Package ‘xegaPopulation’

April 16, 2025

Title Genetic Population Level Functions

Version 1.0.0.7

Description This collection of gene representation-independent functions implements the population layer of extended evolutionary and genetic algorithms and its support. The population layer consists of functions for initializing, logging, observing, evaluating a population of genes, as well as of computing the next population. For parallel evaluation of a population of genes 4 execution models - named Sequential, MultiCore, FutureApply, and Cluster - are provided. They are implemented by configuring the lapply() function. The execution model FutureApply can be externally configured as recommended by Bengtsson (2021) <doi:10.32614/RJ-2021-048>. Configurable acceptance rules and cooling schedules (see Kirkpatrick, S., Gelatt, C. D. J, and Vecchi, M. P. (1983) <doi:10.1126/science.220.4598.671>, and Aarts, E., and Korst, J. (1989, ISBN:0-471-92146-7) offer simulated annealing or greedy randomized approximate search procedure elements. Adaptive crossover and mutation rates depending on population statistics generalize the approach of Stanhope, S. A. and Daida, J. M. (1996, ISBN:0-18-201-031-7).

License MIT + file LICENSE

URL `https://github.com/ageyerschulz/xegaPopulation`

Encoding UTF-8

RoxygenNote 7.3.2

Depends R (>= 4.0.0), parallel, future.apply, utils, stats

Suggests testthat (>= 3.0.0), future, parallelly

Imports xegaGaGene, xegaSelectGene

NeedsCompilation no

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

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

Repository CRAN

Date/Publication 2025-04-16 14:10:02 UTC

Contents

`AcceptBest` *Accepts only genes with equal or better fitness.*

Description

Change the gene by a genetic operator pipeline and return the new gene only if the new gene has at least the same fitness as the gene.

Usage

```
AcceptBest(OperatorPipeline, gene, lF)
```

Arguments

`OperatorPipeline`	Genetic operator pipeline.
`gene`	Gene.
`lF`	Local configuration.

Details

The fitness of genes never decreases. New genes with inferior fitness do not survive.

Value

The new gene, if it is at least as fit as gene else the old gene gene.

See Also

Other Acceptance Rule: `AcceptIVMetropolis()`, `AcceptMetropolis()`, `AcceptNewGene()`

Examples

```
OPpipe1<-function(g, lF){InitGene(lF)}
g1<-lFxegaGaGene$EvalGene(InitGene(lFxegaGaGene), lFxegaGaGene)
g2<-AcceptBest(OPpipe1, g1, lFxegaGaGene)
identical(g1, g2)
```

`AcceptFactory` *Configure the acceptance function of a genetic algorithm.*

Description

`AcceptanceFactory()` implements selection of an acceptance rule.

Current support:

1. "All" returns `AcceptNewGene()` (Default).
2. "Best" returns `AcceptBest()`.
3. "Metropolis" returns `AcceptMetropolis()`.
4. "IVMetropolis" returns `AcceptIVMetropolis()`.

Usage

```
AcceptFactory(method = "All")
```

Arguments

`method`	A string specifying the acceptance rule.

Value

An acceptance rule for genes.

See Also

Other Configuration: `ApplyFactory()`, `CoolingFactory()`, `CrossRateFactory()`, `MutationRateFactory()`, `TerminationFactory()`, `checkTerminationFactory()`, `xegaConfiguration()`, `xegaEvalPopulationFactory()`

`AcceptIVMetropolis` *Individually Adaptive Metropolis Acceptance Rule.*

Description

Change the gene by a genetic operator pipeline. Always accept new genes with fitness improvement. For maximizing fitness accept genes with lower fitness with probability `(runif(1)<exp(-(fitness-newfitness)*beta/Te` and reduce temperature with a cooling schedule. For each gene, the temperature is corrected upward by a term whose size is proportional to the difference between the fitness of the current best gene in the population and the fitness of the gene.

Usage

```
AcceptIVMetropolis(OperatorPipeline, gene, lF)
```

Arguments

`OperatorPipeline`	Genetic operator pipeline.
`gene`	Gene.
`lF`	Local configuration.

Details

The temperature is updated at the end of each generation in the main loop of the genetic algorithm.

Value

The new gene if it has at least equal performance as the old gene else the old gene.

References

Locatelli, M. (2000): Convergence of a Simulated Annealing Algorithm for Continuous Global Optimization. Journal of Global Optimization, 18:219-233. <doi:10.1023/A:1008339019740>

The-Crankshaft Publishing (2023): A Comparison of Cooling Schedules for Simulated Annealing. <URL:https://what-when-how.com/artificial-intelligence/a-comparison-of-cooling-schedules-for-simulated-annealing-artificial-intelligence/>

See Also

Other Acceptance Rule: `AcceptBest()`, `AcceptMetropolis()`, `AcceptNewGene()`

Examples

```
parm<-function(x){function() {return(x)}}
lFxegaGaGene$Beta<-parm(1)
lFxegaGaGene$TempK<-parm(10)
set.seed(2)
OPpipe1<-function(g, lF){InitGene(lF)}
g1<-lFxegaGaGene$EvalGene(InitGene(lFxegaGaGene), lFxegaGaGene)
lFxegaGaGene$CBestFitness<-parm(g1$fit)
g2<-AcceptMetropolis(OPpipe1, g1, lFxegaGaGene)
```

`AcceptMetropolis` *Metropolis Acceptance Rule.*

Description

Change the gene by a genetic operator pipeline. Always accept a new gene with a fitness improvement. For maximizing fitness accept genes with lower fitness with probability `(runif(1)<exp(-(fitness-newfitness)*be` and reduce temperature with a cooling schedule. Used: `Temperature<-alpha*Temperature` with `alpha<1`.

Usage

```
AcceptMetropolis(OperatorPipeline, gene, lF)
```

Arguments

`OperatorPipeline`	Genetic operator pipeline.
`gene`	Gene.
`lF`	Local configuration.

Details

The temperature is updated at the end of each generation in the main loop of the genetic algorithm.

Value

The new gene if it has at least equal performance as the old gene else the old gene.

References

Kirkpatrick, S., Gelatt, C. D. J, and Vecchi, M. P. (1983): Optimization by Simulated Annealing. Science, 220(4598): 671-680. <doi:10.1126/science.220.4598.671>

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., Teller, E. (1953): Equation of state calculations by fast computing machines. Journal of Chemical Physics, 21(6):1087 – 1092. <doi:10.1063/1.1699114>

See Also

Other Acceptance Rule: `AcceptBest()`, `AcceptIVMetropolis()`, `AcceptNewGene()`

Examples

```
parm<-function(x){function() {return(x)}}
lFxegaGaGene$Beta<-parm(1)
lFxegaGaGene$TempK<-parm(10)
OPpipe1<-function(g, lF){InitGene(lF)}
g1<-lFxegaGaGene$EvalGene(InitGene(lFxegaGaGene), lFxegaGaGene)
g2<-AcceptMetropolis(OPpipe1, g1, lFxegaGaGene)
```

`AcceptNewGene` *Accepts a new gene.*

Description

Executes a genetic operator pipeline. The new gene is returned.

Usage

```
AcceptNewGene(OperatorPipeline, gene, lF)
```

Arguments

`OperatorPipeline`	Genetic operator pipeline (an R function).
`gene`	Gene.
`lF`	Local configuration.

Value

New gene.

See Also

Other Acceptance Rule: `AcceptBest()`, `AcceptIVMetropolis()`, `AcceptMetropolis()`

Examples

```
id<-function(x, lF){x}
g1<-InitGene(lFxegaGaGene)
AcceptNewGene(id, g1, lFxegaGaGene)
```

ApplyFactory *Configure the the execution model for gene evaluation.*

Description

The current approach to distribution/parallelization of the genetic algorithm is to parallelize the evaluation of the fitness function only. The execution model defines the function `lF$lapply()` used in the function `EvalPopulation()`.

Usage

```
ApplyFactory(method = "Sequential")
```

Arguments

`method`	The label of the execution model: "Sequential" | "MultiCore" | "MultiCoreHet" | "FutureApply" | "FutureApplyHet" | "Cluster" | "ClusterHet" .

Details

Currently we support the following parallelization models:

1. "Sequential": Uses `base::lapply()`. (Default).
2. "MultiCore": Uses `parallel::mclapply()`. For tasks with approximately the same execution time.
3. "MultiCoreHet": Uses `parallel::mclapply()`. For tasks with a high variance of execution times.
4. "FutureApply": Uses `future.apply::future_lapply()` Plans must be set up and worker processes must be stopped.
5. "Cluster": Uses `parallel:parLapply()`. A cluster object must be set up and the worker processes must be stopped.

The execution model **"MultiCore"** provides parallelization restricted to a single computer: The master process starts R slave processes by fork() which are are run in separate memory spaces. At the time of fork() both memory spaces have the same content. Memory writes performed by one of the processes do not affect the other.

The execution model **"FutureApply"** makes the possibilities of the future backends for a wide range of parallel and distributed architectures available. The models of parallel resolving a future use different types of communication between master and slaves:

1. `plan(sequential)` configures sequential execution. Default.
2. `w<-5; plan(multicore, workers=w)` configures an asynchronous multicore execution of futures on 5 workers.

3. `w<-8; plan(multisession, workers=w)` configures a multisession environment with 5 workers. The evaluation of the future is done in parallel in 5 other R sessions on the same machine. Communication is done via socket connections, the R sessions started serve multiple futures over their life time. The worker R sessions are stopped by calling `plan(sequential)`. The number of parallel sessions is restricted by the availability of connections. Up to R version 4.3, a maximum of 125 connections is available.
4. `w<-7; plan(callr, workers=w)` configures the evaluation of futures on top of the `callr` package. The `callr` package creates for each future a separate R session. The communications is via files of serialized R objects. The advantages of `callr` are:
   (a) Each `callr` future is evaluated in a new R session which ends as soon as the value of the future has been collected.
   (b) The number of parallel `callr` futures is not restricted by the number of available connections, because the communication is based on files of serialized R objects.
   (c) No ports are used. This means no port clashes with other processes and no firewall issues.
5. Setting up a cluster environment for resolving futures works as follows. Write a function with the following elements:
   (a) Generate a cluster object:
   `cl<-makeClusterPSOCK(workers)`
   (b) Set up an on.exit condition for stopping the worker processes.
   `on.exit(parallel::stopCluster(cl))`
   (c) Set up the plan for resolving the future:
   `oldplan<-plan(cluster, workers=cl)`
   (d) Call the function with `future.apply::future_lapply`. E.g. the genetic algorithm.
   (e) Restore the previous plan: `plan(oldplan)`

   The cluster processes may be located on one or several computers. The communication between the processes is via sockets. Remote computers must allow the use of ssh to start R-processes without an interactive login.

The execution model **"Cluster"** allows the configuration of master-slave processing on local and remote machines.

For evaluating tasks with highly variable execution times, it is recommended to use the corresponding heterogenous execution models which assign one task per computing node and start a new task to a node as soon as his task is finished. These execution models are "MultiCoreHet", "FutureApplyHet", and "ClusterHet". Note that the communication and synchronization overhead of these execution models is substantially higher than for the homogenous execution models.

Value

A function with the same result as the `lapply()`-function.

See Also

Other Configuration: `AcceptFactory()`, `CoolingFactory()`, `CrossRateFactory()`, `MutationRateFactory()`, `TerminationFactory()`, `checkTerminationFactory()`, `xegaConfiguration()`, `xegaEvalPopulationFactory()`

`checkTerminatedFalse` *Check terminatedFalse()*

Description

Check terminatedFalse()

Usage

```
checkTerminatedFalse(penv, max)
```

Arguments

`penv`	A problem environment.
`max`	Maximize?

Value

A named list

- `$OK` `TRUE`
- `$penv` `penv`

`checkTerminateError` *Check terminateError()*

Description

Check terminateError()

Usage

```
checkTerminateError(penv, max)
```

Arguments

`penv`	A problem environment.
`max`	Maximize?

Value

A named list

- `$OK` `TRUE`
- `$penv` `penv`

`checkTerminatePAC` *Check terminatePAC()*

Description

Check terminatePAC()

Usage

```
checkTerminatePAC(penv, max)
```

Arguments

`penv`	A problem environment.
`max`	Maximize?

Value

A named list

- `$OK` `TRUE`
- `$penv` `penv`

`checkTerminationFactory` *Configure consistency checks and adapt* `penv` *for terminationConditions.*

Description

For each termination condition, a check must be provided. A check fails (stops) if the consistency requirements of a termination condition are not fulfilled. However, a check may modify the problem environment to establish consistency.

Usage

```
checkTerminationFactory(method = "NoTermination")
```

Arguments

`method`	A string specifying the termination condition.

Value

A check function.

See Also

Other Configuration: `AcceptFactory()`, `ApplyFactory()`, `CoolingFactory()`, `CrossRateFactory()`, `MutationRateFactory()`, `TerminationFactory()`, `xegaConfiguration()`, `xegaEvalPopulationFactory()`

`ConstCRate` *Constant crossover rate.*

Description

Constant crossover rate.

Usage

```
ConstCRate(fit, lF)
```

Arguments

`fit`	Fitness of gene.
`lF`	Local configuration.

Value

Constant crossover rate.

See Also

Other Rates: `ConstMRate()`

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list(CrossRate1=parm(0.20))
ConstCRate(100, lF)
ConstCRate(50, lF)
```

`ConstMRate` *Constant mutation rate.*

Description

Constant mutation rate.

Usage

```
ConstMRate(fit, lF)
```

Arguments

`fit`	Fitness of gene.
`lF`	Local configuration.

Value

Constant mutation rate.

See Also

Other Rates: `ConstCRate()`

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list()
lF$MutationRate1<-parm(0.20)
ConstMRate(100, lF)
ConstMRate(50, lF)
```

`CoolingFactory` *Configure the cooling schedule of the acceptance function of a genetic algorithm.*

Description

`CoolingFactory()` implements selection of a cooling schedule method.

Current support:

1. "ExponentialMultiplicative" returns `ExponentialMultiplicativeCooling`. (Default)
2. "LogarithmicMultiplicative" returns `LogarithmicMultiplicativeCooling`.
3. "PowerMultiplicative" returns `PowerMultiplicativeCooling`. `coolingPower=1` specifies linear multiplicative cooling, `coolingPower=2` specifies quadratic multiplicative cooling.

4. "PowerAdditive" returns `PowerAdditiveCooling`. `coolingPower=1` specifies linear additive cooling, `coolingPower=2` specifies quadratic additive cooling.
5. "ExponentialAdditive" returns `ExponentialAdditiveCooling`.
6. "TrigonometricAdditive" returns `TrigonometricAdditiveCooling`.

Usage

```
CoolingFactory(method = "ExponentialMultiplicative")
```

Arguments

`method`	A string specifying the cooling schedule.

Value

A cooling schedule.

See Also

Other Configuration: `AcceptFactory()`, `ApplyFactory()`, `CrossRateFactory()`, `MutationRateFactory()`, `TerminationFactory()`, `checkTerminationFactory()`, `xegaConfiguration()`, `xegaEvalPopulationFactory()`

`Cross2Gene` *Import for examples.*

Description

Import for examples.

Usage

```
Cross2Gene(gg1, gg2, lF)
```

Arguments

`gg1`	a gene
`gg2`	a gene
`lF`	list of local functions

Value

a list of two genes

`CrossGene` *Import for examples.*

Description

Import for examples.

Usage

```
CrossGene(gg1, gg2, lF)
```

Arguments

`gg1`	a gene
`gg2`	a gene
`lF`	list of local functions

Value

a list of one gene

`CrossRateFactory` *Configure the crossover function of a genetic algorithm.*

Description

`CrossRateFactory()` implements selection of one of the crossover rate 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. "Const" returns `ConstCRate()`.
2. "IV" returns `IACrate()`. This function gives bad genes a higher cross rate.

Usage

```
CrossRateFactory(method = "Const")
```

Arguments

`method`	A string specifying a function for the crossover rate.

Value

Crossover rate function.

See Also

Other Configuration: `AcceptFactory()`, `ApplyFactory()`, `CoolingFactory()`, `MutationRateFactory()`, `TerminationFactory()`, `checkTerminationFactory()`, `xegaConfiguration()`, `xegaEvalPopulationFactory()`

Examples

```
f<-CrossRateFactory("Const")
f(10, list(CrossRate1=function() {0.2}))
```

ExponentialAdditiveCooling

Exponential additive cooling.

Description

This schedule decreases in proportion to the inverse of `exp` raised to the power of the temperature cycle in `lF$Generations()` (= number of generations) fractions between the starting temperature `temp0` and the final temperature `tempN`.

Usage

```
ExponentialAdditiveCooling(k, lF)
```

Arguments

`k`	Number of steps to discount.
`lF`	Local configuration.

Details

Temperature is updated at the end of each generation in the main loop of the genetic algorithm. `lF$Temp0()` is the starting temperature. `lF$TempN()` is the final temperature. `lF$Generations()` is the number of generations (time).

Value

The temperature at time k.

References

The-Crankshaft Publishing (2023) A Comparison of Cooling Schedules for Simulated Annealing. <https://what-when-how.com/artificial-intelligence/a-comparison-of-cooling-schedules-for-simulated-annealing-artificial-intelligence/>

See Also

Other Cooling: `ExponentialMultiplicativeCooling()`, `LogarithmicMultiplicativeCooling()`, `PowerAdditiveCooling()`, `PowerMultiplicativeCooling()`, `TrigonometricAdditiveCooling()`

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list(Temp0=parm(100), TempN=parm(10), Generations=parm(50))
ExponentialAdditiveCooling(0, lF)
ExponentialAdditiveCooling(2, lF)
```

`ExponentialMultiplicativeCooling` *Exponential multiplicative cooling.*

Description

The temperature at time k is the net present value of the starting temperature. The discount factor is `lF$Alpha()`. `lF$Alpha()` should be in `[0, 1]`.

Usage

```
ExponentialMultiplicativeCooling(k, lF)
```

Arguments

`k`	Number of steps to discount.
`lF`	Local configuration.

Details

Temperature is updated at the end of each generation in the main loop of the genetic algorithm. `lF$Temp0()` is the starting temperature. `lF$Alpha()` is the discount factor.

Value

Temperature at time k.

References

Kirkpatrick, S., Gelatt, C. D. J, and Vecchi, M. P. (1983): Optimization by Simulated Annealing. Science, 220(4598): 671-680. <doi:10.1126/science.220.4598.671>

See Also

Other Cooling: `ExponentialAdditiveCooling()`, `LogarithmicMultiplicativeCooling()`, `PowerAdditiveCooling()`, `PowerMultiplicativeCooling()`, `TrigonometricAdditiveCooling()`

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list(Temp0=parm(100), Alpha=parm(0.99))
ExponentialMultiplicativeCooling(0, lF)
ExponentialMultiplicativeCooling(2, lF)
```

`futureLapply` *Future apply of R-package* `future.apply`.

Description

The `lapply()` function is redefined as as `future.apply::future_lapply()`. Henrik Bengtsson recommends that the configuration of the parallel/distributed programming environment should be kept outside the package and left to the user. The advantage is that the user may take advantage of all parallel/distributed available backends for the Future API.

Usage

```
futureLapply(pop, EvalGene, lF)
```

Arguments

`pop`	Population of genes.
`EvalGene`	Function for evaluating a gene.
`lF`	Local function factory which provides all functions needed in `EvalGene`.

Details

Be aware that

- `future_lapply()` assumes that each function evaluation need approximately the same time.
- Best results are obtained if `popsize` modulo `workers` is `0`.

Value

Fitness vector.

References

Bengtsson H (2021). “A Unifying Framework for Parallel and Distributed Processing in R using Futures.” The R Journal, 13(2), 208–227. <doi:10.32614/RJ-2021-048>

See Also

Other Execution Model: `MClapply()`, `MClapplyHet()`, `PparLapply()`, `PparLapplyHet()`, `futureLapplyHet()`

Examples

```
pop<-xegaInitPopulation(1000, lFxegaGaGene)
library(future)
plan(multisession, workers=2)
popnew<-futureLapply(pop, lFxegaGaGene$EvalGene, lFxegaGaGene)
plan(sequential)
```

futureLapplyHet — *Future apply of R-package* future.apply *configured for a tasks with heterogenous execution times.*

Description

The lapply() function is redefined as as future.apply::future_lapply().

Henrik Bengtsson recommends that the configuration of the parallel/distributed programming environment should be kept outside the package and left to the user. The advantage is that the user may take advantage of all parallel/distributed available backends for the Future API.

Usage

```
futureLapplyHet(pop, EvalGene, lF)
```

Arguments

pop	Population of genes.
EvalGene	Function for evaluating a gene.
lF	Local function factory which provides all functions needed in EvalGene.

Details

This configuration has an increased communication and synchronization overhead.

Value

Fitness vector.

References

Bengtsson H (2021). "A Unifying Framework for Parallel and Distributed Processing in R using Futures." The R Journal, 13(2), 208–227. <doi:10.32614/RJ-2021-048>

See Also

Other Execution Model: MClapply(), MClapplyHet(), PparLapply(), PparLapplyHet(), futureLapply()

Examples

```
pop<-xegaInitPopulation(30, lFxegaGaGene)
library(future)
plan(multisession, workers=2)
popnew<-futureLapplyHet(pop, lFxegaGaGene$EvalGene, lFxegaGaGene)
plan(sequential)
```

IACRate *Individually adaptive crossover rate.*

Description

The basic idea is to apply crossover to a gene whose fitness is below a threshold value with higher probability to give it a chance to improve. The threshold value is computed by `lF$CutoffFit()*lF$CBestFitness()`.

Usage

```
IACRate(fit, lF)
```

Arguments

`fit`	Fitness of gene.
`lF`	Local configuration.

Details

The following constants are used: `lF$CrossRate1()<lF$CrossRate2()`, and `lF$CutoffFit()` in [0, 1].

Value

Crossover rate of a gene depending on its fitness.

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 Adaptive Rates: `IAMRate()`

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list()
lF$CrossRate1<-parm(0.20)
lF$CrossRate2<-parm(0.40)
lF$CutoffFit<-parm(0.60)
lF$CBestFitness<-parm(105)
IACRate(100, lF)
IACRate(50, lF)
```

`IAMBitRate` *Individually adaptive mutation rate. (Bit mutation Rate)*

Description

Adaptivity of a local operator mutation parameter. Currently not used. Implements a threshold rule. The rule is implemented directly in IVAdaptiveMutateGene. in package xegaGaGene. Move?

Usage

```
IAMBitRate(fit, lF)
```

Arguments

`fit`	Fitness of gene.
`lF`	Local configuration.

Details

TODO: Move this xegaGaGene and generalize the bit mutation operator and introduce a factory for bit mutation rates. Rationale: Local parameters are representation dependent.

Value

Mutation rate of a gene depending on its fitness.

`IAMRate` *Individually adaptive mutation rate.*

Description

The probability of applying a mutation operator to a gene. The idea is that a gene selected for reproduction whose fitness is below a threshold value is mutated with a higher probability to give it a chance.

Usage

```
IAMRate(fit, lF)
```

Arguments

`fit`	Fitness of gene.
`lF`	Local configuration.

Details

The probability of applying a mutation operator is determined by a threshold: If the fitness of a gene is higher than `lF$CutoffFit()*lF$CBestFitness()`, than return `lF$MutationRate1()` else `lF$MutationRate2()`.

Note that the idea is also applicable to gene specific local mutation operators. For example, the bit mutation rate of mutation operators for binary genes.

Value

Mutation rate of a gene depending on its fitness.

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 Adaptive Rates: `IACRate()`

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list()
lF$MutationRate1<-parm(0.20)
lF$MutationRate2<-parm(0.40)
lF$CutoffFit<-parm(0.60)
lF$CBestFitness=parm(105)
IAMRate(100, lF)
IAMRate(50, lF)
```

`InitGene` *Import for examples.*

Description

Import for examples.

Usage

```
InitGene(lF)
```

Arguments

`lF`	a list of local functions

Value

a new random gene

`lFxegaGaGene` *Import for examples.*

Description

Import for examples.

Import lFxegaGaGene

Usage

```
lFxegaGaGene

lFxegaGaGene
```

Format

An object of class `list` of length 29.

An object of class `list` of length 29.

`LogarithmicMultiplicativeCooling` *Logarithmic multiplicative cooling.*

Description

This schedule decreases by the inverse proportion of the natural logarithm of `k`. `lF$Alpha()` should be larger than 1.

Usage

```
LogarithmicMultiplicativeCooling(k, lF)
```

Arguments

`k`	Number of steps to discount.
`lF`	Local configuration.

Details

Temperature is updated at the end of each generation in the main loop of the genetic algorithm. `lF$Temp0()` is the starting temperature. `lF$Alpha()` is a scaling factor.

Value

Temperature at time k.

Aarts, E., and Korst, J. (1989): Simulated Annealing and Boltzmann Machines. A Stochastic Approach to Combinatorial Optimization and Neural Computing. John Wiley & Sons, Chichester. (ISBN:0-471-92146-7)

See Also

Other Cooling: `ExponentialAdditiveCooling()`, `ExponentialMultiplicativeCooling()`, `PowerAdditiveCooling()`, `PowerMultiplicativeCooling()`, `TrigonometricAdditiveCooling()`

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list(Temp0=parm(100), Alpha=parm(1.01))
LogarithmicMultiplicativeCooling(0, lF)
LogarithmicMultiplicativeCooling(2, lF)
```

`MClapply` *MultiCore apply of library parallel.*

Description

The evaluation of the fitness of the genes of the population is distributed to one worker on each core of the CPU of the local machine. The package `parallel` of base R is used. The number of cores is provided by `lF$Cores`.

Usage

```
MClapply(pop, EvalGene, lF)
```

Arguments

`pop`	Population of genes.
`EvalGene`	Function for evaluating a gene.
`lF`	Local function configuration which provides all functions needed in `EvalGene()`.

Details

Be aware that

- `parallel::mclapply()` assumes that each function evaluation needs approximately the same time.
- Best results are obtained if `popsize` modulo `cores-1` is `0`.
- Does not work on Windows.

Value

Fitness vector.

See Also

Other Execution Model: `MClapplyHet()`, `PparLapply()`, `PparLapplyHet()`, `futureLapply()`, `futureLapplyHet()`

Examples

```
library(parallelly)
if (supportsMulticore()){
lFxegaGaGene$Cores<-function() {2}
pop<-xegaInitPopulation(1000, lFxegaGaGene)
popnew<-MClapply(pop, lFxegaGaGene$EvalGene, lFxegaGaGene)
}
```

`MClapplyHet` *MultiCore apply of library parallel for heterogenous tasks.*

Description

The evaluation of the fitness of the genes of the population is distributed to one worker on each core of the CPU of the local machine. The package `parallel` of base R is used. The number of cores is provided by `lF$Cores`.

Usage

```
MClapplyHet(pop, EvalGene, lF)
```

Arguments

`pop`	Population of genes.
`EvalGene`	Function for evaluating a gene.
`lF`	Local function configuration which provides all functions needed in `EvalGene()`.

Details

Be aware that

- `parallel::mclapply()` assumes that each function evaluation needs approximately the same time.
- Best results are obtained if `popsize` modulo `cores-1` is `0`.
- Does not work on Windows.

Value

Fitness vector.

See Also

Other Execution Model: `MClapply()`, `PparLapply()`, `PparLapplyHet()`, `futureLapply()`, `futureLapplyHet()`

Examples

```
library(parallelly)
if (supportsMulticore()){
lFxegaGaGene$Cores<-function() {2}
pop<-xegaInitPopulation(10, lFxegaGaGene)
popnew<-MClapplyHet(pop, lFxegaGaGene$EvalGene, lFxegaGaGene)
}
```

`MetropolisAcceptanceProbability` *Metropolis acceptance probability.*

Description

Metropolis acceptance probability.

Usage

```
MetropolisAcceptanceProbability(d, beta, temperature)
```

Arguments

`d`	Distance between the fitness of the old and the new gene.
`beta`	Constant.
`temperature`	Temperature.

Value

Acceptance probability.

See Also

Other Diagnostic: `MetropolisTable()`

Examples

```
MetropolisAcceptanceProbability(d=0, beta=1, temperature=10)
MetropolisAcceptanceProbability(d=1, beta=1, temperature=10)
```

`MetropolisTable` *Metropolis acceptance probability table.*

Description

Metropolis acceptance probability table.

Usage

```
MetropolisTable(d = 1, beta = 2, temperature = 1000, alpha = 0.9, steps = 1000)
```

Arguments

`d`	Distance between the fitness of the old and the new gene.
`beta`	Constant.
`temperature`	Temperature.
`alpha`	Cooling constant in [0, 1].
`steps`	Number of steps.

Value

Data frame with the columns alpha, beta, temperature, d (distance between fitness), and probability of acceptance.

See Also

Other Diagnostic: `MetropolisAcceptanceProbability()`

Examples

```
MetropolisTable(d=2, beta=2, temperature=10, alpha=0.99, steps=10)
```

`MutationRateFactory` *Configure the mutation rate function of a genetic algorithm.*

Description

The `MutationRateFactory()` implements selection of one of the crossover rate 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. "Const" returns `ConstMRate()` (Default).
2. "IV" returns `IAMrate()`. This function gives bad genes a higher mutation rate.

Usage

```
MutationRateFactory(method = "Const")
```

Arguments

method	A string specifying a function for the mutation rate.

Value

A mutation rate function.

See Also

Other Configuration: AcceptFactory(), ApplyFactory(), CoolingFactory(), CrossRateFactory(), TerminationFactory(), checkTerminationFactory(), xegaConfiguration(), xegaEvalPopulationFactory()

Examples

```
f<-MutationRateFactory("Const")
f(10, list(MutationRate1=function() {0.2}))
```

PowerAdditiveCooling *Power additive cooling.*

Description

This schedule decreases by a power of the n (= number of generations) linear fractions between the starting temperature `lF$Temp0` and the final temperature `lF$tempN`.

Usage

```
PowerAdditiveCooling(k, lF)
```

Arguments

k	Number of steps to discount.
lF	Local configuration.

Details

Temperature is updated at the end of each generation in the main loop of the genetic algorithm. `lF$Temp0()` is the starting temperature. `lF$TempN()` is the final temperature. `lF$CoolingPower()` is an exponential factor. `lF$Generations()` is the number of generations (time).

Value

Temperature at time k.

References

The-Crankshaft Publishing (2023) A Comparison of Cooling Schedules for Simulated Annealing. <https://what-when-how.com/artificial-intelligence/a-comparison-of-cooling-schedules-for-simulated-annealing-artificial-intelligence/>

See Also

Other Cooling: `ExponentialAdditiveCooling()`, `ExponentialMultiplicativeCooling()`, `LogarithmicMultiplicati`
`PowerMultiplicativeCooling()`, `TrigonometricAdditiveCooling()`

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list(Temp0=parm(100), TempN=parm(10), Generations=parm(50), CoolingPower=parm(2))
PowerAdditiveCooling(0, lF)
PowerAdditiveCooling(2, lF)
```

PowerMultiplicativeCooling

Power multiplicative cooling.

Description

This schedule decreases by the inverse proportion of a power of `k`. `lF$Alpha()` should be larger than 1.

Usage

```
PowerMultiplicativeCooling(k, lF)
```

Arguments

`k`	Number of steps to discount.
`lF`	Local configuration.

Details

Temperature is updated at the end of each generation in the main loop of the genetic algorithm. For `lF$CoolingPower()==1` and `lF$CoolingPower()==2` this results in the the linear and quadratic multiplicative cooling schemes in A Comparison of Cooling Schedules for Simulated Annealing. `lF$Temp0()` is the starting temperature. `lF$Alpha()` is a scaling factor. `lF$CoolingPower()` is an exponential factor.

Value

Temperature at time k.

References

The-Crankshaft Publishing (2023) A Comparison of Cooling Schedules for Simulated Annealing. <https://what-when-how.com/artificial-intelligence/a-comparison-of-cooling-schedules-for-simulated-annealing-artificial-intelligence/>

See Also

Other Cooling: `ExponentialAdditiveCooling()`, `ExponentialMultiplicativeCooling()`, `LogarithmicMultiplicati`
`PowerAdditiveCooling()`, `TrigonometricAdditiveCooling()`

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list(Temp0=parm(100), Alpha=parm(1.01), CoolingPower=parm(2))
PowerMultiplicativeCooling(0, lF)
PowerMultiplicativeCooling(2, lF)
```

`PparLapply` *uses parLapply of library parallel for using workers on machines in a local network.*

Description

uses parLapply of library parallel for using workers on machines in a local network.

Usage

```
PparLapply(pop, EvalGene, lF)
```

Arguments

`pop`	a population of genes.
`EvalGene`	the function for evaluating a gene.
`lF`	the local function factory which provides all functions needed in `EvalGene`.

Value

Fitness vector.

Warning

This section has not been properly tested. Random number generation? Examples?

See Also

Other Execution Model: `MClapply()`, `MClapplyHet()`, `PparLapplyHet()`, `futureLapply()`, `futureLapplyHet()`

Examples

```
parm<-function(x) {function() {x}}
pop<-xegaInitPopulation(1000, lFxegaGaGene)
library(parallel)
clus<-makeCluster(2)
lFxegaGaGene$cluster<-parm(clus)
popnew<-PparLapply(pop, lFxegaGaGene$EvalGene, lFxegaGaGene)
stopCluster(clus)
```

`PparLapplyHet` *uses parLapplyLB of library parallel for using workers on machines in a local network.*

Description

uses parLapplyLB of library parallel for using workers on machines in a local network.

Usage

```
PparLapplyHet(pop, EvalGene, lF)
```

Arguments

`pop`	a population of genes.
`EvalGene`	the function for evaluating a gene.
`lF`	the local function factory which provides all functions needed in `EvalGene`.

Value

Fitness vector.

Warning

This section has not been properly tested. Random number generation? Examples?

See Also

Other Execution Model: `MClapply()`, `MClapplyHet()`, `PparLapply()`, `futureLapply()`, `futureLapplyHet()`

Examples

```
parm<-function(x) {function() {x}}
pop<-xegaInitPopulation(1000, lFxegaGaGene)
library(parallel)
clus<-makeCluster(2)
lFxegaGaGene$cluster<-parm(clus)
popnew<-PparLapplyHet(pop, lFxegaGaGene$EvalGene, lFxegaGaGene)
stopCluster(clus)
```

`ReplicateGene` *Import for examples.*

Description

Import for examples.

Usage

```
ReplicateGene(pop, fit, lF)
```

Arguments

`pop`	the population.
`fit`	the fitness-
`lF`	list of local functions

Value

a list with one gene

`terminateAbsoluteError` *Terminates, if the absolute deviation from the global optimum is small.*

Description

`terminateAbsoluteError()` returns `TRUE` if the value of the current solution is in the interval from (globalOptimum - eps) to (globalOptimum + eps).

Usage

```
terminateAbsoluteError(solution, lF)
```

Arguments

`solution`	A named list with at least the following elements: $name, $fitness, $value, $numberOfSolutions, $genotype, $phenotype, $phenotypeValue.
`lF`	Local function configuration. It must contain • `lF$penv$globalOptimum()` which returns the global optimum. • `lF$TerminationEps()` which specifies the the maximal allowed deviation of the current best solution from the global optimum.

Details

Useful for benchmark functions with known global optima.

Value

Boolean.

See Also

Other Termination Condition: `terminateGEQ()`, `terminateLEQ()`, `terminatePAC()`, `terminateRelativeError()`, `terminateRelativeErrorZero()`, `terminatedFalse()`

Examples

```
parm<-function(x){function() {return(x)}}
olst<-list(); olst$value<-10
penv<-list(); penv$globalOptimum<-parm(olst)
lF<-list(); lF$penv<-penv; lF$TerminationEps<-parm(1.2);lF$Max<-parm(1.0)
solution<-list(); solution$genotype<-list(); solution$genotype$fit<-8.0
terminateAbsoluteError(solution, lF)
solution<-list(); solution$genotype<-list(); solution$genotype$fit<-8.9
terminateAbsoluteError(solution, lF)
```

`terminatedFalse` *No termination condition.*

Description

A boolean function which always returns `FALSE`.

Usage

```
terminatedFalse(solution, lF)
```

Arguments

`solution`	A named list with at least the following elements: $name, $fitness, $value, $numberOfSolutions, $genotype, $phenotype, $phenotypeValue.
`lF`	Local function configuration.

Value

FALSE

See Also

Other Termination Condition: `terminateAbsoluteError()`, `terminateGEQ()`, `terminateLEQ()`, `terminatePAC()`, `terminateRelativeError()`, `terminateRelativeErrorZero()`

Examples

```
lF<-list()
terminatedFalse(1.0, lF)
```

---

terminateGEQ *Terminates, if the solution is greater equal a threshold.*

Description

`terminateGEQ()` returns `TRUE` if the value of the current solution is greater or equal `lF$TerminationThreshold()`.

Usage

```
terminateGEQ(solution, lF)
```

Arguments

`solution`	A named list with at least the following elements: $name, $fitness, $value, $numberOfSolutions, $genotype, $phenotype, $phenotypeValue.
`lF`	Local function configuration. It must contain • `lF$TerminationThreshold()` which returns a numeric value.

Value

Boolean.

See Also

Other Termination Condition: `terminateAbsoluteError()`, `terminateLEQ()`, `terminatePAC()`, `terminateRelativeError()`, `terminateRelativeErrorZero()`, `terminatedFalse()`

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list(); lF$TerminationThreshold<-parm(9.2)
solution<-list(); solution$phenotypeValue<-8.0
terminateGEQ(solution, lF)
solution<-list(); solution$phenotypeValue<-9.6
terminateGEQ(solution, lF)
```

`terminateLEQ` *Terminates, if the solution is less equal a threshold.*

Description

`terminateLEQ()` returns `TRUE` if the value of the current solution is less or equal `lF$TerminationThreshold()`.

Usage

```
terminateLEQ(solution, lF)
```

Arguments

`solution`	A named list with at least the following elements: $name, $fitness, $value, $numberOfSolutions, $genotype, $phenotype, $phenotypeValue.
`lF`	Local function configuration. It must contain • `lF$TerminationThreshold()` which returns a numeric value.

Value

Boolean.

See Also

Other Termination Condition: `terminateAbsoluteError()`, `terminateGEQ()`, `terminatePAC()`, `terminateRelativeError()`, `terminateRelativeErrorZero()`, `terminatedFalse()`

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list(); lF$TerminationThreshold<-parm(9.2)
solution<-list(); solution$phenotypeValue<-8.0
terminateLEQ(solution, lF)
solution<-list(); solution$phenotypeValue<-9.6
terminateLEQ(solution, lF)
```

`terminatePAC` *Terminates if relative deviation from estimated PAC bound for optimum is small. Works at 0.*

Description

`terminatePAC()` returns `TRUE` if the value of the current solution is in the interval from (PACopt - (PACopt*eps)) to (PACopt + (PACopt*eps)). If PACopt is zero, test interval (0-eps) to (0+eps).

Usage

```
terminatePAC(solution, lF)
```

Arguments

`solution`	A named list with at least the following elements: $name, $fitness, $value, $numberOfSolutions, $genotype, $phenotype, $phenotypeValue.
`lF`	Local function configuration. It must contain • `lF$PACopt()` which returns an estimation of an upper PAC bound `ub` for the global optimum `g` with `P(ub<g)<lF$PACdelta()`. • `lF$TerminationEps()` which specifies the the fraction of the global optimum used for computing the upper and lower bounds for the interval in which the best current solution must be for terminating the algorithm.

Details

By an idea of M. Talagrand we estimate `lF$PACopt()` from the mean `m` and the standard deviation `s` of the population fitness of the first population of the genetic algorithm we compute `m+s*qnorm(lF$PACdelta(), lower.tail=FALSE)` when the function we optimize is in Hilbert space. For other spaces, this has to be adapted.

Value

Boolean.

See Also

Other Termination Condition: `terminateAbsoluteError()`, `terminateGEQ()`, `terminateLEQ()`, `terminateRelativeError()`, `terminateRelativeErrorZero()`, `terminatedFalse()`

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list(); lF$PACopt<-parm(10.0); lF$TerminationEps<-parm(1.2);lF$Max<-parm(1.0)
solution<-list(); solution$genotype<-list();  solution$genotype$fit<-0.5
terminatePAC(solution, lF)
solution<-list(); solution$genotype<-list();  solution$genotype$fit<-9.6
terminatePAC(solution, lF)
```

`terminateRelativeError` *Terminates, if the relative deviation from the global optimum is small.*

Description

`terminateRelativeError()` returns `TRUE` if the value of the current solution is in the interval from (globalOptimum - (globalOptimum*eps)) to (globalOptimum + (globalOptimum*eps)).

Usage

```
terminateRelativeError(solution, lF)
```

Arguments

`solution`	A named list with at least the following elements: $name, $fitness, $value, $numberOfSolutions, $genotype, $phenotype, $phenotypeValue.
`lF`	Local function configuration. It must contain • `lF$penv$globalOptimum()` which returns the global optimum. • `lF$TerminationEps()` which specifies the the fraction of the global optimum used for computing the upper and lower bounds for the interval in which the best current solution must be for terminating the algorithm.

Details

Useful for benchmark functions with known global optima. Note that for a global optimum of `0` this function fails.

Value

Boolean.

See Also

Other Termination Condition: `terminateAbsoluteError()`, `terminateGEQ()`, `terminateLEQ()`, `terminatePAC()`, `terminateRelativeErrorZero()`, `terminatedFalse()`

Examples

```
parm<-function(x){function() {return(x)}}
olst<-list(); olst$value<-10
penv<-list(); penv$globalOptimum<-parm(olst)
lF<-list(); lF$penv<-penv; lF$TerminationEps<-parm(1.2);lF$Max<-parm(1.0)
solution<-list(); solution$genotype<-list();  solution$genotype$fit<-8.0
terminateRelativeError(solution, lF)
solution<-list(); solution$genotype<-list();  solution$genotype$fit<-9.6
terminateRelativeError(solution, lF)
```

`terminateRelativeErrorZero`

Terminates if relative deviation from optimum is small. Works at 0.

Description

`terminateRelativeErrorZero()` returns `TRUE` if the value of the current solution is in the interval from (globalOptimum - (globalOptimum*eps)) to (globalOptimum + (globalOptimum*eps)). If globalOptimum is zero, test interval (0-eps) to (0+eps).

Usage

```
terminateRelativeErrorZero(solution, lF)
```

Arguments

solution	A named list with at least the following elements: $name, $fitness, $value, $numberOfSolutions, $genotype, $phenotype, $phenotypeValue.
lF	Local function configuration. It must contain • `lF$penv$globalOptimum()` which returns the global optimum. • `lF$TerminationEps()` which specifies the the fraction of the global optimum used for computing the upper and lower bounds for the interval in which the best current solution must be for terminating the algorithm.

Details

Useful for benchmark functions with known global optima. Note that for a global optimum of `0` this function terminates if the current optimum is between `0-terminationEps` and `0+terminationEps`.

Value

Boolean.

See Also

Other Termination Condition: `terminateAbsoluteError()`, `terminateGEQ()`, `terminateLEQ()`, `terminatePAC()`, `terminateRelativeError()`, `terminatedFalse()`

Examples

```
parm<-function(x){function() {return(x)}}
olst<-list(); olst$value<-0
penv<-list(); penv$globalOptimum<-parm(olst)
lF<-list(); lF$penv<-penv; lF$TerminationEps<-parm(1.2);lF$Max<-parm(1.0)
solution<-list(); solution$genotype<-list();  solution$genotype$fit<-0.5
terminateRelativeErrorZero(solution, lF)
solution<-list(); solution$genotype<-list();  solution$genotype$fit<-9.6
terminateRelativeErrorZero(solution, lF)
```

`TerminationFactory` *Configure the termination condition(s) a genetic algorithm.*

Description

`TerminationFactory()` implements the selection of a termination method.

Current support:

1. "NoTermination" returns `terminatedFalse`. (Default)
2. "AbsoluteError" returns `terminateAbsoluteError()`. For benchmark functions with known global optima. Termination condition is fulfilled if the current best solution is in the interval from (globalOptimum-eps) to (globalOptimum+eps).
3. "RelativeError" returns `terminateRelativeError()`. For benchmark functions with known global optima. Termination condition is fulfilled if the current best solution is in the interval from (globalOptimum-(globalOptimum*eps)) to (globalOptimum+(globalOptimum*eps)). Does not specify an interval if globalOptimum is zero.
4. "RelativeErrorZero" returns `terminateRelativeErrorZero()`. For benchmark functions with known global optima. Termination condition is fulfilled if the current best solution is in the interval from (globalOptimum-(globalOptimum*eps)) to (globalOptimum+(globalOptimum*eps)). If the globalOptimum is zero, the interval is from `-terminationEps` to `terminationEps`.
5. "PAC" returns `terminatePAC()`. Terminates, as soon as the fitness is is better than a confidence interval depending on the mean and `stats::qnorm(PACdelta, lower.tail=FALSE)` times the standard deviation of the fitness of the initial population.
6. "GEQ" returns `terminateGEQ()`. Terminates as soon as the phenotype value of the solution is greater equal than `lF$TerminationThreshol()`.
7. "LEQ" returns `terminateLEQ()`. Terminates as soon as the phenotype value of the solution is less equal than `lF$TerminationThreshol()`.

Usage

```
TerminationFactory(method = "NoTermination")
```

Arguments

`method`	A string specifying the termination condition.

Value

A boolean function implementing the termination condition.

See Also

Other Configuration: `AcceptFactory()`, `ApplyFactory()`, `CoolingFactory()`, `CrossRateFactory()`, `MutationRateFactory()`, `checkTerminationFactory()`, `xegaConfiguration()`, `xegaEvalPopulationFactory()`

TrigonometricAdditiveCooling

Trigonometric additive cooling.

Description

This schedule decreases in proportion to the cosine of the temperature cycle in `lF$Generations()` (= number of generations) fractions between the starting temperature `lF$Temp0()` and the final temperature `lF$TempN()`.

Usage

```
TrigonometricAdditiveCooling(k, lF)
```

Arguments

`k`	Number of steps (time).
`lF`	Local configuration.

Details

Temperature is updated at the end of each generation in the main loop of the genetic algorithm. `lF$Temp0()` is the starting temperature. `lF$TempN()` is the final temperature. `lF$Generations()` is the number of generations (time).

Value

Temperature at time k.

References

The-Crankshaft Publishing (2023) A Comparison of Cooling Schedules for Simulated Annealing. <https://what-when-how.com/artificial-intelligence/a-comparison-of-cooling-schedules-for-simulated-annealing-artificial-intelligence/>

See Also

Other Cooling: `ExponentialAdditiveCooling()`, `ExponentialMultiplicativeCooling()`, `LogarithmicMultiplicati`
`PowerAdditiveCooling()`, `PowerMultiplicativeCooling()`

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list(Temp0=parm(100), TempN=parm(10), Generations=parm(50))
TrigonometricAdditiveCooling(0, lF)
TrigonometricAdditiveCooling(2, lF)
```

`xegaBestGeneInPopulation` *Extracts indices of best genes in population.*

Description

`xegaBestGeneInPopulation()` extracts the indices of the best genes in the population.

Usage

```
xegaBestGeneInPopulation(fit)
```

Arguments

`fit`	Fitness vector of a population of genes.

Details

You might use: `which(max(fit)==fit)`. But this is slower!

Value

List of the indices of the best genes in the population.

See Also

Other Population Layer: `xegaBestInPopulation()`, `xegaEvalPopulation()`, `xegaInitPopulation()`, `xegaLogEvalsPopulation()`, `xegaNextPopulation()`, `xegaObservePopulation()`, `xegaRepEvalPopulation()`, `xegaSummaryPopulation()`

Examples

```
pop10<-xegaInitPopulation(10, lFxegaGaGene)
epop10<-xegaEvalPopulation(pop10, lFxegaGaGene)
xegaBestGeneInPopulation(epop10$fit)
```

`xegaBestInPopulation` *Best solution in the population.*

Description

`xegaBestInPopulation()` extracts the best individual of a population and reports fitness, value, genotype, and phenotype:

1. `fitness`: The fitness value of the genetic algorithm.
2. `value`: The function value of the problem environment.
3. `genotype`: The gene representation.
4. `phenotype`: The problem representation. E.g. a parameter list, a program, ...

We report one of the best solutions.

Usage

```
xegaBestInPopulation(pop, fit, lF, allsolutions = FALSE)
```

Arguments

`pop`	Population of genes.
`fit`	Vector of fitness values of pop.
`lF`	Local function configuration.
`allsolutions`	If TRUE, also return a list of all solutions.

Value

Named list with the following elements:

- `$name`: The name of the problem environment.
- `$fitness`: The fitness value of the best solution.
- `$value`: The evaluated best gene.
- `$numberOfSolutions`: The number of solutions.
- `$genotype`: The best gene.
- `$phenotype`: The parameters of the solution (the decoded gene).
- `$phenotypeValue`: The value of the function of the parameters of the solution (the decoded gene).
- `$allgenotypes`: The genotypes of all best solutions. (allsolutions==TRUE)
- `$allphenotypes`: The phenotypes of all best solutions. (allsolutions==TRUE)

See Also

Other Population Layer: xegaBestGeneInPopulation(), xegaEvalPopulation(), xegaInitPopulation(), xegaLogEvalsPopulation(), xegaNextPopulation(), xegaObservePopulation(), xegaRepEvalPopulation(), xegaSummaryPopulation()

Examples

```
pop10<-xegaInitPopulation(10, lFxegaGaGene)
epop10<-xegaEvalPopulation(pop10, lFxegaGaGene)
xegaBestInPopulation(epop10$pop, epop10$fit, lFxegaGaGene)
```

xegaConfiguration *Remembers R command command with which algorithm has been called.*

Description

`xegaConfiguration()` returns the command with which the genetic algorithm has been called. For replicating computational experiments with genetic algorithms.

Usage

```
xegaConfiguration(GAname, penv, grammar, env)
```

Arguments

`GAname`	Name of genetic algorithm's main function. (Currently: "Run").
`penv`	The expression for the problem environment `penv`. Use: `substitute(penv)`.
`grammar`	The grammar `grammar`. Use: `substitute(grammar)`.
`env`	Environment with variable value bindings. Use: `environment()`.

Value

A named list with the following elements:

- `$GAconf`: A text string with the call of the genetic algorithm (the function we want to capture the call).
- `$GAenv`: The environment with the arguments bound to the values when the genetic algorithm was called.

Warning

- $GAenv is correct only for simple arguments (strings or numbers) not for complex objects like problem environments.
- `future.apply::future_lapply()` is configured by a plan statement which must be issued before calling the genetic algorithm. At the moment, the plan chosen is not remembered.

See Also

Other Configuration: AcceptFactory(), ApplyFactory(), CoolingFactory(), CrossRateFactory(), MutationRateFactory(), TerminationFactory(), checkTerminationFactory(), xegaEvalPopulationFactory()

Examples

```
GA<-function(pe, grammar=NULL, nope=1.5, sle="test", ok=TRUE)
{xegaConfiguration("GA", substitute(pe), substitute(grammar), environment())}
Para<-5
GA(Para)
Cube<-7
GA(Cube, 2, 3, 4)
```

---

xegaEvalPopulation *Evaluates a population of genes in a problem environment*

Description

`xegaEvalPopulation()` evaluates a population of genes in a problem environment.

Usage

```
xegaEvalPopulation(pop, lF)
```

Arguments

pop	Population of genes.
lF	Local function configuration.

Details

Parallelization of the evaluation of fitness functions is possible by defining `lF$lapply`.

Value

List of

- `$pop` gene vector,
- `$fit` fitness vector,
- `$evalFail` number of failed evaluations.

See Also

Other Population Layer: xegaBestGeneInPopulation(), xegaBestInPopulation(), xegaInitPopulation(), xegaLogEvalsPopulation(), xegaNextPopulation(), xegaObservePopulation(), xegaRepEvalPopulation(), xegaSummaryPopulation()

Examples

```
pop10<-xegaInitPopulation(10, lFxegaGaGene)
lFxegaGaGene[["lapply"]]<-ApplyFactory(method="Sequential")
result<-xegaEvalPopulation(pop10, lFxegaGaGene)
```

---

xegaEvalPopulationFactory

Configures the evaluation of the population of a genetic algorithm.

Description

`xegaEvalPopulationFactory()` implements the selection of the evaluation function for the population of a genetic algorithm.

Current support:

1. "EvalPopulation" returns `xegaEvalPopulation`. (Default)
2. "RepEvalPopulation" returns `xegaReplEvalPopulation`. For stochastic functions. Needs `lF$rep()` for the number of repetitions and `lF$apply()` for the (parallel) apply function.

Usage

```
xegaEvalPopulationFactory(method = "EvalPopulation")
```

Arguments

`method`	A string specifying the termination condition.

Value

A boolean function implementing the termination condition.

See Also

Other Configuration: `AcceptFactory()`, `ApplyFactory()`, `CoolingFactory()`, `CrossRateFactory()`, `MutationRateFactory()`, `TerminationFactory()`, `checkTerminationFactory()`, `xegaConfiguration()`

xegaInitPopulation *Initializes a population of genes.*

Description

`xegaInitPopulation()` initializes a population of genes.

Usage

```
xegaInitPopulation(popsize, lF)
```

Arguments

`popsize`	Population size.
`lF`	Local function configuration.

Value

List of genes.

See Also

Other Population Layer: `xegaBestGeneInPopulation()`, `xegaBestInPopulation()`, `xegaEvalPopulation()`, `xegaLogEvalsPopulation()`, `xegaNextPopulation()`, `xegaObservePopulation()`, `xegaRepEvalPopulation()`, `xegaSummaryPopulation()`

Examples

```
pop10<-xegaInitPopulation(10, lFxegaGaGene)
```

xegaLogEvalsPopulation *Combine fitness, generations, and the phenotype of the gene.*

Description

Combine fitness, generations, and the phenotype of the gene.

Usage

```
xegaLogEvalsPopulation(pop, evallog, generation, lF)
```

Arguments

`pop`	Population.
`evallog`	Evaluation log.
`generation`	Generation logged.
`lF`	Local function configuration.

Value

Update of the evaluation log. The evaluation log is a list of decoded and evaluated genes. A list item of the evaluation log has the following elements:

- `$generation`: The generation.
- `$fit`: The fitness value.
- `$sigma`: The standard deviation of the fitness value, if it exists. Default: `0`.
- `$obs`: The number of observations for computing the fitness value, if it exists. Default: `0`.
- `$phenotype`: The phenotype of the gene.

See Also

Other Population Layer: `xegaBestGeneInPopulation()`, `xegaBestInPopulation()`, `xegaEvalPopulation()`, `xegaInitPopulation()`, `xegaNextPopulation()`, `xegaObservePopulation()`, `xegaRepEvalPopulation()`, `xegaSummaryPopulation()`

Examples

```
pop10<-xegaInitPopulation(10, lFxegaGaGene)
epop10<-xegaEvalPopulation(pop10, lFxegaGaGene)
logevals<-list()
logevals
logevals<-xegaLogEvalsPopulation(epop10$pop, logevals, 1, lFxegaGaGene)
logevals
```

`xegaNextPopulation` *Computes the next population of genes.*

Description

`xegaNextPopulation()` builds the next population by repeatedly calling `ReplicateGene()`.

Usage

```
xegaNextPopulation(pop, fit, lF)
```

Arguments

pop	Population of genes.
fit	Fitness.
lF	Local configuration.

Details

The current version is sequential. For parallelization, a restructuring of the main loop with an integration of `xegaNextPopulation` and `xegaEvalPopulation` is planned, because this allows the parallelization of a large part of the genetic operations which are sequential in the current version.

Value

Population of genes.

See Also

Other Population Layer: `xegaBestGeneInPopulation()`, `xegaBestInPopulation()`, `xegaEvalPopulation()`, `xegaInitPopulation()`, `xegaLogEvalsPopulation()`, `xegaObservePopulation()`, `xegaRepEvalPopulation()`, `xegaSummaryPopulation()`

Examples

```
lFxegaGaGene$MutationRate<-MutationRateFactory(method="Const")
lFxegaGaGene$ReplicateGene<-ReplicateGene
lFxegaGaGene$Accept<-AcceptFactory(method="All")
pop10<-xegaInitPopulation(10, lFxegaGaGene)
epop10<-xegaEvalPopulation(pop10, lFxegaGaGene)
newpop<-xegaNextPopulation(epop10$pop, epop10$fit, lFxegaGaGene)
```

`xegaObservePopulation` *Observe summary statistics of the fitness of the population.*

Description

`xegaObservePopulation()` reports summary statistics of the fitness of the population.

Usage

```
xegaObservePopulation(fit, v = vector())
```

Arguments

fit	Vector of fitness values of a population.
v	Vector of population statistic vectors.

Details

Population statistics are used for

- implementing individually variable operator rates and
- visualizing the progress of the algorithm.

Value

Vector of population statistics. If position x modulo 8 equals

1. 1: Mean fitness.
2. 2: Min fitness.
3. 3: Lower-hinge (approx. 1st quartile) of fitness.
4. 4: Median fitness.
5. 5: Upper-hinge (approx. 3rd quartile) of fitness.
6. 6: Max fitness.
7. 7: Variance.
8. 8: Mean absolute deviation.

See Also

Other Population Layer: `xegaBestGeneInPopulation()`, `xegaBestInPopulation()`, `xegaEvalPopulation()`, `xegaInitPopulation()`, `xegaLogEvalsPopulation()`, `xegaNextPopulation()`, `xegaRepEvalPopulation()`, `xegaSummaryPopulation()`

Examples

```
pop10<-xegaInitPopulation(10, lFxegaGaGene)
epop10<-xegaEvalPopulation(pop10, lFxegaGaGene)
popStats<-xegaObservePopulation(epop10$fit)
popStats<-xegaObservePopulation(epop10$fit, popStats)
matrix(popStats, ncol=8, byrow=TRUE)
```

`xegaPopulation` *Package xegaPopulation.*

Description

Population level functions

Details

The `xegaPopulation` package provides the representation independent functions of the population level of the simple genetic algorithm xegaX packages:

- File xegaPopulation.R:
  - Initializing a population of genes.
  - Getting the indices of the best genes in a population of genes for getting the best solution(s) in a population of genes.
  - Configurable summary report of population fitness statistics.
  - Observation of the summary statistics of a population of genes.
  - Logging of the phenotype and the value of the phenotype.
- File xegaNextPopulation.R:
  - Computation of the next population of genes.
  - Evaluation of the next population of genes.

  Future: Improved support for parallelization suggests a different division of labor:
  - Construct a list of abstract task descriptions with one element per gene.
  - Provide for a parallel execution of these task descriptions. This requires changes in the structuring of the operator pipelines and the replicate gene functions for the different gene representations and algorithms.
  - Performance improvement depends on the gene representation and on the use of function evaluations in the genetic machinery. For example, for the TSP problem, function evaluations are embedded into most of the mutation operators.
- File acceptance.R: Acceptance rules for new genes and a function factory for configuring them.
- File cooling.R: Cooling schedules for temperature reduction.
- File localAdaptivity.R: Unused. Move to gene dependent packages planned.
- File adaptivityCrossover.R: Functions constant and adaptive crossover rates.
- File adaptivityMutation.R: Functions constant and adaptive mutation rates.
- File parModel.R: Execution models for parallelization.
  - "Sequential": Configures lapply as `lapply()`.
  - "MultiCore": Configures lapply as `parallel::mclapply()`. The number of cores is set by `lF$Core()`.
- File configuration.R: Documenting how the algorithm was called. Support for the replication of computational experiments (replicate and replay).

Interface of Acceptance Rules

```
newGene<-accept(OperatorPipeline, gene, lF)
```

1. Accept all new genes: Identity function. For genetic algorithms.
2. Accept best: Accepts the gene with the highest fitness. For greedy and randomized greedy algorithms (hill-climbing algorithms).

3. The Metropolis and the individually variable Metropolis rule: If the new gene gene is better, accept it. If the old gene is better, make a biased random choice. The probability of accepting a decrease in fitness depends on the fitness distance between genes, a constant beta for scaling the exponential decay and a temperature parameter and for the individually variable Metropolis rule a correction term which depends on the distance to the best known fitness of the run.

Constants for Acceptance Rules.

Constant	Default	Used in
lF$Beta()	?	AcceptMetropolis()
		AcceptIVMetropolis()
lF$TempK()	?	AcceptMetropolis()
		AcceptIVMetropolis()
lF$lFCBestFitness()	None	AcceptIVMetropolis()

Interface of Cooling Schedules

```
Temperature<-cooling(k, lF)
```

Cooling schedules convert the progress of the time in the algorithm (measured in generations) into a temperature. The temperature influences the probability of accepting a gene with less fitness than its parent gene.

Constants for Cooling Schedules.

Constant	Default	Used in
lF$Alpha()	?	ExponentialMultiplicativeCooling()
	?	LogarithmicMultiplicativeCooling()
	?	PowerMultiplicativeCooling()
lF$Temp0()	?	ExponentialMultiplicativeCooling()
	?	LogarithmicMultiplicativeCooling()
	?	PowerMultiplicativeCooling()
	?	PowerAdditiveCooling()
	?	ExponentialAdditiveCooling()
	?	TrigonometricAdditiveCooling()
lF$TempN()	?	PowerAdditiveCooling()
	?	ExponentialAdditiveCooling()
	?	TrigonometricAdditiveCooling()
lF$CoolingPower()	?	PowerMultiplicativeCooling()
	?	PowerAdditiveCooling()
lF$Generations()		PowerAdditiveCooling()
		ExponentialAdditiveCooling()
	?	TrigonometricAdditiveCooling()

Interface of Rates

```
rate<-rateFunction(fit, lF)
```

Crossover and mutation rate functions may be adaptive. The interface allows for dependencies of the rate on fitness and constants in the local configuration.

Constants for Adaptive Crossover and Mutation Rates

Constant	**Default**	**Used in**
lF$CrossRate1()	?	IACRate()
lF$CrossRate2()	?	IACRate()
lF$MutationRate1()		IAMRate()
lF$MutationRate2()		IAMRate()
lF$CutoffFit()	?	IACRate()
lF$CBestFitness()		IACRate()
		IAMRate()

Interface of Termination Conditions

`hasTerminated<-Terminate(solution, lF)`

The interface allows the specification of termination conditions for the genetic algorithm. The abstract function `Terminate` returns a boolean value. TBD

Dependencies of Termination Conditions

Condition	**Requires**
terminatedFalse()	-
terminateAbsoluteError()	lF$penv$globalOptimum()
	lF$TerminationEps
terminateRelativeError()	lF$penv$globalOptimum()
	lF$TerminationEps

The Architecture of the xegaX-Packages

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

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

  - xegaPermGene for permutation-based genetic algorithms.
  - xegaDfGene for derivation free algorithms as e.g. differential evolution.
  - xegaGpGene for grammar-based genetic algorithms.
  - xegaGeGene for grammatical evolution algorithms.

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

Copyright

(c) 2023 Andreas Geyer-Schulz

License

MIT

URL

https://github.com/ageyerschulz/xegaPopulation

Installation

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

Author(s)

Andreas Geyer-Schulz

See Also

Useful links:

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

---

xegaRepEvalPopulation *Evaluates a population of genes in a a problem environment repeatedly.*

Description

`xegaRepEvalPopulation()` evaluates a population of genes in a problem environment `lF$rep` times. The results of repeatedly evaluating a gene are aggregated:

- `gene$fit` is the mean fitness,
- `gene$var` is the fitness variance,
- `gene$std` is the standard deviation of the fitness, and
- `gene$obs` is the number of repetitions.

Usage

```
xegaRepEvalPopulation(pop, lF)
```

Arguments

`pop`	Population of genes.
`lF`	Local function configuration.

Details

Parallelization of the evaluation of fitness functions is possible by defining `lF$lapply`.

Value

List of

- `$pop` gene vector,
- `$fit` fitness vector,
- `$evalFail` number of failed evaluations.

See Also

Other Population Layer: `xegaBestGeneInPopulation()`, `xegaBestInPopulation()`, `xegaEvalPopulation()`, `xegaInitPopulation()`, `xegaLogEvalsPopulation()`, `xegaNextPopulation()`, `xegaObservePopulation()`, `xegaSummaryPopulation()`

Examples

```
    parm<-function(x){function() {return(x)}}
pop10<-xegaInitPopulation(10, lFxegaGaGene)
lFxegaGaGene[["lapply"]]<-ApplyFactory(method="Sequential")
lFxegaGaGene[["rep"]]<-parm(3)
result<-xegaRepEvalPopulation(pop10, lFxegaGaGene)
```

`xegaSummaryPopulation` *Provide elementary summary statistics of the fitness of the population.*

Description

`SummaryPopulation()` reports on the fitness and the value of the best solution in the population.

The value of `lF$Verbose()` 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 1 solution.

Usage

```
xegaSummaryPopulation(pop, fit, lF, iter = 0)
```

Arguments

`pop`	Population of genes.
`fit`	Vector of fitness values of pop.
`lF`	Local function configuration.
`iter`	The generation. Default: `0`.

Value

The number `0`.

See Also

Other Population Layer: `xegaBestGeneInPopulation()`, `xegaBestInPopulation()`, `xegaEvalPopulation()`, `xegaInitPopulation()`, `xegaLogEvalsPopulation()`, `xegaNextPopulation()`, `xegaObservePopulation()`, `xegaRepEvalPopulation()`

Examples

```
pop10<-xegaInitPopulation(10, lFxegaGaGene)
epop10<-xegaEvalPopulation(pop10, lFxegaGaGene)
rc<-xegaSummaryPopulation(epop10$pop, epop10$fit, lFxegaGaGene, iter=12)
```

Index