable Bit Mutation	xegaGaIVAdaptiveMutateGene()	"IVM"	bitmutrate, bitmutrate2, and cutoffFit
Algorithm:	**"sgde"**	**Package:**	**xegaDfGene**
Differential Evolution Mutation	xegaDfMutateGeneDE()	"MutateGene" or "MutateGeneDe"	lF$ScaleFactor() (Configurable)
Algorithm:	**"sgperm"**	**Package:**	**xegaPermGene**
Generalized Order Based Mutation	xegaPermMutateGeneOrderBased()	"MutateGene" "MutateGeneOrderBased"	bitmutrate
k Inversion Mutation	xegaPermMutateGenekInversion()	"MutateGenekInversion"	lambda

2-Opt Mutation	xegaPermMutateGene2Opt()	"MutateGene2Opt"	max2opt
k-Opt LK Mutation (Lin-Kernighan)	xegaPermMutateGenekOptLK()	"MutateGenekOptLK"	max2opt
Greedy Path Mutation	xegaPermMutateGeneGreedy()	"MutateGeneGreedy"	lambda
Best Greedy Path Mutation	xegaPermMutateGeneBestGreedy()	"MutateGeneBestGreedy"	lambda
Random Mutation Operator	xegaPermMutateMix()	"MutateGeneMix"	
Algorithm:	**"sgp"**	**Package:**	**xegaGpGene**
Derivation Tree Mutation	xegaGpMutateAllGene()	"MutateGene" or "MutateAllGene"	maxmutdepth
Filtered Derivation Tree Mutation	xegaGpMutateGeneFilter()	"MutateFilterGene"	maxmutdepth, minmutinsertiondepth, and maxmutinsertiondepth

Acceptance

Acceptance rules are extensions of genetic and evolutionary algorithms which to the best of my knowledge have their origin in simulated annealing. An acceptance rule compares the fitness value of a modified gene with the fitness value of its parent and determines which of the two genes is passed into the next population.

An acceptance rule is only executed as part of the genetic operator pipeline, if `replicate="Kid1"` or `replicate="DE"`.

Two classes of acceptance rules are provided:

- Simple acceptance rules.
  - Accept the new gene unconditionally (configured by `accept="All"`). The new gene is always passed to the next population. Choose the rule for configuring a classic genetic algorithm. (The default).
  - Accept only best gene (configured by `accept="Best"`). This acceptance rule guarantees an increasing fitness curve over the run of the algorithm. For example, classic differential evolution uses this acceptance rule.
- Configurable acceptance rules. The rules always accept a new gene with a fitness improvement. They also accept a new gene with a lower fitness with a probability which depends on the fitness difference of the old and the new gene and a temperature parameter which is reduced over the algorithm run by a configurable cooling schedule.
  - The Metropolis acceptance rule (configured by `accept="Metropolis"`). The larger the parameter `beta` is set, the faster the drop in acceptance probability.
  - The individually adaptive Metropolis acceptance rule (configured by `accept="IVMetropolis"`). The larger the parameter `beta` is set, the faster the drop in acceptance probability. Individually adaptive means that the temperature is corrected. The correction (increase) of temperature depends on the difference between the fitness of the currently known best solution and the and the fitness of the new gene.

The cooling schedule updates the temperature parameter at the end of the main loop. The following cooling schedules are available:

- Exponential multiplicative cooling (configured by `cooling="ExponentialMultiplicative"`). Depends on the discount factor `alpha` and the start temperature `temp0`.
- Logarithmic multiplicative cooling (configured by `cooling="LogarithmicMultiplicative"`). Depends on the scaling factor `alpha` and the start temperature `temp0`.
- Power multiplicative cooling (configured by `cooling="PowerMultiplicative"`). Depends on the scaling factor `alpha`, the cooling power exponent `coolingPower`, and the start temperature `temp0`.
- Power additive cooling (configured by `cooling="PowerAdditive"`). Depends on the number of generations `generations`, the cooling power exponent `coolingPower`, the start temperature `temp0`, and the final temperature `tempN`.
- Exponential additive cooling (configured by `cooling="ExponentialAdditive"`). Depends on the number of generations `generations`, the start temperature `temp0`, and the final temperature `tempN`.
- Trigonometric additive cooling (configured by `cooling="TrigonometricAdditive"`). Depends on the number of generations `generations`, the start temperature `temp0`, and the final temperature `tempN`.

See package `xegaPopulation` <https://CRAN.R-project.org/package=xegaPopulation>

Decoder

Decoders are algorithm and task dependent. Their implementation often makes use of a gene map. The table below summarizes the available decoders and gene maps of the current version.

Algorithm:	**"sga"**	**"sgde"**	**"sgperm"**
In package:	xegaGaGene	xegaDfGene	xegaPermGene
Decoder:	xegaGaDecodeGene()	xegaDfDecodeGene()	xegaPermDecodeGene()
Gene map factories:	xegaGaGeneMapFactory()	xegaDfGeneMapFactory()	(Not configurable)
Method	"Bin2Dec"	"Identity"	
Method	"Gray2Dec"		
Method	"Identity"		
Method	"Permutation"		

Algorithm:	**"sgp"**	**"sge"**
In package:	xegaGpGene	xegaGeGene
Decoder:	xegaGpDecodeGene()	xegaGeDecodeGene()
Gene map factories:	(Not configurable)	xegaGeGeneMapFactory()
Method		"Mod"
Method		"Buck"

Evaluation

The method of evaluation of a gene is configured by `evalmethod`: "EvalGeneU" means that the function is always executed, "Deterministic" evaluates a gene only once, and "Stochastic" incrementally updates mean and variance of a stochastic function. If `reportEvalErrors==TRUE`, evaluation

failures are reported. However, for grammatical evolution without gene repair this should be set to FALSE. See package xegaSelectGene <https://CRAN.R-project.org/package=xegaSelectGene>

Distributed and Parallel Processing

The current scope of parallelization is the parallel evaluation of genes (the steps marked with (eval) in the genetic operator pipeline. This strategy is less efficient for differential evolution and permutation-based genetic algorithms because of the embedding of repeated evaluations into genetic operators.

In general, distributed and parallel processing requires a sequence of three steps:

1. Configure and start the distributed or parallel infrastructure.
2. Distribute processing and collect results. In a evolutionary or genetic algorithm the architectural pattern used for implementation coarse-grained parallelism by parallel evaluation of the fitness of the genes of a population is the master/worker pattern. In principle, the `lapply()`-function for evaluating a population of genes is replaced by a parallel version.
3. Stop the distributed or parallel infrastructure.

For evolutionary and genetic algorithm, the second step is controlled by two parameters, namely `executionModel` and `uParApply`:

1. If `uParApply=NULL`, then `executionModel` provides four ways of evaluating the fitness of a population of genes:
   (a) `executionModel="Sequential"`: The apply function used is `base::lapply()`. (Default).
   (b) `executionModel="MultiCore"`: The apply function used is `parallel::mclapply()`. If the number of cores is not specied by `cores`, the number of available cores is determined by `parallelly::availableCores()`.
   (c) `executionModel="FutureApply"`: The apply function used is `future.apply::future_lapply()`. The parallel/distributed model depends on a proper `future::plan()` statement.
   (d) `executionModel="Cluster"`: The apply function used is `parallel::parLapply()`. The information about the configuration of the computing cluster (master, port, list of workers) must be provided by `Cluster=cl` where `cl<-parallel::makeClusterPSOCK(rep(localhost, 5))` generates the cluster object and starts the R processes (of 5 workers in the same machine).
2. Assume that a user-defined parallel apply function has been defined and called `UPARAPPLY`. By setting `uParApply=UPARAPPLY`, the `lapply()` function used is `UPARAPPLY()`. This overrides the specification by `executionModel`. For example, parallelization via the MPI interface can be achieved by providing a user-defined parallel `lapply()` function which is implemented by a user-defined function whose function body is the line `Rmpi::mpi.parLapply(pop, FUN=EvalGene, lF=lF)`.

See package xegaPopulation <https://CRAN.R-project.org/package=xegaPopulation>

Acknowledgment. The author acknowledges support by the state of Baden-Württemberg through bwHPC.

Reporting

- verbose controls the information reported on the screen. If verbose is 1, then one dot is printed per generation to the console.
- reportEvalErrors=TRUE reports the output of errors of fitness function evaluations to the console. Grammatical evolution (algorithm "sge") routinely attempts to evaluate incomplete derivation trees. This leads to an evaluation error of the fitness function.
- profile=TRUE measures the time spent in executing the main blocks of the algorithm: InitPopulation(), NextPopulation(), EvalPopulation(), ObservePopulation(), and SummaryPopulation(). The measurements are stored in the named list $timer of the result object of the algorithm.
- allSolutions=TRUE collects all solutions with the same fitness value. The lists of the genotypes and phenotypes of these solutions are stored in $solution$allgenotypes and $allphenotypes of the result object of the algorithm.
- batch=TRUE writes the result object and logevals=TRUE writes a list of all evaluated genes to a rds-file in the current directory. path allows to write the rds-files into another directory. The existence of the directory specified by path is not checked. batch=TRUE combined with verbose=TRUE should be used in batch environments on HPC environments.

See Also

Other Main Program: xegaReRun()

Examples

```
a<-xegaRun(penv=Parabola2D, generations=10, popsize=20, verbose=0)
b<-xegaRun(penv=Parabola2D, algorithm="sga", generations=10, max=FALSE,
   verbose=1, replay=5, profile=TRUE)
c<-xegaRun(penv=Parabola2D, max=FALSE, algorithm="sgde",
   popsize=20, generations=50,
   mutation="MutateGeneDE", scalefactor="Uniform", crossover="UCrossGene",
   genemap="Identity", replication="DE",
   selection="UniformP", mateselection="UniformP", accept="Best")
envXOR<-NewEnvXOR()
BG<-compileBNF(booleanGrammar())
d<-xegaRun(penv=envXOR, grammar=BG, algorithm="sgp",
   generations=5, popsize=20, verbose=0)
e<-xegaRun(penv=envXOR, grammar=BG, algorithm="sge", genemap="Mod",
   generations=5, popsize=20, reportEvalErrors=FALSE, verbose=1)
f<-xegaRun(penv=lau15, max=FALSE, algorithm="sgperm",
   genemap="Identity", mutation="MutateGeneMix")
```

---

xegaVersion *About this version.*

Description

About this version.

Usage

```
xegaVersion(verbose = TRUE)
```

Arguments

verbose	Boolean. If TRUE (Default), print package information and version number to console.

Value

Version number (invisible).

Examples

```
xegaVersion()
```

Index