Package ‘xegaSelectGene’

February 2, 2024

Title Selection of Genes and Gene Representation Independent Functions

Version 1.0.0.0

Description This collection of gene representation-independent
mechanisms for evolutionary and genetic algorithms contains
four groups of functions:
First, functions for selecting a gene
in a population of genes according to its fitness value
and for adaptive scaling of the fitness values as well as for
performance optimization and measurement offer several
variants for implementing the survival of the fittest.
Second, evaluation functions for
deterministic functions avoid recomputation.
Evaluation of stochastic functions incrementally improve
the estimation of the mean and variance of fitness values
at almost no additional cost. Evaluation functions
for gene repair handle error-correcting decoders.
Third, timing and counting functions for profiling the
algorithm pipeline are provided to assess bottlenecks in
the algorithms.
Fourth, a small collection of problem environments
for function optimization, combinatorial optimization, and
grammar-based genetic programming and grammatical evolution
is provided for tutorial examples.
The methods in the package are described by the following
references:
Baker, James E. (1987, ISBN:978-08058-0158-8),
De Jong, Kenneth A. (1975)
<`https://deepblue.lib.umich.edu/handle/2027.42/4507`>,
Geyer-Schulz, Andreas (1997, ISBN:978-3-7908-0830-X),
Grefenstette, John J. (1987, ISBN:978-08058-0158-8),
Grefenstette, John J. and Baker, James E.
(1989, ISBN:1-55860-066-3),
Holland, John (1975, ISBN:0-472-08460-7),
Lau, H. T. (1986) <doi:10.1007/978-3-642-61649-5>,
Price, Kenneth V., Storn, Rainer M. and Lampinen, Jouni A. (2005)
<doi:10.1007/3-540-31306-0>,

Reynolds, J. C. (1993) <doi:10.1007/BF01019459>,
Schaffer, J. David (1989, ISBN:1-55860-066-3),
Wenstop, Fred (1980) <doi:10.1016/0165-0114(80)90031-7>,
Whitley, Darrell (1989, ISBN:1-55860-066-3),
Wickham, Hadley (2019, ISBN:978-815384571).

License MIT + file LICENSE

URL <`https://github.com/ageyerschulz/xegaSelectGene`>

Encoding UTF-8

RoxygenNote 7.2.3

Suggests testthat (>= 3.0.0)

Collate 'evalGene.R' 'scaling.R' 'selectGene.R'
'selectGeneBenchmark.R' 'timer.R' 'DeJongF4.R' 'Parabola2D.R'
'newXOR.R' 'newTSP.R' 'xegaSelectGene-package.R'

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 2024-02-02 19:40:02 UTC

R topics documented:

`ContinuousScaleFitness` *Dispersion Ratio Based Fitness Scaling.*

Description

Dispersion Ratio Based Fitness Scaling.

Usage

```
ContinuousScaleFitness(fit, lF)
```

Arguments

`fit`	A fitness vector.
`lF`	Local configuration.

Value

Scaled fitness vector.

See Also

Other Scaling: `DispersionRatio()`, `ScaleFitness()`, `ScalingFitness()`, `ThresholdScaleFitness()`

Other Adaptive Parameter: `ThresholdScaleFitness()`

Examples

```
lF<-list()
lF$Offset<-parm(0.0001)
lF$RDMWeight<-parm(2)
lF$RDM<-parm(1.2)
fit<-sample(10, 20, replace=TRUE)
fit
ContinuousScaleFitness(fit, lF)
```

`Counted` *Transformation into a counted function*

Description

`Counted` takes two functions as arguments: The function whose call frequency should be measured and a counter object created by `newCounter()`. It returns a counted function.

Usage

```
Counted(FUN, counter)
```

Arguments

`FUN`	A function whose run time should be measured.
`counter`	A counter generated by `newCounter()`.

Value

A counted function.

See Also

Other Performance Measurement: `Timed()`, `newCounter()`, `newTimer()`

Examples

```
test<-function(v) {sum(v)}
testCounter<-newCounter()
testCounted<-Counted(test, testCounter)
testCounter("Show")
testCounted(sample(10,10)); testCounted(sample(10,10))
testCounter("Show")
```

`DeJongF4Factory` *Factory for function F4 (30-dimensional quartic with noise)*

Description

This function factory sets up the problem environment for De Jong's function F4. F4 is a 30-dimensional quartic function with Gaussian noise. It is a continuous, convex, unimodal, high-dimensional quartic function with Gaussian noise. For validation, $\epsilon = 3 * \sigma$ will work most of the time. Note: There exist 2^{30} maxima (without noise)!

Usage

```
DeJongF4Factory()
```

Value

A problem environment represented as a list of functions:

- `$name()`: The name of the problem environment.
- `$bitlength()`: The vector of the number of bits of each parameter of the function.
- `$genelength()`: The number of bits of the gene.
- `$lb()`: The vector of lower bounds of the parameters.
- `$ub()`: The vector of upper bounds of the parameters.
- `$f(parm, gene=0, lF=0))`: The fitness function.

Additional elements:

- `$describe()`: Print a description of the problem environment to the console.
- `$solution()`: The solution structure. A named list with `minimum`, `maximum` and 2 lists of equivalent solutions: `minpoints`, `maxpoints`.

References

De Jong, Kenneth A. (1975): *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, Michigan, Ann Arbor, pp. 203-206. <https://deepblue.lib.umich.edu/handle/2027.42/4507>

See Also

Other Problem Environments: `DelayedPFactory()`, `Parabola2DEarlyFactory()`, `Parabola2DErrFactory()`, `Parabola2DFactory()`, `envXOR`, `lau15`, `newEnvXOR()`, `newTSP()`

Examples

```
DeJongF4<-DeJongF4Factory()
DeJongF4$name()
DeJongF4$bitlength()
DeJongF4$genelength()
DeJongF4$lb()
DeJongF4$ub()
DeJongF4$f(c(2.01, -1.05, 4.95, -4.3, -3.0))
DeJongF4$f(c(2.01, -1.05, 4.95, -4.3, -3.0))
DeJongF4$describe()
DeJongF4$solution()
```

`DelayedPFactory` *Factory for a 2-dimensional quadratic parabola with delayed execution.*

Description

This list of functions sets up the problem environment for a 2-dimensional quadratic parabola with a delayed execution of 0.1s. This function aims to test strategies of distributed/parallel execution of functions.

Usage

```
DelayedPFactory()
```

Details

The factory contains examples of all functions which form the interface of a problem environment to the simple genetic algorithm with binary-coded genes of package `xega`.

Value

A problem environment represented as a list of functions:

- `$name()`: The name of the problem environment.
- `$bitlength()`: The vector of the number of bits of each parameter of the function.
- `$genelength()`: The number of bits of the gene.
- `$lb()`: The vector of lower bounds of the parameters.
- `$ub()`: The vector of upper bounds of the parameters.
- `$f(parm, gene=0, lF=0))`: The fitness function.

Additional elements:

- `$describe()`: Print a description of the problem environment to the console.
- `$solution()`: The solution structure. A named list with `minimum`, `maximum` and 2 lists of equivalent solutions: `minpoints`, `maxpoints`.

See Also

Parabola2D

Other Problem Environments: `DeJongF4Factory()`, `Parabola2DEarlyFactory()`, `Parabola2DErrFactory()`, `Parabola2DFactory()`, `envXOR`, `lau15`, `newEnvXOR()`, `newTSP()`

Examples

```
DelayedP<-DelayedPFactory()
DelayedP$f(c(2.2, 1.0))
```

`DispersionMeasureFactory` *Configure dispersion measure.*

Description

`DispersionMeasureFactory` returns a function for the dispersion measure as specified by a label. If an invalid label is specified, the configuration fails.

Usage

```
DispersionMeasureFactory(method = "var")
```

Arguments

`method`	A dispersion measure. • "var": Variance (Default). • "std": Standard deviation. • "mad": Median absolute deviation (`mad(vec, constant=1)`). • "cv": Coefficient of variation". • "range": Range. • "iqr": Inter quartile range (approximated by the lower and upper hinge of `fivenum`). If an invalid label is specified, the configuration fails.

Value

A function which computes the dispersion measure from the vector of population statistics produced by `xegaObservePopulation` of package `xegaPopulation`.

See Also

Other Configuration: `EvalGeneFactory()`, `ScalingFactory()`, `SelectGeneFactory()`

Examples

```
require(stats)
fit<-sample(30, 20, replace=TRUE)
populationStats<-c(mean(fit), fivenum(fit), var(fit), mad(fit, constant=1))
dm<-DispersionMeasureFactory("var")
dm(populationStats)
dm<-DispersionMeasureFactory("range")
dm(populationStats)
```

`DispersionRatio` *Dispersion Ratio*

Description

The dispersion ratio is computed as the ratio `DM(t)/DM(k)` where `DM(t)` is the dispersion measure of period t and `DM(k)` the dispersion measure of period `max(1, (t-k))`. `k` is specified by `lF$ScalingDelay`.

Usage

```
DispersionRatio(popStat, DM, lF)
```

Arguments

`popStat`	Population statistics.
`DM`	Dispersion function.
`lF`	Local configuration.

Details

The dispersion ratio may take unreasonably high and low values leading to numerical underflow or overflow of fitness values. Therefore, we use hard thresholding to force the dispersion ratio into the interval `[lF$DRmin(), lF$DRmax()]`. The default interval is `[0.5, 2.0]`.

Value

Dispersion ratio.

See Also

Other Scaling: `ContinuousScaleFitness()`, `ScaleFitness()`, `ScalingFitness()`, `ThresholdScaleFitness()`

Examples

```
p<-matrix(0, nrow=3, ncol=8)
p[1,]<-c(14.1,  0.283,  5.53, 14.0, 19.4, 38.1, 90.2, 6.54)
p[2,]<-c(20.7,  0.794, 14.63, 19.0, 26.5, 38.8, 71.4, 5.27)
p[3,]<-c(24.0,  6.007, 16.89, 24.1, 29.2, 38.8, 73.4, 6.50)
F<-list()
F$ScalingDelay<-function() {1}
F$DRmax<-function() {2.0}
F$DRmin<-function() {0.5}
dm<-DispersionMeasureFactory("var")
DispersionRatio(p, dm, F)
F$ScalingDelay<-function() {2}
DispersionRatio(p, dm, F)
```

`envXOR` *The problem environment* `envXOR` *for programming the XOR function either by grammar-based genetic programming or grammatical evolution.*

Description

The problem environment `envXOR` is a list with the following elements:

- `envXOR$name`: `"envXOR"`, the name of the problem environment.
- `envXOR$buildtest(expr)`: The function which builds the environment for evaluating the expression with a binding of the variables to the parameters.
- `envXOR$TestCases`: The truth table of the XOR function.
- `envXOR$f(expr, gene=NULL, lF=NULL)`: The fitness function. `expr` is the string with the logical expression to be evaluated.

Usage

```
envXOR
```

Format

An object of class `list` of length 4.

See Also

Other Problem Environments: `DeJongF4Factory()`, `DelayedPFactory()`, `Parabola2DEarlyFactory()`, `Parabola2DErrFactory()`, `Parabola2DFactory()`, `lau15`, `newEnvXOR()`, `newTSP()`

`EvalGene` *Evaluate a gene*

Description

`EvalGene` is the abstract function which evaluates a gene.

Usage

```
EvalGene(gene, lF)
```

Arguments

`gene`	A gene (representation independent).
`lF`	The local configuration (a function factory provided by the `xegaX` package).

Details

For minimization problems, the fitness value must be multiplied by -1. The constant function `lF$Max()` returns 1 for a maximization and -1 for a minimization problem.

Value

A gene.

See Also

Other Evaluation Functions: `EvalGeneDet()`, `EvalGeneR()`, `EvalGeneStoch()`, `EvalGeneU()`

Examples

```
DeJongF4<-DeJongF4Factory()
lF<-NewlFevalGenes(DeJongF4)
EvalGene<-EvalGeneFactory()
g1<-list(evaluated=FALSE, fit=0, gene1=c(1.0, -1.5))
g1
g2<-EvalGene(g1, lF)
g2
```

`EvalGeneDet` *Evaluates a gene in a deterministic problem environment.*

Description

`EvalGeneDet` evaluates a gene in a problem environment if it has not been evaluated yet. The repeated evaluations of a gene are omitted.

Usage

```
EvalGeneDet(gene, lF)
```

Arguments

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

Details

If the evaluation of the fitness function of the problem environment fails, we catch the error and return `NA`.

Value

A gene (with `$evaluated==TRUE`).

See Also

Other Evaluation Functions: `EvalGeneR()`, `EvalGeneStoch()`, `EvalGeneU()`, `EvalGene()`

Examples

```
Parabola2D<-Parabola2DFactory()
lF<-NewlFevalGenes(Parabola2D)
g1<-list(evaluated=FALSE, fit=0, gene1=c(1.0, -1.5))
g2<-list(evaluated=FALSE, fit=0, gene1=c(0.0, 0.0))
g1a<-EvalGeneDet(g1, lF)
EvalGeneDet(g1a, lF)
g2a<-EvalGeneDet(g2, lF)
EvalGeneDet(g2a, lF)
```

EvalGeneFactory — *Configure the evaluation function of a genetic algorithm.*

Description

EvalGeneFactory implements the selection of one of the evaluation functions for a gene 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.

Usage

```
EvalGeneFactory(method = "EvalGeneU")
```

Arguments

method	Available methods are: • "EvalGeneU": Evaluate gene (Default). Function EvalGeneU. • "EvalGeneR": If the gene has been repaired by a decoder, the gene is replaced by the repaired gene. Function EvalGeneR. • "Deterministic": A gene which has been evaluated is not reevaluated. Function EvalGeneDet. • "Stochastic": The fitness mean and the fitness variance are incrementally updated. Genes remaining in the population over several generations, the fitness mean converges to the expected mean. Function EvalGeneStoch.

Value

An evaluation function.

See Also

Other Configuration: DispersionMeasureFactory(), ScalingFactory(), SelectGeneFactory()

Examples

```
set.seed(5)
DeJongF4<-DeJongF4Factory()
lF<-NewlFevalGenes(DeJongF4)
EvalGene<-EvalGeneFactory("EvalGeneU")
g1<-list(evaluated=FALSE, evalFail=FALSE, fit=0, gene1=c(1.0, -1.5))
g1
g2<-EvalGene(g1, lF)
g2
EvalGene<-EvalGeneFactory("Deterministic")
g3<-EvalGene(g2, lF)
g3
set.seed(5)
EvalGene<-EvalGeneFactory("Stochastic")
```

```
g1<-list(evaluated=FALSE, evalFail=FALSE, fit=0, gene1=c(1.0, -1.5))
g1
g2<-EvalGene(g1, lF)
g2
g3<-EvalGene(g2, lF)
g3
```

EvalGeneR *Evaluates a repaired gene in a problem environment.*

Description

`EvalGeneR` evaluates a repaired gene in a problem environment.

Usage

```
EvalGeneR(gene, lF)
```

Arguments

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

Details

If the decoder repairs a gene, the repaired gene must replace the original gene.

Value

A gene (with `$evaluated==TRUE`).

See Also

EvalGeneU

Other Evaluation Functions: `EvalGeneDet()`, `EvalGeneStoch()`, `EvalGeneU()`, `EvalGene()`

Examples

```
Parabola2D<-Parabola2DFactory()
lF<-NewlFevalGenes(Parabola2D)
g1<-list(evaluated=FALSE, fit=0, gene1=c(1.0, -1.5, 3.37))
g2<-list(evaluated=FALSE, fit=0, gene1=c(0.0, 0.0, 0.0))
EvalGeneR(g1, lF)
EvalGeneR(g2, lF)
```

`EvalGeneStoch` *Evaluates a gene in a stochastic problem environment.*

Description

`EvalGeneStoch` evaluates a gene in a stochastic problem environment.

Usage

```
EvalGeneStoch(gene, lF)
```

Arguments

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

Details

In a stochastic problem environment, the expected fitness is maximized. The computation of the expectation is done by incrementally updating the mean. For this, need the number of evaluations of the gene (`$obs` of the gene). In addition, we compute the incremental variance of the expected fitness stored in `$var`. The standard deviation is then `gene$var/gene$obs`.

If the evaluation of the fitness function of the problem environment fails, we catch the error and return `NA` for the first evaluation of the gene. If the gene has been evaluated, we return the old gene.

Value

A gene with the elements

- `$evaluated`: Boolean.
- `$evalFail`: Boolean.
- `$fit`: Mean fitness of gene.
- `$gene1`: Gene.
- `$obs`: Number of evaluations of gene.
- `$var`: Variance of fitness.
- `$sigma`: Standard deviation of fitness.

See Also

Other Evaluation Functions: `EvalGeneDet()`, `EvalGeneR()`, `EvalGeneU()`, `EvalGene()`

Examples

```
DeJongF4<-DeJongF4Factory()
lF<-NewlFevalGenes(DeJongF4)
g1<-list(evaluated=FALSE, evalFail=FALSE, fit=0, gene1=c(1.0, -1.5))
g1
g2<-EvalGeneStoch(g1, lF)
g2
g3<-EvalGeneStoch(g2, lF)
g3
g4<-EvalGeneStoch(g3, lF)
g4
g5<-EvalGeneStoch(g4, lF)
g5
```

`EvalGeneU` *Evaluates a gene in a problem environment*

Description

`EvalGeneU` evaluates a gene in a problem environment.

Usage

```
EvalGeneU(gene, lF)
```

Arguments

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

Details

If the evaluation of the fitness function of the problem environment fails, the following strategy is used: We catch the error and print it, ignore it:

The error handler returns `NA`.

We check for the error and update the gene:

- `$evaluated` TRUE.
- `$evalFail` TRUE.
- `$fit` is set to the minimum fitness in the population.

The boolean function `lF$ReportEvalErrors` controls the output of error messages for evaluation failures. Rationale: In grammatical evolution, the standard approach ignores attempts the evaluate incomplete programs.

Value

A gene (with `$evaluated==TRUE`).

Future improvement

Provide configurable error handlers. Rationale: Make debugging for new problem environments easier. Catch communication problems in distributed/parallel environments.

See Also

Other Evaluation Functions: `EvalGeneDet()`, `EvalGeneR()`, `EvalGeneStoch()`, `EvalGene()`

Examples

```
Parabola2D<-Parabola2DFactory()
lF<-NewlFevalGenes(Parabola2D)
g1<-list(evaluated=FALSE, fit=0, gene1=c(1.0, -1.5, 3.37))
g2<-list(evaluated=FALSE, fit=0, gene1=c(0.0, 0.0, 0.0))
EvalGeneU(g1, lF)
EvalGeneU(g2, lF)
```

`lau15` *The problem environment* `lau15` *for a traveling salesman problem.*

Description

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

The problem environment `lau15` is a list with the following functions:

1. `lau15$name()`: `"lau15"`, the name of the TSP problem environment.
2. `lau15$genelength()`: 15, the number of cities on the round trip.
3. `lau15$dist()`: The distance matrix of the problem.
4. `lau15$cities()`: A list of city names or the vector `1:lau15$genelength()`.
5. `lau15$f (permutation, gene = 0, lF = 0, tour = TRUE)`: The fitness function. Computes the roundtrip for permutation of cities.
6. `lau15$solution()`: 291, the known optimal solution of lau15.
7. `lau15$path()`: The permutation for the optimal roundtrip.
8. `lau15$show(p)`: Prints the roundtrip `p`.
9. `lau15$greedy(startposition, k)`: Computes a path of length `k` starting at `startposition` by choosing the nearest city.
10. `lau15$kBestGreedy(k, tour=TRUE)`: Computes the best greedy path/tour with k cities.
11. `lau15$rnd2Opt(permutation, maxtries=5)`: Tries to find a better permutation by at most 5 random 2-opt heuristics.
12. `lau15$LinKernighan(permutation, maxtries=5)`: A randomized Lin-Kernigan heuristic implemented as a sequence of randomized 2-opt moves.

Usage

```
lau15
```

Format

An object of class `list` of length 12.

References

Lau, H. T. (1986): *Combinatorial Heuristic Algorithms in FORTRAN*. Springer, 1986. p. 61. <doi:10.1007/978-3-642-61649-5>

See Also

Other Problem Environments: `DeJongF4Factory()`, `DelayedPFactory()`, `Parabola2DEarlyFactory()`, `Parabola2DErrFactory()`, `Parabola2DFactory()`, `envXOR`, `newEnvXOR()`, `newTSP()`

`newCounter` *Counter*

Description

`newCounter` sets up a counter object with one internal state variable, namely `count` to count the number of counter calls.

Usage

```
newCounter()
```

Details

Generate a counter: `a<-newCounter()` sets up the counter `a`. The counter `a` supports three methods:

1. `a()` or `a("Measure")` or `a(method="Measure")` **starts** the timer when called 1st, 3rd, 5th, ... time and **stops** the timer when called the 2nd, 4th, 6th, ... time. The calls can be manually inserted before and after a block of R-code for profiling.
2. `a("Count")` or `a(method="Count")` returns the number of times the function/block or R-code has been executed.

Value

`newCounter()` returns a counter function.

`a_counter_function()` returns the number of times it has been called (invisible).

`a_counter_function("Show")` returns the number of executions of the `a_counter_function`.

See Also

Other Performance Measurement: `Counted()`, `Timed()`, `newTimer()`

Examples

```
a<-newCounter()
a(); a()
a("Show")
```

newEnvXOR *Generates a problem environment for the XOR problem.*

Description

Generates a problem environment for the XOR problem.

Usage

```
newEnvXOR()
```

Value

The problem environment for the XOR problem with

- `$name`: `"envXOR"`, the name of the problem environment.
- `$buildtest(expr)`: The function which builds the environment for evaluating the expression by binding the variables to the parameters.
- `$TestCases`: The truthtable of the XOR function.
- `$f(expr, gene=NULL, lF=NULL)`: The fitness function. `expr` is the string with the logical expression to be evaluated.

The problem environment.

See Also

Other Problem Environments: `DeJongF4Factory()`, `DelayedPFactory()`, `Parabola2DEarlyFactory()`, `Parabola2DErrFactory()`, `Parabola2DFactory()`, `envXOR`, `lau15`, `newTSP()`

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)
```

NewlFevalGenes *Generate local functions and objects*

Description

`NewlFevalGenes` returns the list of functions containing a definition of all local objects required for the use of evaluation functions. We reference this object as local configuration. When adding additional evaluation functions, this must be extended by the constant (functions) needed to configure them.

Usage

```
NewlFevalGenes(penv)
```

Arguments

`penv`	A problem environment.

Value

The local configuration. A list of functions.

Examples

```
Parabola2D<-Parabola2DFactory()
lF<-NewlFevalGenes(Parabola2D)
lF$Max()
```

NewlFselectGenes *Generate local functions and objects.*

Description

`NewlFselectGenes` returns the list of functions which contains a definition of all local objects required for the use of selection functions. We reference this object as local configuration. When adding additional selection functions, this must be extended by the constant (functions) needed to configure them.

Usage

```
NewlFselectGenes()
```

Value

Local configuration `lF`.

Examples

```
lF<-NewlFselectGenes()
lF$Max()
```

`newTimer` *Timer for R code chunks.*

Description

`newTimer` sets up a timer object with two internal state variables, namely `count` to count the number of timer calls and `tUsed` to calculate the total time spent in a code block between two timer calls.

Usage

```
newTimer()
```

Details

- Generate a timer: `a<-newTimer()` sets up the timer `a`. The timer `a` supports three methods:
  1. `a()` or `a("Measure")` or `a(method="Measure")` **starts** the timer when called 1st, 3rd, 5th, ... time and **stops** the timer when called the 2nd, 4th, 6th, ... time. The calls can be manually inserted before and after a block of R-code for profiling.
  2. `a("TimeUsed")` or `a(method="TimeUsed")` returns the time used in seconds.
  3. `a("Count")` or `a(method="Count")` returns the number of times the function/block or R-code has been executed.
- The second way of usage is with the `Timed` function:
  1. Generate a timer: `a<-newTimer()` sets up the timer `a`.
  2. You convert a function `b` into a timed function `bTimed` by `bTimed<-Timed(a, b)`.
  3. You use `bTimed` instead of `b`.
  4. At the end, you can query the aggregated time and the aggregated number of executions by `a("TimeUsed")` and `a("Count")`, respectively.

Value

`newTimer()` returns a timer function.

`a_timer_function()` returns the used time in seconds (invisible).

`a_timer_function("TimeUsed")` returns the used time in seconds.

`a_timer_function("Count")` returns the number of executions of a timed function and/or a timed block of R-Code in seconds.

See Also

Other Performance Measurement: `Counted()`, `Timed()`, `newCounter()`

Examples

```
a<-newTimer()
a(); Sys.sleep(2); a()
a("TimeUsed")
a("Count")
```

`newTSP` *Generate a TSP problem environment*

Description

`newTSP` generates the problem environment for a traveling salesman problem (TSP).

Usage

```
newTSP(D, Name, Cities = NA, Solution = NA, Path = NA)
```

Arguments

`D`	A `n` x `n` distance matrix.
`Name`	The name of the problem environment.
`Cities`	The names of the cities.
`Solution`	Solution of problem (if known). Default: `NA`
`Path`	Optimal permutation of cities (if known). As integer vector. Default: `NA`.

Details

`newTSP` provides several local permutation improvement heuristics: a greedy path of length k starting from city i, the best greedy path of length k, a random 2-Opt-move, and a sequence of random 2-Opt moves. They help to find bounds for the TSP or to implement special purpose mutation operators.

Value

A problem environment for the TSP.

1. `$name()` a string with the name of the environment
2. `$cities()` a vector length `n` of city names.
3. `$dist()` the `n` x `n` distance matrix between `n` cities.
4. `$genelength()` the size of the permutation `n`. E.g. for a TSP: the number of cities.
5. `$f(permutation, gene, lF, tour=TRUE)` the fitness function of the TSP. If `tour==FALSE`, the path length is computed (without the cost from city `n` to city 1).
   With a permutation of size `n` as argument.
6. `$show(p)` shows tour through the cities in path `p` with its cost.

7. `$greedy(startPosition, k)` computes a `k`-step greedy minimal cost path beginning at the city `start`. For `k+1=n` the greedy solution gives an upper bound for the TSP.
8. `$kBestGreedy(k, tour=TRUE)` computes the best greedy subtour with `k+1` cities. For `tour=FALSE`, the best greedy subpath with `k+1` cities is computed.
9. `$rnd2Opt(permutation, maxTries=5)` returns a permutation improved by a single random 2-Opt-move after at most `maxTries=5` attempts.
10. `$LinKernighan(permutation, maxTries=5, show=FALSE)` returns the best permutation found after several random 2-Opt-moves with at most `maxTries=5` attempts. The loop stops after the first 2-Opt-move which does not improve the solution.
11. `$solution()` known optimal solution
12. `$path()` known optimal round trip

See Also

Other Problem Environments: `DeJongF4Factory()`, `DelayedPFactory()`, `Parabola2DEarlyFactory()`, `Parabola2DErrFactory()`, `Parabola2DFactory()`, `envXOR`, `lau15`, `newEnvXOR()`

Examples

```
a<-matrix(0, nrow=15, ncol=15)
a[1,]<- c(0, 29, 82, 46, 68, 52, 72, 42, 51,  55,  29,  74,  23,  72,  46)
a[2,]<- c(29,  0, 55, 46, 42, 43, 43, 23, 23,  31,  41,  51,  11,  52,  21)
a[3,]<- c(82, 55,  0, 68, 46, 55, 23, 43, 41,  29,  79,  21,  64,  31,  51)
a[4,]<-c(46, 46, 68,  0, 82, 15, 72, 31, 62,  42,  21,  51,  51,  43,  64)
a[5,]<-c(68, 42, 46, 82,  0, 74, 23, 52, 21,  46,  82,  58,  46,  65,  23)
a[6,]<-c(52, 43, 55, 15, 74,  0, 61, 23, 55,  31,  33,  37,  51,  29,  59)
a[7,]<-c(72, 43, 23, 72, 23, 61,  0, 42, 23,  31,  77,  37,  51,  46,  33)
a[8,]<-c(42, 23, 43, 31, 52, 23, 42,  0, 33,  15,  37,  33,  33,  31,  37)
a[9,]<-c(51, 23, 41, 62, 21, 55, 23, 33,  0,  29,  62,  46,  29,  51,  11)
a[10,]<-c(55, 31, 29, 42, 46, 31, 31, 15, 29,  0,  51,  21,  41,  23,  37)
a[11,]<-c(29, 41, 79, 21, 82, 33, 77, 37, 62,  51,   0,  65,  42,  59,  61)
a[12,]<-c(74, 51, 21, 51, 58, 37, 37, 33, 46,  21,  65,   0,  61,  11,  55)
a[13,]<-c(23, 11, 64, 51, 46, 51, 51, 33, 29,  41,  42,  61,   0,  62,  23)
a[14,]<-c(72, 52, 31, 43, 65, 29, 46, 31, 51,  23,  59,  11,  62,   0,  59)
a[15,]<-c(46, 21, 51, 64, 23, 59, 33, 37, 11,  37,  61,  55,  23,  59,   0)
lau15<-newTSP(a, Name="lau15")
lau15$name()
lau15$genelength()
b<-sample(1:15, 15, FALSE)
lau15$f(b)
lau15$f(b, tour=TRUE)
lau15$show(b)
lau15$greedy(1, 14)
lau15$greedy(1, 1)
```

`Parabola2DEarlyFactory` *Factory for a 2-dimensional quadratic parabola with early termination check.*

Description

This list of functions sets up the problem environment for a 2-dimensional quadratic parabola.

Usage

```
Parabola2DEarlyFactory()
```

Details

The factory contains examples of all functions which form the interface of a problem environment to the simple genetic algorithm with binary-coded genes of package `xega`. This factory provides examples of a termination condition, a description function, and a solution function:

- `terminate(solution)` checks for an early termination condition.
- `describe()` shows a description of the function.
- `solution()` returns a list with the `minimum` and the `maximum` values as well as the lists `minpoints` and `maxpoints` of the minimal and the maximal points.

Value

A problem environment represented as a list of functions:

- `$name()`: The name of the problem environment.
- `$bitlength()`: The vector of the number of bits of each parameter of the function.
- `$genelength()`: The number of bits of the gene.
- `$lb()`: The vector of lower bounds of the parameters.
- `$ub()`: The vector of upper bounds of the parameters.
- `$f(parm, gene=0, lF=0))`: The fitness function.

Additional elements:

- `$describe()`: Print a description of the problem environment to the console.
- `$solution()`: The solution structure. A named list with `minimum`, `maximum` and 2 lists of equivalent solutions: `minpoints`, `maxpoints`.

See Also

DelayedP, Parabola2DErr

Other Problem Environments: `DeJongF4Factory()`, `DelayedPFactory()`, `Parabola2DErrFactory()`, `Parabola2DFactory()`, `envXOR`, `lau15`, `newEnvXOR()`, `newTSP()`

Examples

```
Parabola2D<-Parabola2DEarlyFactory()
Parabola2D$f(c(2.2, 1.0))
```

`Parabola2DErrFactory` *Factory for a randomly failing 2-dimensional quadratic parabola.*

Description

This list of functions sets up the problem environment for a 2-dimensional quadratic parabola which produces an error with a probability of 0.5.

Usage

```
Parabola2DErrFactory()
```

Details

The factory contains examples of all functions which form the interface of a problem environment to the simple genetic algorithm with binary-coded genes of package `xega`.

Value

A problem environment represented as a list of functions:

- `$name()`: The name of the problem environment.
- `$bitlength()`: The vector of the number of bits of each parameter of the function.
- `$genelength()`: The number of bits of the gene.
- `$lb()`: The vector of lower bounds of the parameters.
- `$ub()`: The vector of upper bounds of the parameters.
- `$f(parm, gene=0, lF=0))`: The fitness function.

Additional elements:

- `$describe()`: Print a description of the problem environment to the console.
- `$solution()`: The solution structure. A named list with `minimum`, `maximum` and 2 lists of equivalent solutions: `minpoints`, `maxpoints`.

See Also

DelayedP

Other Problem Environments: `DeJongF4Factory()`, `DelayedPFactory()`, `Parabola2DEarlyFactory()`, `Parabola2DFactory()`, `envXOR`, `lau15`, `newEnvXOR()`, `newTSP()`

Examples

```
Parabola2DErr<-Parabola2DErrFactory()
```

`Parabola2DFactory` *Factory for a 2-dimensional quadratic parabola.*

Description

This list of functions sets up the problem environment for a 2-dimensional quadratic parabola.

Usage

```
Parabola2DFactory()
```

Details

The factory contains examples of all functions which form the interface of a problem environment to the simple genetic algorithm with binary-coded genes of package `xega`.

Value

A problem environment represented as a list of functions:

- `$name()`: The name of the problem environment.
- `$bitlength()`: The vector of the number of bits of each parameter of the function.
- `$genelength()`: The number of bits of the gene.
- `$lb()`: The vector of lower bounds of the parameters.
- `$ub()`: The vector of upper bounds of the parameters.
- `$f(parm, gene=0, lF=0))`: The fitness function.

Additional elements:

- `$describe()`: Print a description of the problem environment to the console.
- `$solution()`: The solution structure. A named list with `minimum`, `maximum` and 2 lists of equivalent solutions: `minpoints`, `maxpoints`.

See Also

DelayedP, Parabola2DErr

Other Problem Environments: `DeJongF4Factory()`, `DelayedPFactory()`, `Parabola2DEarlyFactory()`, `Parabola2DErrFactory()`, `envXOR`, `lau15`, `newEnvXOR()`, `newTSP()`

Examples

```
Parabola2D<-Parabola2DFactory()
Parabola2D$f(c(2.2, 1.0))
```

`parm` *Factory for constants*

Description

`parm` builds a constant function for `x`. See Wickham (2019).

Usage

```
parm(x)
```

Arguments

`x`	A constant.

Value

The constant function.

References

Wickham, Hadley (2019): Advanced R, CRC Press, Boca Raton.

Examples

```
TournamentSize<-parm(2)
TournamentSize()
Eps<-parm(0.01)
Eps()
```

`predictSelectTime` *Predict the time use of a selection method for a popsize.*

Description

Predict the time use of a selection method for a popsize.

Usage

```
predictSelectTime(df, method = "Uniform", popsize = 1e+05)
```

Arguments

`df`	Data frame.
`method`	Selection method.
`popsize`	Population size.

Value

List with

- `$model`: The result of `stats::lm`.
- `$predict`: The result of `stats::predict`.

Warning

Uses a quadratic regression model. But the complexities of the functions are of orders O(1), O(n), O(n.ln(n)) and O(n^2).

See Also

Other Benchmark Selection Functions: `runOneBenchmark()`, `runSelectBenchmarks()`, `selectBenchmark()`, `testSelectGene()`

Examples

```
popsizes<-as.integer(seq(from=100, to=200, length.out=5))
a<-runSelectBenchmarks(popsizes, both=TRUE)
b<-predictSelectTime(a, method="SUS", 155)
summary(b$model)
b$predicted
c<- predictSelectTime(a, method="SUS  C", c(155, 500))
summary(c$model)
c$predicted
```

`runOneBenchmark` *Script for testing a single selection functions*

Description

Script for testing a single selection functions

Usage

```
runOneBenchmark(name, limit = c(10, 100, 1000), both = TRUE, verbose = FALSE)
```

Arguments

`name`	Name is one of the following strings. 1. "Uniform" benchmarks `SelectUniform`. 2. "ProportionalOnln" benchmarks `SelectPropFitOnln`. 3. "Proportional" benchmarks `SelectPropFit`. 4. "ProportionalM" benchmarks `SelectPropFitM`. 5. "PropFitDiffOnln" benchmarks `SelectPropFitDiffOnln`. 6. "PropFitDiff" benchmarks `SelectPropFitDiff`.

7. "PropFitDiffM" benchmarks `SelectPropFitDiffM`.
8. "Tournament" benchmarks `SelectTournament`.
9. "Duel" benchmarks `SelectDuel`.
10. "LinearRank" benchmarks `SelectLinearRank`.
11. "SUS" benchmarks `SelectSUS`.

`limit`	Vector of population sizes.
`both`	For `both=TRUE` the selection function is benchmarked with and without transformation. For `both=FALSE`, only the transformed selection functions are benchmarked.
`verbose`	Boolean. Default: `FALSE`. If `TRUE`, the function benchmarked and the population size are printed to the console.

Value

A data frame sorted in ascending order of time of last column.

Warning

The time to run the function for `lim>6` explodes for all benchmark functions with higher than linear complexity. (e.g. `PropFit`, `PropFitdiff`, and `Tournament`).

See Also

Other Benchmark Selection Functions: `predictSelectTime()`, `runSelectBenchmarks()`, `selectBenchmark()`, `testSelectGene()`

Examples

```
runOneBenchmark("Duel", 5, both=FALSE)
runOneBenchmark("PropFitDiffOnln")
```

`runSelectBenchmarks` *Script for testing all selection functions*

Description

Script for testing all selection functions

Usage

```
runSelectBenchmarks(lim = c(10, 100), both = TRUE, verbose = FALSE)
```

Arguments

`lim`	Vector of population sizes.
`both`	For `both=TRUE` the selection function is benchmarked with and without transformation. For `both=FALSE`, only the transformed selection functions are benchmarked.
`verbose`	Boolean. Default: `FALSE`. If `TRUE`, the function benchmarked and the population size are printed to the console.

Value

A data frame sorted in ascending order of the time of the last column. The fastest selection methods come first. The first row contains the population sizes with which the benchmark has been performed. The data frame has `1+length(lim)` columns:

- "Benchmark": The name of the benchmarked selection function. A "C" after the name indicates that the selection function has been transformed into a lookup function.
- `length(lim)` columns with the execution times in seconds.

See Also

Other Benchmark Selection Functions: `predictSelectTime()`, `runOneBenchmark()`, `selectBenchmark()`, `testSelectGene()`

Examples

```
runSelectBenchmarks(lim=c(10, 100), both=TRUE, verbose=TRUE)
runSelectBenchmarks(lim=c(10, 100), both=FALSE)
```

`ScaleFitness` *Scaling Fitness*

Description

Fitness is transformed by a power function `fit^k`. If `k` is

- less than 1: Selection pressure is decreased.
- 1: Selection pressure remains constant.
- larger than 1: Selection pressure is increased.
- 0: Fitness is constant. Random selection.
- smaller than 0: Fitness is `1/k`.

Usage

```
ScaleFitness(fit, k, lF)
```

Arguments

`fit`	A fitness vector.
`k`	Scaling exponent.
`lF`	Local configuration.

Details

Power functions are used for contrast sharpening or softening in image analysis. For fuzzy sets representing the value of a linguistic variable, the power function has been used as concentration or dilation transformations for modeling adverbs.

Value

A scaled fitness vector.

References

Wenstop, Fred (1980) Quantitative Analysis with Linguistic Variables. Fuzzy Sets and Systems, 4(2), pp. 99-115. <doi:10.1016/0165-0114(80)90031-7>

See Also

Other Scaling: `ContinuousScaleFitness()`, `DispersionRatio()`, `ScalingFitness()`, `ThresholdScaleFitness()`

Examples

```
lF<-list()
lF$Offset<-parm(0.0001)
fit<-sample(10, 20, replace=TRUE)
fit
ScaleFitness(fit, 0.5, lF)
```

`ScalingFactory` *Scaling Factory*

Description

Scaling Factory

Usage

```
ScalingFactory(method = "NoScaling")
```

Arguments

`method`	A scaling method. Available methods are: • "NoScaling": Identity (Default). • "ConstantScaling": `fit^k` with constant exponent. Function `ConstantScaling`. • "ThresholdScaling": If the dispersion ratio is larger than `1+threshold`, use a constant scaling exponent with a value below 1 (decrease of selection pressure). Function `ThresholdScaling`. • If the dispersion ratio is lower than `1-threshold`, use a constant scaling exponent with a value above 1 (increase of selection pressure). • "ContinuousScaling": Use weighted dispersion ratio as scaling exponent. Function `ContinuousScaling`.

Value

A scaling function.

See Also

Other Configuration: `DispersionMeasureFactory()`, `EvalGeneFactory()`, `SelectGeneFactory()`

Examples

```
fit<-sample(10, 20, replace=TRUE)
lF<-list()
lF$ScalingExp<-parm(2)
Scale<-ScalingFactory()
fit
Scale(fit, lF)
Scale<-ScalingFactory("ConstantScaling")
Scale(fit, lF)
```

`ScalingFitness` *Abstract interface for ScaleFitness.*

Description

The scaling constant `k` is set by the function `lF$ScalingExp`.

Usage

```
ScalingFitness(fit, lF)
```

Arguments

`fit`	A fitness vector.
`lF`	Local configuration.

Value

Scaled fitness vector

See Also

Other Scaling: `ContinuousScaleFitness()`, `DispersionRatio()`, `ScaleFitness()`, `ThresholdScaleFitness()`

Examples

```
lF<-list()
lF$Offset<-parm(0.0001)
lF$ScalingExp<-parm(2)
fit<-sample(10, 20, replace=TRUE)
fit
ScalingFitness(fit, lF)
```

`selectBenchmark` *Benchmark and stress test of selection functions.*

Description

Times a selection function for populations of size 10 to 10^{limit}.

Usage

```
selectBenchmark(
  method = "Uniform",
  continuation = TRUE,
  limit = c(10, 100, 1000),
  verbose = FALSE
)
```

Arguments

`method`	Selection function. Default: Uniform.
`continuation`	Convert to index function? Default: TRUE.
`limit`	Vector of population sizes.
`verbose`	Boolean. Default: `FALSE`. If `TRUE`, the function benchmarked and the population size are printed to the console.

Value

Vector of execution times in seconds.

See Also

Other Benchmark Selection Functions: `predictSelectTime()`, `runOneBenchmark()`, `runSelectBenchmarks()`, `testSelectGene()`

Examples

```
selectBenchmark(method="Uniform", continuation=TRUE, limit=c(10, 100, 1000))
selectBenchmark(method="SUS", continuation=TRUE, limit=c(5000, 10000, 15000))
selectBenchmark(method="SUS", continuation=FALSE, limit=seq(from=100, to=1000, length.out=5))
```

`SelectDuel` *Deterministic duel.*

Description

`SelectDuel` implements selection by a tournament between 2 randomly selected genes. The best gene always wins. This is the version of tournament selection with the least selection pressure.

Usage

```
SelectDuel(fit, lF, size = 1)
```

Arguments

`fit`	Fitness vector.
`lF`	Local configuration.
`size`	Number of selected genes. Default: 1.

Details

This is an O(n) implementation of tournament selection with a tournament size of 2.

A special case of tournament selection.

Value

The index vector of the selected genes.

See Also

Other Selection Functions: `SelectLRSelective()`, `SelectLinearRankTSR()`, `SelectPropFitDiffM()`, `SelectPropFitDiffOnln()`, `SelectPropFitDiff()`, `SelectPropFitM()`, `SelectPropFitOnln()`, `SelectPropFit()`, `SelectSTournament()`, `SelectSUS()`, `SelectTournament()`, `SelectUniformP()`, `SelectUniform()`

Examples

```
fit<-sample(10, 15, replace=TRUE)
SelectDuel(fit, NewlFselectGenes())
SelectDuel(fit, NewlFselectGenes(), length(fit))
```

`SelectGeneFactory` *Configure the selection function of a genetic algorithm.*

Description

`SelectGeneFactory` implements selection of one of the gene selection 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. "Uniform" returns `SelectUniform`.
2. "UniformP" returns `SelectUniformP`.
3. "ProportionalOnln" returns `SelectPropFitOnln`.
4. "Proportional" returns `SelectPropFit`.
5. "ProportionalM" returns `SelectPropFitM`.
6. "PropFitDiffOnln" returns `SelectPropFitDiffOnln`.
7. "PropFitDiff" returns `SelectPropFitDiff`.
8. "PropFitDiffM" returns `SelectPropFitDiffM`.
9. "Tournament" returns `SelectTournament`.
10. "STournament" returns `SelectSTournament`.
11. "Duel" returns `SelectDuel`.
12. "LRSelective" returns `SelectLRSelective`.
13. "LRTSR" returns `SelectLinearRankTSR`.
14. "SUS" returns a function factory for `SelectSUS`.

Usage

```
SelectGeneFactory(method = "PropFitDiffOnln")
```

Arguments

`method`	A string specifying the selection function.

Details

If `SelectionContinuation()==TRUE` then:

1. In package xegaPopulation function `NextPopulation`, first the functions `SelectGene` and `SelectMate` are transformed by `TransformSelect` to a continuation function with embedded index vector and counter.
2. For each call in `ReplicateGene`, `SelectGene` and `SelectMate` return the index of the selected gene.

Value

A selection function for genes.

See Also

Other Configuration: `DispersionMeasureFactory()`, `EvalGeneFactory()`, `ScalingFactory()`

Examples

```
SelectGene<-SelectGeneFactory("Uniform")
fit<-sample(10, 15, replace=TRUE)
SelectGene(fit, lFselectGenes)
sel<-"Proportional"
SelectGene<-SelectGeneFactory(method=sel)
fit<-sample(10, 15, replace=TRUE)
SelectGene(fit, lFselectGenes)
```

`SelectLinearRankTSR` *Linear rank selection.*

Description

`SelectLinearRankTSR` implements selection with interpolated target sampling rates.

Usage

```
SelectLinearRankTSR(fit, lF, size = 1)
```

Arguments

`fit`	Fitness vector.
`lF`	Local configuration.
`size`	Size of return vector (default: 1).

Details

The target sampling rate is a linear interpolation between `lF$MaxTSR` and `Min<-2-lF$MaxTSR`, because the sum of the target sampling rates is n. The target sampling rates are computed and used as a fitness vector for stochastic universal sampling algorithm implemented by `SelectSUS`. `lF$MaxTSR` should be in [1.0, 2.0].

TODO: More efficient implementation. We use two sorts!

Value

The index vector of selected genes.

References

Grefenstette, John J. and Baker, James E. (1989): How Genetic Algorithms Work: A Critical Look at Implicit Parallelism In Schaffer, J. David (Ed.) *Proceedings of the Third International Conference on Genetic Algorithms on Genetic Algorithms*, pp. 20-27. (ISBN:1-55860-066-3)

See Also

Other Selection Functions: `SelectDuel()`, `SelectLRSelective()`, `SelectPropFitDiffM()`, `SelectPropFitDiffOnln()`, `SelectPropFitDiff()`, `SelectPropFitM()`, `SelectPropFitOnln()`, `SelectPropFit()`, `SelectSTournament()`, `SelectSUS()`, `SelectTournament()`, `SelectUniformP()`, `SelectUniform()`

Examples

```
fit<-sample(10, 15, replace=TRUE)
SelectLinearRankTSR(fit, NewlFselectGenes())
SelectLinearRankTSR(fit, NewlFselectGenes(), length(fit))
```

`SelectLRSelective` *Linear rank selection with selective pressure.*

Description

`SelectLRSelective` implements selection by Whitley's linear rank selection with selective pressure for the GENITOR algorithm. See Whitley, Darrell (1989), p. 121.

Usage

```
SelectLRSelective(fit, lF, size = 1)
```

Arguments

`fit`	Fitness vector.
`lF`	Local configuration.
`size`	Size of return vector (default: 1).

Details

The selection pressure is configured by the constant function `lF$SelectionBias()`. Its values should be strictly larger than 1 and preferably below 2. The default is set to 1.5. A value of 1.0 means uniform random selection.

Value

The index vector of selected genes.

References

Whitley, Darrell (1989): The GENITOR Algorithm and Selection Pressure. Why Rank-Based Allocation of Reproductive Trials is Best. In Schaffer, J. David (Ed.) *Proceedings of the Third International Conference on Genetic Algorithms on Genetic Algorithms*, pp. 116-121. (ISBN:1-55860-066-3)

See Also

Other Selection Functions: `SelectDuel()`, `SelectLinearRankTSR()`, `SelectPropFitDiffM()`, `SelectPropFitDiffOnln()`, `SelectPropFitDiff()`, `SelectPropFitM()`, `SelectPropFitOnln()`, `SelectPropFit()`, `SelectSTournament()`, `SelectSUS()`, `SelectTournament()`, `SelectUniformP()`, `SelectUniform()`

Examples

```
fit<-sample(10, 15, replace=TRUE)
SelectLRSelective(fit, NewlFselectGenes())
SelectLRSelective(fit, NewlFselectGenes(), length(fit))
```

`SelectPropFit` *Selection proportional to fitness* $O(n\hat{}2)$.

Description

`SelectPropFit` implements selection proportional to fitness. Negative fitness vectors are shifted to R^+. The default of the function `lf$Offset` is 1. Holland's schema theorem uses this selection function. See John Holland (1975) for further information.

Usage

```
SelectPropFit(fit, lF, size = 1)
```

Arguments

`fit`	Fitness vector.
`lF`	Local configuration.
`size`	Number of selected genes. Default: 1.

Value

The index vector of the selected genes.

Warning

There is a potential slow for-loop in the code.

References

Holland, John (1975): *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor. (ISBN:0-472-08460-7)

See Also

Other Selection Functions: `SelectDuel()`, `SelectLRSelective()`, `SelectLinearRankTSR()`, `SelectPropFitDiffM()`, `SelectPropFitDiffOnln()`, `SelectPropFitDiff()`, `SelectPropFitM()`, `SelectPropFitOnln()`, `SelectSTournament()`, `SelectSUS()`, `SelectTournament()`, `SelectUniformP()`, `SelectUniform()`

Examples

```
fit<-sample(10, 15, replace=TRUE)
SelectPropFit(fit, NewlFselectGenes())
SelectPropFit(fit, NewlFselectGenes(), length(fit))
```

`SelectPropFitDiff` *Selection proportional to fitness differences.*

Description

`SelectPropFitDiff` implements selection proportional to fitness differences. It selects a gene out of the population with a probability proportional to the fitness difference to the gene with minimal fitness. The fitness of survival of the gene with minimal fitness is set by `lF$Eps()` to `0.01` per default. See equation (7.45) Andreas Geyer-Schulz (1997), p. 205.

Usage

```
SelectPropFitDiff(fit, lF, size = 1)
```

Arguments

`fit`	Fitness vector.
`lF`	Local configuration.
`size`	Number of selected genes. Default: 1.

Value

The index vector of the selected genes.

Note

`SelectPopFitDiff` is a dynamic scaling function. Complexity: $O(n^2)$.

References

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

See Also

Other Selection Functions: SelectDuel(), SelectLRSelective(), SelectLinearRankTSR(), SelectPropFitDiffM(), SelectPropFitDiffOnln(), SelectPropFitM(), SelectPropFitOnln(), SelectPropFit(), SelectSTournament(), SelectSUS(), SelectTournament(), SelectUniformP(), SelectUniform()

Examples

```
fit<-sample(10, 15, replace=TRUE)
SelectPropFitDiff(fit, NewlFselectGenes())
SelectPropFitDiff(fit, NewlFselectGenes(), length(fit))
```

SelectPropFitDiffM *Selection proportional to fitness differences.*

Description

`SelectPropFitDiffM` implements selection proportional to fitness differences. It selects a gene from the population with a probability proportional to the fitness difference to the gene with minimal fitness. The fitness of survival of the gene with minimal fitness is set by `lF$Eps()` to `0.01` per default. See equation (7.45) Andreas Geyer-Schulz (1997), p. 205.

Usage

```
SelectPropFitDiffM(fit, lF, size = 1)
```

Arguments

fit	Fitness vector.
lF	Local configuration.
size	Number of selected genes. Default: 1.

Value

The index vector of the selected genes.

Warning

`outer` uses $O(n^2)$ memory cells.

Note

`SelectPopFitDiff` is a dynamic scaling function.

References

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

See Also

Other Selection Functions: SelectDuel(), SelectLRSelective(), SelectLinearRankTSR(), SelectPropFitDiffOnln(), SelectPropFitDiff(), SelectPropFitM(), SelectPropFitOnln(), SelectPropFit(), SelectSTournament(), SelectSUS(), SelectTournament(), SelectUniformP(), SelectUniform()

Examples

```
fit<-sample(10, 15, replace=TRUE)
SelectPropFitDiffM(fit, NewlFselectGenes())
SelectPropFitDiffM(fit, NewlFselectGenes(), length(fit))
```

`SelectPropFitDiffOnln` *Selection proportional to fitness differences O(n ln(n)).*

Description

`SelectPropFitDiffOnln` implements selection proportional to fitness differences. Negative fitness vectors are shifted to R^+. The default of the function `lf$Offset` is `1`. Holland's schema theorem uses this selection function. See John Holland (1975) for further information.

Usage

```
SelectPropFitDiffOnln(fit, lF, size = 1)
```

Arguments

`fit`	Fitness vector.
`lF`	Local configuration.
`size`	Number of selected genes. Default: 1.

Details

This is a fast implementation which gives exactly the same results as the functions `SelectPropFitDiff` and `SelectPropDiffFitM`. Its runtime is $O(n.ln(n))$.

Value

The index vector of the selected genes.

Credits

The code of this function has been adapted by Fabian Aisenbrey.

Warning

There is a potential slow for-loop in the code.

References

Holland, John (1975): *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor. (ISBN:0-472-08460-7)

See Also

Other Selection Functions: `SelectDuel()`, `SelectLRSelective()`, `SelectLinearRankTSR()`, `SelectPropFitDiffM()`, `SelectPropFitDiff()`, `SelectPropFitM()`, `SelectPropFitOnln()`, `SelectPropFit()`, `SelectSTournament()`, `SelectSUS()`, `SelectTournament()`, `SelectUniformP()`, `SelectUniform()`

Examples

```
fit<-sample(10, 15, replace=TRUE)
SelectPropFitDiffOnln(fit, NewlFselectGenes())
SelectPropFitOnln(fit, NewlFselectGenes(), length(fit))
```

`SelectPropFitM` *Selection proportional to fitness (vector/matrix).*

Description

`SelectPropFitM` implements selection proportional to fitness. Negative fitness vectors are shifted to R^+. The default of the function `lf$Offset` is 1. Holland's schema theorem uses this selection function. See John Holland (1975) for further information.

Usage

```
SelectPropFitM(fit, lF, size = 1)
```

Arguments

`fit`	Fitness vector.
`lF`	Local configuration.
`size`	Number of selected genes. Default: 1.

Value

The index vector of the selected genes.

Warning

The code is completely written in vector/matrix operations. `outer` uses $O(n^2)$ memory cells.

References

Holland, John (1975): *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor. (ISBN:0-472-08460-7)

See Also

Other Selection Functions: `SelectDuel()`, `SelectLRSelective()`, `SelectLinearRankTSR()`, `SelectPropFitDiffM()`, `SelectPropFitDiffOnln()`, `SelectPropFitDiff()`, `SelectPropFitOnln()`, `SelectPropFit()`, `SelectSTournament()`, `SelectSUS()`, `SelectTournament()`, `SelectUniformP()`, `SelectUniform()`

Examples

```
fit<-sample(10, 15, replace=TRUE)
SelectPropFitM(fit, NewlFselectGenes())
SelectPropFitM(fit, NewlFselectGenes(), length(fit))
```

`SelectPropFitOnln` *Selection proportional to fitness O(n ln(n)).*

Description

`SelectPropFitOnln` implements selection proportional to fitness. Negative fitness vectors are shifted to R^+. The default of the function `lf$Offset` is 1. Holland's schema theorem uses this selection function. See John Holland (1975) for further information.

@details This is a fast implementation with equivalent results to the functions `SelectPropFit` and `SelectPropFitM`. Its runtime is $O(n.ln(n))$.

Usage

```
SelectPropFitOnln(fit, lF, size = 1)
```

Arguments

`fit`	Fitness vector.
`lF`	Local configuration.
`size`	Number of selected genes. Default: 1.

Value

The index vector of the selected genes.

Credits

The code of this function has been written by Fabian Aisenbrey.

References

Holland, John (1975): *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor. (ISBN:0-472-08460-7)

See Also

Other Selection Functions: `SelectDuel()`, `SelectLRSelective()`, `SelectLinearRankTSR()`, `SelectPropFitDiffM()`, `SelectPropFitDiffOnln()`, `SelectPropFitDiff()`, `SelectPropFitM()`, `SelectPropFit()`, `SelectSTournament()`, `SelectSUS()`, `SelectTournament()`, `SelectUniformP()`, `SelectUniform()`

Examples

```
fit<-sample(10, 15, replace=TRUE)
SelectPropFitOnln(fit, NewlFselectGenes())
SelectPropFitOnln(fit, NewlFselectGenes(), length(fit))
```

`SelectSTournament` *Stochastic tournament selection.*

Description

`SelectSTournament` implements selection through a stochastic tournament between `lF$TournamentSize()` randomly selected genes. A gene wins a tournament with a probability proportional to its fitness. The default of `lF$TournamentSize()` is 2. A tournament with 2 participants has the least selection pressure.

`lF$TournamentSize` must be less than the population size.

Usage

```
SelectSTournament(fit, lF, size = 1)
```

Arguments

`fit`	Fitness vector.
`lF`	Local configuration.
`size`	Number of selected genes. Default: 1.

Value

The index vector of the selected genes.

See Also

Other Selection Functions: `SelectDuel()`, `SelectLRSelective()`, `SelectLinearRankTSR()`, `SelectPropFitDiffM()`, `SelectPropFitDiffOnln()`, `SelectPropFitDiff()`, `SelectPropFitM()`, `SelectPropFitOnln()`, `SelectPropFit()`, `SelectSUS()`, `SelectTournament()`, `SelectUniformP()`, `SelectUniform()`

Examples

```
fit<-sample(10, 15, replace=TRUE)
SelectSTournament(fit, NewlFselectGenes())
SelectSTournament(fit, NewlFselectGenes(), length(fit))
```

SelectSUS *Stochastic universal sampling.*

Description

SelectSUS implements selection by Baker's stochastic universal sampling method. SUS is a strictly sequential algorithm which has zero bias and minimal spread. SUS uses a single random number for each generation. See Baker, James E. (1987), p. 16.

Usage

```
SelectSUS(fit, lF, size = 1)
```

Arguments

fit	Fitness vector.
lF	Local configuration.
size	Number of selected genes. Default: 1.

Value

The index vector of the selected genes.

References

Baker, James E. (1987): Reducing Bias and Inefficiency in the Selection Algorithm. In Grefenstette, John J.(Ed.) *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms*, pp. 14-21. (ISBN:978-08058-0158-8)

See Also

Other Selection Functions: SelectDuel(), SelectLRSelective(), SelectLinearRankTSR(), SelectPropFitDiffM(), SelectPropFitDiffOnln(), SelectPropFitDiff(), SelectPropFitM(), SelectPropFitOnln(), SelectPropFit(), SelectSTournament(), SelectTournament(), SelectUniformP(), SelectUniform()

Examples

```
fit<-sample(10, 15, replace=TRUE)
SelectSUS(fit, NewlFselectGenes())
SelectSUS(fit, NewlFselectGenes(), length(fit))
```

SelectTournament *Tournament selection.*

Description

SelectTournament implements selection by doing a tournament between lF$TournamentSize() randomly selected genes. The best gene always wins. The default of lF$TournamentSize() is 2. This is the version with the least selection pressure.

lF$TournamentSize must be less than the population size.

Usage

```
SelectTournament(fit, lF, size = 1)
```

Arguments

fit	Fitness vector.
lF	Local configuration.
size	Number of selected genes. Default: 1.

Value

The index vector of the selected genes.

See Also

Other Selection Functions: SelectDuel(), SelectLRSelective(), SelectLinearRankTSR(), SelectPropFitDiffM(), SelectPropFitDiffOnln(), SelectPropFitDiff(), SelectPropFitM(), SelectPropFitOnln(), SelectPropFit(), SelectSTournament(), SelectSUS(), SelectUniformP(), SelectUniform()

Examples

```
fit<-sample(10, 15, replace=TRUE)
SelectTournament(fit, NewlFselectGenes())
SelectTournament(fit, NewlFselectGenes(), length(fit))
```

SelectUniform *Selection with uniform probability.*

Description

`SelectUniform` implements selection by choosing a gene with equal probability.

Usage

```
SelectUniform(fit, lF, size = 1)
```

Arguments

`fit`	Fitness vector.
`lF`	Local configuration.
`size`	Number of selected genes. Default: 1.

Details

This selection function is useful:

1. To specify mating behavior in crossover operators.
2. For computer experiments without selection pressure.
3. For computing random search solutions as a benchmark.

Value

The index vector of the selected genes.

See Also

Other Selection Functions: `SelectDuel()`, `SelectLRSelective()`, `SelectLinearRankTSR()`, `SelectPropFitDiffM()`, `SelectPropFitDiffOnln()`, `SelectPropFitDiff()`, `SelectPropFitM()`, `SelectPropFitOnln()`, `SelectPropFit()`, `SelectSTournament()`, `SelectSUS()`, `SelectTournament()`, `SelectUniformP()`

Examples

```
fit<-sample(10, 15, replace=TRUE)
SelectUniform(fit, NewlFselectGenes())
SelectUniform(fit, NewlFselectGenes(), length(fit))
```

SelectUniformP — *Selection with uniform probability without replacement.*

Description

`SelectUniformP` implements selection by choosing a gene with equal probability without replacement. Usage:

1. To specify mating behavior in crossover operators.
2. For computer experiments without selection pressure.

Usage

```
SelectUniformP(fit, lF, size = 1)
```

Arguments

`fit`	Fitness vector.
`lF`	Local configuration.
`size`	Number of selected genes. Default: 1.

Details

Selection without replacement guarantees that vectors of different indices are selected. A vector of the size of the population is a permutation of indices. This property is needed for the classic variant of differential evolution.

Value

The index vector of the selected genes.

References

Price, Kenneth V., Storn, Rainer M. and Lampinen, Jouni A. (2005) The Differential Evolution Algorithm (Chapter 2), pp. 37-134. In: Differential Evolution. A Practical Approach to Global Optimization. Springer, Berlin. <doi:10.1007/3-540-31306-0>

See Also

Other Selection Functions: `SelectDuel()`, `SelectLRSelective()`, `SelectLinearRankTSR()`, `SelectPropFitDiffM()`, `SelectPropFitDiffOnln()`, `SelectPropFitDiff()`, `SelectPropFitM()`, `SelectPropFitOnln()`, `SelectPropFit()`, `SelectSTournament()`, `SelectSUS()`, `SelectTournament()`, `SelectUniform()`

Examples

```
fit<-sample(10, 15, replace=TRUE)
SelectUniformP(fit, NewlFselectGenes())
SelectUniformP(fit, NewlFselectGenes(), length(fit))
```

`STournament` *Stochastic tournament of size* `k`.

Description

`STournament` is implemented in two steps:

1. A subset of size k of the population is selected with uniform probability.
2. A gene is selected with probability proportional to fitness.

Usage

```
STournament(fit, lF)
```

Arguments

`fit`	Fitness vector.
`lF`	Local configuration.

Value

Index of candidate.

Examples

```
fit<-sample(10, 15, replace=TRUE)
STournament(fit, NewlFselectGenes())
```

`testEvalGeneStoch` *Test of incremental mean, variance, and standard deviation.*

Description

Test of incremental mean, variance, and standard deviation.

Usage

```
testEvalGeneStoch(gene, lF, rep)
```

Arguments

`gene`	A gene.
`lF`	A local function list with a problem environment.
`rep`	Number of repeated evaluations.

Value

A gene.

Examples

```
DeJongF4<-DeJongF4Factory()
lF<-NewlFevalGenes(DeJongF4)
g1<-list(evaluated=FALSE, evalFail=FALSE, fit=0, gene1=c(1.0, -1.5))
g10<-testEvalGeneStoch(g1, lF, rep=10)
g10
```

`testSelectGene` *Test a gene selection function*

Description

`testSelectGene` implements testing a selection function. It collects the results of the repeated execution of the selection function given a fitness function.

Usage

```
testSelectGene(
  fit,
  method = "Uniform",
  howOften = 100,
  lF = NewlFselectGenes(),
  continuation = TRUE,
  verbose = FALSE
)
```

Arguments

`fit`	Fitness vector.
`method`	String specifying the selection function. See `SelectGeneFactory`.
`howOften`	Integer.
`lF`	Local configuration.
`continuation`	Convert to index function?
`verbose`	Boolean. Default: `FALSE`. If `TRUE`, the exection time of the transformation of the selection function into a quasi-continuation function is printed on the console.

Value

- `$fit` fitness vector.
- `$newPop` indices of survivors in fitness vector.
- `$time` time in seconds.
- `$size` population size.
- `$method` selection method used.

See Also

Other Benchmark Selection Functions: `predictSelectTime()`, `runOneBenchmark()`, `runSelectBenchmarks()`, `selectBenchmark()`

Examples

```
fit1<-rep(10,10)
fit2<-fit1+runif(rep(10,1))
fit3<-sample(100, 10, replace=TRUE)
testSelectGene(fit2, method="Tournament", howOften=100)
testSelectGene(fit3, method="Tournament", howOften=10)
```

`ThresholdScaleFitness` *Dispersion Ratio Based Fitness Scaling.*

Description

Fitness is transformed by a power function `fit^lF$ScalingExp`. If `lF$ScalingExp` is

- less than 1: Selection pressure is decreased.
- 1: Selection pressure remains constant.
- larger than 1: Selection pressure is increased.
- 0: Fitness is constant. Random selection.
- smaller than 1: Fitness is `1/fit^lF$ScalingExp`.

Usage

```
ThresholdScaleFitness(fit, lF)
```

Arguments

`fit`	Fitness vector.
`lF`	Local configuration.

Value

Scaled fitness vector.

See Also

Other Scaling: `ContinuousScaleFitness()`, `DispersionRatio()`, `ScaleFitness()`, `ScalingFitness()`

Other Adaptive Parameter: `ContinuousScaleFitness()`

Examples

```
lF<-list()
lF$Offset<-parm(0.0001)
lF$ScalingThreshold<-parm(0.05)
lF$RDM<-parm(1.0)
lF$ScalingExp<-parm(0.5)
lF$ScalingExp2<-parm(2)
fit<-sample(10, 20, replace=TRUE)
fit
ThresholdScaleFitness(fit, lF)
lF$RDM<-parm(1.2)
ThresholdScaleFitness(fit, lF)
lF$RDM<-parm(0.8)
ThresholdScaleFitness(fit, lF)
```

`Timed` *Transformation into a timed function*

Description

`Timed` takes two functions as arguments, namely the function whose time and call frequency should be measured and a timer object created by `newTimer()`. It returns a timed function.

Usage

```
Timed(FUN, timer)
```

Arguments

`FUN`	A function whose run time should be measured.
`timer`	A timer generated by `newTimer()`.

Value

A timed function.

See Also

Other Performance Measurement: `Counted()`, `newCounter()`, `newTimer()`

Examples

```
    test<-function(seconds) {Sys.sleep(seconds)}
    testTimer<-newTimer()
    testTimed<-Timed(test, testTimer)
    testTimer("Count"); testTimer("TimeUsed")
    testTimed(1); testTimed(2)
    testTimer("Count")
    testTimer("TimeUsed")
```

`Tournament` *Deterministic tournament of size* `k`.

Description

`Tournament` is implemented in two steps:

1. A subset of size k of the population is selected with uniform probability.
2. A gene is selected with probability proportional to fitness.

Usage

```
Tournament(fit, lF)
```

Arguments

`fit`	Fitness vector.
`lF`	Local configuration.

Details

In each generation, the worst gene in a population dies.

Value

Index of the best candidate.

Examples

```
fit<-sample(10, 15, replace=TRUE)
Tournament(fit, NewlFselectGenes())
```

`TransformSelect` *Convert a selection function into a continuation.*

Description

`TransformSelect` precomputes the indices of genes to be selected and converts the selection function into an access function to the next index. The access function provides a periodic random index stream with a period length of the population size. In a genetic algorithm with a fixed size population, this avoids recomputation of the selection functions for each gene and its mate.

Usage

```
TransformSelect(fit, lF, SelectFUN)
```

Arguments

`fit`	Fitness vector.
`lF`	Local configuration.
`SelectFUN`	Selection function.

Details

The motivation for this transformation is:

1. We avoid the recomputation of potentially expensive selection functions. E.g. In population-based genetic algorithms, the selection function is computed twice per generation instead of more than generation times the population size.
2. No additional control flow is needed.
3. Dynamic reconfiguration is possible.
4. All selection functions have a common abstract interface and, therefore, can be overloaded by specialized concrete implementations. (Polymorphism).

The implementation idea is adapted from the continuation passing style in functional programming. See Reynolds, J. C. (1993).

Value

A function with a state which consists of the precomputed gene index vector, its `length`, and a `counter`. The function increments the counter in the state of its environment and returns the precomputed gene index at position `modulo((counter+1),length)` in the precomputed index vector in its environment. The function supports the same interface as a selection function.

Parallelization/Distribution

1. We use this tranformation if only the evaluation of genes should be parallelized/distributed.
2. If the complete replication of genes is parallelized, this transformation cannot be used in its current form. The current implementations of the selection functions can not easily be parallelized.

References

Reynolds, J. C. (1993): The discoveries of continuations. *LISP and Symbolic Computation* 6, 233-247. <doi:10.1007/BF01019459>

Examples

```
fit<-sample(10, 15, replace=TRUE)
newselect<-TransformSelect(fit, NewlFselectGenes(), SelectSUS)
newselect(fit, NewlFselectGenes())
newselect(fit, NewlFselectGenes(), 5)
newselect(fit, NewlFselectGenes(), 10)
newselect(fit, NewlFselectGenes(), 10)
```

`xegaSelectGene` *Package xegaSelectGene.*

Description

Selection functions for genetic algorithms.

Details

The `selectGene` package provides selection and scaling functions for genetic algorithms. All functions of this package are independent of the gene representation.

- Scaling functions and dispersion measures are in scaling.R
- Selection functions are in selectGene.R. For selection functions, a transformation to index access functions is provided (a limited form of function continuation).
- Benchmark functions for selection functions are in selectGeneBenchmark.R. Except for uniform selection, the continuation form of selection functions should be used.
- Evaluation functions are in evalGene.R.
- Counting and timing of function executions are provided by transformation functions in timer.R
- Problem environments for examples and unit tests for
  - function optimization: DeJongF4.R (stochastic functions) and Parabola2D.R (delayed execution for benchmarking of parallelism, deterministic function, deterministic function with early termination check, function with random failures)
  - combinatorial optimization: newTSP.R (for the traveling salesman problem).
  - boolean function learning: newXOR.R (for the XOR problem).

Interface Scaling Functions

All scaling functions must implement the following abstract interface:

`function name(fit, lF)`

Parameters

- `fit` A fitness vector.
- `lF` Local configuration.

Return Value

Scaled fitness vector.

Interface Dispersion Measures

All dispersion measure functions must implement the following abstract interface:

`function name(popstatvec)`

Parameters

- `popstatvec` Vector of population statistics.

  The internal state of the genetic algorithm is described by a matrix of the history of population statistics. Each row consists of 8 population statistics (mean, min, Q1, median, Q3, max, var, mad). A row is a vector of population statistics.

Return Value

Dispersion measure (real).

Interface Selection Functions

All selection functions must implement the following abstract interface:

`function name(fit, lF, size)`

Parameters

- `fit` a vector of fitness values.
- `lF` a local function list.
- `size` the number of indices returned.

Return Value

A vector of indices of length `size`.

All selection functions are implemented WITHOUT a default assignment to `lF`.

A missing configuration should raise an error!

The default value of `size` is `1`.

Constants

Some scaling and selection functions use constants which should be configured. We handle these constants by constant functions created by `parm(constant)`. We store all of these functions in the list of local functions `lF`. The rationale is to reduce the number of parameters of selection functions and to provide a uniform interface for selection functions.

Table of Scaling Constants

Constant	Default	Used in
lF$Offset	1	ScaleFitness
lF$ScalingExp	1	ScalingFitness, ThresholdScaleFitness
lF$ScalingExp2	1	ThresholdScaleFitness
lF$ScalingThreshold	1	ThresholdScaleFitness
lF$RDMWeight	1.0	ContinuousScaleFitness

lF$DRMin	0.5	DispersionRatio
lF$DRMax	2.0	DispersionRatio
lF$ScalingDelay	1	DispersionRatio

State Variable	Start Value	Used in
lF$RDM	1.0	ThresholdScaleFitness
		ContinuousScaleFitness
		xega::RunGA

Table of Selection Constants

Constant	Default	Used in
lF$SelectionContinuation	TRUE	xegaPopulation::xegaNextPopulation
lF$Offset	1	SelectPropFitOnLn
		SelectPropFit
		SelectPropFitM
		SelectPropFitDiffOnLn
		SelectPropFitDiff
		SUS
		SelectLinearRankTSR
lF$eps	0.01	SelectPropFitDiffM
lF$TournamentSize	2	Tournament
		SelectTournament
		STournament
		SelectSTournament
lF$SelectionBias	1.5	SelectLRSelective
lF$MaxTSR	1.5	SelectLinearRankTSR

Parallel/Distributed Execution

All selection functions in this package return

1. the index of a selected gene. The configured selection function is executed each time a gene must be selected in the gene replication process. This allows a parallelization/distribution of the complete gene replication process and the fitness evaluation. However, the price to pay is a recomputation of the selection algorithms for each gene and each mate (which may be costly). The execution time of Baker's SUS function explodes when used in this way.
2. a vector of indices of the selected genes. We compute a vector of indices for genes and their mates, and we replace the selection function with a quasi-continuation function with precomputed indices which when called, returns the next index. The selection computation is executed once for each generation without costly recomputation. The cost of selecting a gene

and its mate is the cost of indexing an integer in a vector. This version is faster for almost all selection functions (Sequential computation).

The parallelization of quasi-continuation function is not yet implemented.

Constant Functions for Configuration

The following constant functions are expected to be in the local function list lF.

- `Offset()` in `SelectPropFit`: Since all fitness values must be larger than 0, in case of negative fitness values, `Offset()` is the value of the minimum fitness value (default: 1).
- `Eps()` in `SelectPropFitDiff`: `Eps()` is a very small value to eliminate differences of 0.
- `TournamentSize()` in `SelectTournament`: Specifies the size of the tournament. Per default: 2.
- `SelectionBias()` in `SelectLinearRank`. This constant must be larger than 1.0 and usually should be set at most to 2.0. Increasing `SelectionBias()` increases selection pressure. Beyond 2.0, there is the danger of premature convergence.

Performance Measurement

The file `Timer.R`: Functions for timing and counting.

The file `selectGeneBenchmark.R`: A benchmark of selection functions.

Interface Function Evaluation and Methods

All evaluation functions must implement the following abstract interface:

`function` `name(gene, lF)`

Parameters

- `gene` a gene.
- `lF` a local function list.

Return Value

A gene.

The file `evalGene.R` contains different function evaluation methods.

1. `EvalGeneU` evaluates a gene unconditionally. (Default.)
2. `EvalGeneR` evaluates a gene unconditionally and allows the repair of the gene by the decoder.
3. `EvalGeneDet` memoizes the evaluation of a gene in the in the gene. Genes are evaluated only once. This leads to a performance improvement for deterministic functions.
4. `EvalGeneStoch` computes an incremental average of the value of a gene. The average converges to the true value as the number of repeated evaluations of a gene increases.

Gene Representation

A gene is a named list:

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

The Architecture of the xegaX-Packages

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

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

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

Copyright

(c) 2023 Andreas Geyer-Schulz

License

MIT

URL

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

Installation

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

Author(s)

Andreas Geyer-Schulz

Index