Package ‘ecr’

July 10, 2017

Title Evolutionary Computation in R

Description Framework for building evolutionary algorithms for both single- and multi-objective continuous or discrete optimization problems. A set of predefined evolutionary building blocks and operators is included. Moreover, the user can easily set up custom objective functions, operators, building blocks and representations sticking to few conventions. The package allows both a black-box approach for standard tasks (plug-and-play style) and a much more flexible white-box approach where the evolutionary cycle is written by hand.

Version 2.1.0

Encoding UTF-8

Date 2017-07-09

Maintainer Jakob Bossek <`j.bossek@gmail.com`>

License GPL-3

URL `https://github.com/jakobbossek/ecr2`

BugReports `https://github.com/jakobbossek/ecr2/issues`

Depends BBmisc (>= 1.6), smoof (>= 1.4), ParamHelpers (>= 1.1)

Imports checkmate (>= 1.1), parallelMap (>= 1.3), reshape2 (>= 1.4.1), ggplot2 (>= 1.0.0)

Suggests testthat (>= 0.9.1), knitr, rmarkdown

ByteCompile yes

LazyData yes

RoxygenNote 6.0.1

VignetteBuilder knitr

NeedsCompilation yes

Author Jakob Bossek [aut, cre, cph],
Michael H. Buselli [ctb, cph],
Wessel Dankers [ctb, cph],
Carlos M. Fonseca [ctb, cph],
Manuel Lopez-Ibanez [ctb, cph],
Luis Paquete [ctb, cph],
Joshua Knowles [ctb, cph],

Eckart Zitzler [ctb, cph],
Olaf Mersmann [ctb]

Repository CRAN

Date/Publication 2017-07-10 12:27:41 UTC

R topics documented:

`approximateNadirPoint` *Reference point approximations.*

Description

Helper functions to compute nadir or ideal point from sets of points, e.g., multiple approximation sets.

Usage

```
approximateNadirPoint(..., sets = NULL)

approximateIdealPoint(..., sets = NULL)
```

Arguments

`...`	`[matrix]` Arbirary number of matrizes.
`sets`	`[list]` List of matrizes. This is an alternative way of passing the sets. Can be used exclusively or combined with `...`.

Value

`numeric` Reference point.

`asemoa` *Implementation of the NSGA-II EMOA algorithm by Deb.*

Description

The AS-EMOA, short for aspiration set evolutionary multi-objective algorithm aims to incorporate expert knowledge into multi-objective optimization [1]. The algorithm expects an aspiration set, i.e., a set of reference points. It then creates an approximation of the pareto front close to the aspiration set utilizing the average Hausdorff distance.

Usage

```
asemoa(fitness.fun, n.objectives = NULL, minimize = NULL, n.dim = NULL,
  lower = NULL, upper = NULL, mu = 10L, aspiration.set = NULL,
  normalize.fun = NULL, dist.fun = ecr:::computeEuclideanDistance, p = 1,
  parent.selector = setup(selSimple), mutator = setup(mutPolynomial, eta =
  25, p = 0.2, lower = lower, upper = upper), recombinator = setup(recSBX, eta
  = 15, p = 0.7, lower = lower, upper = upper),
  terminators = list(stopOnIters(100L)))
```

Arguments

`fitness.fun`	`[function]` The fitness function.
`n.objectives`	`[integer(1)]` Number of objectives of `obj.fun`. Optional if `obj.fun` is a benchmark function from package **smoof**.

`minimize`	`[logical(n.objectives)]` Logical vector with ith entry `TRUE` if the ith objective of `fitness.fun` shall be minimized. If a single logical is passed, it is assumed to be valid for each objective.
`n.dim`	`[integer(1)]` Dimension of the decision space.
`lower`	`[numeric]` Vector of minimal values for each parameter of the decision space in case of float or permutation encoding. Optional if `obj.fun` is a benchmark function from package **smoof**.
`upper`	`[numeric]` Vector of maximal values for each parameter of the decision space in case of float or permutation encoding. Optional if `obj.fun` is a benchmark function from package **smoof**.
`mu`	`[integer(1)]` Population size. Default is 10.
`aspiration.set`	`[matrix]` The aspiration set. Each column contains one point of the set.
`normalize.fun`	`[function]` Function used to normalize fitness values of the individuals before computation of the average Hausdorff distance. The function must have the formal arguments “set” and “aspiration.set”. Default is `NULL`, i.e., no normalization at all.
`dist.fun`	`[function]` Distance function used internally by Hausdorff metric to compute distance between two points. Expects a single vector of coordinate-wise differences between points. Default is `computeEuclideanDistance`.
`p`	`[numeric(1)]` Parameter p for the average Hausdorff metric. Default is 1.
`parent.selector`	`[ecr_selector]` Selection operator which implements a procedure to copy individuals from a given population to the mating pool, i. e., allow them to become parents.
`mutator`	`[ecr_mutator]` Mutation operator of type `ecr_mutator`.
`recombinator`	`[ecr_recombinator]` Recombination operator of type `ecr_recombinator`.
`terminators`	`[list]` List of stopping conditions of type “ecr_terminator”. Default is to stop after 100 iterations.

Value

`ecr_multi_objective_result`

Note

This is a pure R implementation of the AS-EMOA algorithm. It hides the regular ecr interface and offers a more R like interface while still being quite adaptable.

References

[1] Rudolph, G., Schuetze, S., Grimme, C., Trautmann, H: An Aspiration Set EMOA Based on Averaged Hausdorff Distances. LION 2014: 153-156. [2] G. Rudolph, O. Schuetze, C. Grimme, and H. Trautmann: A Multiobjective Evolutionary Algorithm Guided by Averaged Hausdorff Distance to Aspiration Sets, pp. 261-273 in A.-A. Tantar et al. (eds.): Proceedings of EVOLVE - A bridge between Probability, Set Oriented Numerics and Evolutionary Computation V, Springer: Berlin Heidelberg 2014.

`computeAverageHausdorffDistance`
Average Hausdorff Distance computation.

Description

Computes the average Hausdroff distance measure between two point sets.

Usage

```
computeAverageHausdorffDistance(A, B, p = 1, normalize = FALSE,
  dist.fun = computeEuclideanDistance)
```

Arguments

`A`	`[matrix]` First point set (each column corresponds to a point).
`B`	`[matrix]` Second point set (each column corresponds to a point).
`p`	`[numeric(1)]` Parameter p of the average Hausdoff metric. Default is 1.
`normalize`	`[logical(1)]` Should the front be normalized on basis of `B`? Default is `FALSE`.
`dist.fun`	`[matrix]` Distance function to compute distance between points x and y. Expects a single numeric vector d with the coordinate-wise differences di = (xi - yi). Default is `computeEuclideanDist`.

Value

`numeric(1)` Average Hausdorff distance of sets `A` and `B`.

`computeCrowdingDistance` *Compute the crowding distance of a set of points.*

Description

The crowding distance is a measure of spread of solutions in the approximation of the Pareto front. It is used, e.g., in the NSGA-II algorithm as a second selection criterion.

Usage

```
computeCrowdingDistance(x)
```

Arguments

`x`	`[matrix]` Numeric matrix with each column representing a point.

Value

`numeric` Vector of crowding distance values.

References

K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Transactions on Evolutionary Computation In Evolutionary Computation, IEEE Transactions on, Vol. 6, No. 2. (07 April 2002), pp. 182-197, doi:10.1109/4235.996017

`computeDistanceFromPointToSetOfPoints` *Computes distance between a single point and set of points.*

Description

Helper to compute distance between a single point and a point set.

Usage

```
computeDistanceFromPointToSetOfPoints(a, B,
  dist.fun = computeEuclideanDistance)
```

Arguments

`a`	`[numeric(1)]` Point given as a numeric vector.
`B`	`[matrix]` Point set (each column corresponds to a point).
`dist.fun`	`[matrix]` Distance function to compute distance between points x and y. Expects a single numeric vector d with the coordinate-wise differences di = (xi - yi). Default is `computeEuclideanDist`.

Value

`numeric(1)`

---

computeGenerationalDistance

Computes Generational Distance.

Description

Helper to compute the Generational Distance (GD) between two sets of points.

Usage

```
computeGenerationalDistance(A, B, p = 1, normalize = FALSE,
  dist.fun = computeEuclideanDistance)
```

Arguments

`A`	`[matrix]` First point set (each column corresponds to a point).
`B`	`[matrix]` Second point set (each column corresponds to a point).
`p`	`[numeric(1)]` Parameter p of the average Hausdoff metric. Default is 1.
`normalize`	`[logical(1)]` Should the front be normalized on basis of `B`? Default is `FALSE`.
`dist.fun`	`[matrix]` Distance function to compute distance between points x and y. Expects a single numeric vector d with the coordinate-wise differences di = (xi - yi). Default is `computeEuclideanDist`.

Value

`numeric(1)`

computeHV	*Functions for the calculation of the dominated hypervolume (contribution).*

Description

The function `computeHV` computes the dominated hypervolume of a set of points given a reference set whereby `computeHVContr` computes the hypervolume contribution of each point.

If no reference point is given the nadir point of the set `x` is determined and a positive offset with default 1 is added. This is to ensure that the reference point dominates all of the points in the reference set.

Usage

```
computeHV(x, ref.point = NULL)

computeHVContr(x, ref.point = NULL, offset = 1)
```

Arguments

`x`	`[matrix]` Matrix of points (column-wise).
`ref.point`	`[numeric	NULL]` Reference point. Set to the maximum in each dimension by default if not provided.
`offset`	`[numeric(1)]` Offset to be added to each component of the reference point only in the case where no reference is provided and one is calculated automatically.

Value

`numeric(1)` Dominated hypervolume in the case of `computeHV` and the dominated hypervolume contributions for each point in the case of `computeHVContr`.

Note

: Keep in mind that this function assumes all objectives to be minimized. In case at least one objective is to be maximized the matrix `x` needs to be transformed accordingly in advance.

`computeInvertedGenerationalDistance` *Computes Inverted Generational Distance.*

Description

Helper to compute the Inverted Generational Distance (IGD) between two sets of points.

Usage

```
computeInvertedGenerationalDistance(A, B, p = 1, normalize = FALSE,
  dist.fun = computeEuclideanDistance)
```

Arguments

`A`	`[matrix]` First point set (each column corresponds to a point).
`B`	`[matrix]` Second point set (each column corresponds to a point).
`p`	`[numeric(1)]` Parameter p of the average Hausdoff metric. Default is 1.
`normalize`	`[logical(1)]` Should the front be normalized on basis of `B`? Default is `FALSE`.
`dist.fun`	`[matrix]` Distance function to compute distance between points x and y. Expects a single numeric vector d with the coordinate-wise differences di = (xi - yi). Default is `computeEuclideanDist`.

Value

`numeric(1)`

`dominated` *Check for pareto dominance.*

Description

These functions take a numeric matrix `x` where each column corresponds to a point and return a logical vector. The i-th position of the latter is `TRUE` if the i-th point is dominated by at least one other point for `dominated` and `FALSE` for `nonDominated`.

Usage

```
dominated(x)

nondominated(x)
```

Arguments

x	[matrix] Numeric (d x n) matrix where d is the number of objectives and n is the number of points.

Value

logical

---

dominates — *Dominance relation check.*

Description

Check if a vector dominates another (`dominates`) or is dominated by another (`isDominated`). There are corresponding infix operators `dominates` and `isDominatedBy`.

Usage

```
dominates(x, y)

isDominated(x, y)

x %dominates% y

x %isDominatedBy% y
```

Arguments

x	[numeric] First vector.
y	[numeric] Second vector.

Value

logical(1)

`doNondominatedSorting` *Fast non-dominated sorting algorithm.*

Description

Fast non-dominated sorting algorithm proposed by Deb. Non-dominated sorting expects a set of points and returns a set of non-dominated fronts. In short words this is done as follows: the non-dominated points of the entire set are determined and assigned rank 1. Afterwards all points with the current rank are removed, the rank is increased by one and the procedure starts again. This is done until the set is empty, i.~e., each point is assigned a rank.

Usage

```
doNondominatedSorting(x)
```

Arguments

`x`	`[matrix]` Numeric matrix of points. Each column contains one point.

Value

`list` List with the following components

- **ranks** Integer vector of ranks of length `ncol(x)`. The higher the rank, the higher the domination front the corresponding point is located on.
- **dom.counter** Integer vector of length `ncol(x)`. The i-th element is the domination number of the i-th point.

Note

This procedure is the key survival selection of the famous NSGA-II multi-objective evolutionary algorithm (see `nsga2`).

References

[1] Deb, K., Pratap, A., and Agarwal, S. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation, 6 (8) (2002), 182-197.

ecr *Interface to ecr similar to the optim function.*

Description

The most flexible way to setup evolutionary algorithms with ecr is by explicitely writing the evolutionary loop utilizing various ecr utlity functions. However, in everyday life R users frequently need to optimize a single-objective R function. The `ecr` function thus provides a more R like interface for single objective optimization similar to the interface of the `optim` function.

Usage

```
ecr(fitness.fun, minimize = NULL, n.objectives = NULL, n.dim = NULL,
  lower = NULL, upper = NULL, n.bits, representation, mu, lambda,
  perm = NULL, p.recomb = 0.7, p.mut = 0.3, survival.strategy = "plus",
  n.elite = 0L, custom.constants = list(), log.stats = list(fitness =
  list("min", "mean", "max")), log.pop = FALSE, monitor = NULL,
  initial.solutions = NULL, parent.selector = NULL,
  survival.selector = NULL, mutator = NULL, recombinator = NULL,
  terminators = list(stopOnIters(100L)), ...)
```

Arguments

`fitness.fun`	`[function]` The fitness function.
`minimize`	`[logical(n.objectives)]` Logical vector with ith entry `TRUE` if the ith objective of `fitness.fun` shall be minimized. If a single logical is passed, it is assumed to be valid for each objective.
`n.objectives`	`[integer(1)]` Number of objectives of `obj.fun`. Optional if `obj.fun` is a benchmark function from package **smoof**.
`n.dim`	`[integer(1)]` Dimension of the decision space.
`lower`	`[numeric]` Vector of minimal values for each parameter of the decision space in case of float or permutation encoding. Optional if `obj.fun` is a benchmark function from package **smoof**.
`upper`	`[numeric]` Vector of maximal values for each parameter of the decision space in case of float or permutation encoding. Optional if `obj.fun` is a benchmark function from package **smoof**.
`n.bits`	`[integer(1)]` Number of bits to use for binary representation.

`representation`	`[character(1)]` Genotype representation of the parameters. Available are “binary”, “float”, “permutation” and “custom”.
`mu`	`[integer(1)]` Number of individuals in the population.
`lambda`	`[integer(1)]` Number of individuals generated in each generation.
`perm`	`[integer(1)	vector]` Either a single integer number. In this case the set is assumed to be `1:perm`. Alternatively, a set, i.e., a vector of elements can be passed which should form each individual.
`p.recomb`	`[numeric(1)]` Probability of two parents to perform crossover. Default is 0.7.
`p.mut`	`[numeric(1)]` Probability to apply mutation to a child. Default is 0.1.
`survival.strategy`	`[character(1)]` Determines the survival strategy used by the EA. Possible are “plus” for a classical (mu + lambda) strategy and “comma” for (mu, lambda). Default is “plus”.
`n.elite`	`[integer(1)]` Number of fittest individuals of the current generation that shall be copied to the next generation without changing. Keep in mind, that the algorithm does not care about this option if the `survival.strategy` is set to ’plus’. Default is 0.
`custom.constants`	`[list]` Additional constants which should be available to all generators and operators. Defaults to empty list.
`log.stats`	`[list]` (Named) list of scalar functions to compute statistics on the fitness values in each generation. See `initLogger` for more information. Default is to log fitness minimum, mean and maximum values.
`log.pop`	`[logical(1)]` Shall the entire population be saved in each generation? Default is `FALSE`.
`monitor`	`[function]` Monitoring function. Default is `NULL`, i.e. no monitoring.
`initial.solutions`	`[list]` List of individuals which should be placed in the initial population. If the number of passed individuals is lower than `mu`, the population will be filled up by individuals generated by the corresponding generator. Default is `NULL`, i.e., the entire population is generated by the population generator.
`parent.selector`	`[ecr_selector]` Selection operator which implements a procedure to copy individuals from a given population to the mating pool, i. e., allow them to become parents.

`survival.selector`	[`ecr_selector`] Selection operator which implements a procedure to extract individuals from a given set, which should survive and set up the next generation.
`mutator`	[`ecr_mutator`] Mutation operator of type `ecr_mutator`.
`recombinator`	[`ecr_recombinator`] Recombination operator of type `ecr_recombinator`.
`terminators`	[`list`] List of stopping conditions of type “ecr_terminator”. Default is to stop after 100 iterations.
`...`	[any] Further arguments passed down to `fitness.fun`.

Value

`ecr_result`

Examples

```
fn = function(x) {
   sum(x^2)
}
lower = c(-5, -5); upper = c(5, 5)
res = ecr(fn, n.dim = 2L, n.objectives = 1L, lower = lower, upper = lower,
 representation = "float", mu = 20L, lambda = 10L,
  mutator = setup(mutGauss, lower = lower, upper = upper))
```

`ecr_parallelization` *Parallelization in ecr*

Description

In ecr it is possible to parallelize the fitness function evaluation to make use, e.g., of multiple CP cores or nodes in a HPC cluster. For maximal flexibility this is realized by means of the **parallelMap** package (see the official GitHub page for instructions on how to set up parallelization). The different levels of parallelization can be specified in the `parallelStart*` function. At them moment only the level “ecr.evaluateFitness” is supported.

Keep in mind that parallelization comes along with some overhead. Thus activating parallelization, e.g., for evaluation a fitness function which is evaluated lightning-fast, may result in higher computation time. However, if the function evaluations are computationally more expensive, parallelization leads to significant running time benefits.

`ecr_result` *Result object.*

Description

S3 object returned by `ecr` containing the best found parameter setting and value in the single-objective case and the Pareto-front/-set in case of a multi-objective optimization problem. Moreover a set of further information, e.g., reason of termination, the control object etc. are returned.

The single objective result object contains the following fields:

task The `ecr_optimization_task`.

best.x Overall best parameter setting.

best.y Overall best objective value.

log Logger object.

last.population Last population.

last.fitness Numeric vector of fitness values of the last population.

message Character string describing the reason of termination.

In case of a solved multi-objective function the result object contains the following fields:

task The `ecr_optimization_task`.

log Logger object.

pareto.idx Indizes of the non-dominated solutions in the last population.

pareto.front (n x d) matrix of the approximated non-dominated front where n is the number of non-dominated points and d is the number of objectives.

pareto.set Matrix of decision space values resulting with objective values given in pareto.front.

last.population Last population.

message Character string describing the reason of termination.

`emoaIndEps` *Computation of the unary epsilon-indicator.*

Description

Functions for the computation of unary and binary measures which are useful for the evaluation of the performace of EMOAs. See the references section for literature on these indicators.

Given a set of points `points`, `emoaIndEps` computes the unary epsilon-indicator provided a set of reference points `ref.points`.

The `emoaIndHV` function computes the hypervolume indicator Hyp(X, R, r). Given a set of point X (`points`), another set of reference points R (`ref.points`) (which maybe the true Pareto front) and a reference point r (`ref.point`) it is defined as Hyp(X, R, r) = HV(R, r) - HV(X, r).

Usage

```
emoaIndEps(points, ref.points)

emoaIndHV(points, ref.points, ref.point = NULL)

emoaIndR1(points, ref.points, ideal.point = NULL, nadir.point = NULL,
  lambda = NULL, utility = "tschebycheff")

emoaIndR2(points, ref.points, ideal.point = NULL, nadir.point = NULL,
  lambda = NULL, utility = "tschebycheff")

emoaIndR3(points, ref.points, ideal.point = NULL, nadir.point = NULL,
  lambda = NULL, utility = "tschebycheff")
```

Arguments

`points`	`[matrix]` Matrix of points.
`ref.points`	`[matrix]` Set of reference points.
`ref.point`	`[numeric]` A single reference point used, e.g., for the computation of the hypervolume indicator via `emoaIndHV`. If `NULL` the nadir point of the union of the `points` and `ref.points` is used.
`ideal.point`	`[numeric]` The utopia point of the true Pareto front, i.e., each component of the point contains the best value if the other objectives are neglected.
`nadir.point`	`[numeric]` Nadir point of the true Pareto front.
`lambda`	`[integer(1)]` Number of weight vectors to use in estimating the utility function.
`utility`	`[character(1)]` Name of the utility function to use. Must be one of “weightedsum”, “tschebycheff” or “augmented tschbycheff”.

Value

`numeric(1)` Epsilon indicator.

`evaluateFitness` *Computes the fitness value(s) for each individual of a given set.*

Description

This function expects a list of individuals, computes the fitness and always returns a matrix of fitness values; even in single-objective optimization a (1 x n) matrix is returned for consistency, where n is the number of individuals. This function makes use of `parallelMap` to parallelize the fitness evaluation.

Usage

```
evaluateFitness(control, inds, ...)
```

Arguments

`control`	[ecr_control] Control object.
`inds`	[list] List of individuals.
`...`	[any] Optional parameters passed down to fitness function.

Value

`matrix` .

`generateOffspring` *Helper functions for offspring generation*

Description

Function `mutate` expects a control object, a list of individuals, and a mutation probability. The mutation operator registered in the control object is then applied with the given probability to each individual. Function `recombinate` expects a control object, a list of individuals as well as their fitness matrix and creates `lambda` offspring individuals by recombining parents from `inds`. Which parents take place in the parent selection depends on the `parent.selector` registered in the control object. Finally, function `generateOffspring` is a wrapper for both `recombinate` and `mutate`. Both functions are applied subsequently to generate new individuals by variation and mutation.

Usage

```
generateOffspring(control, inds, fitness, lambda, p.recomb = 0.7,
  p.mut = 0.1)

mutate(control, inds, p.mut = 0.1, slot = "mutate", ...)

recombinate(control, inds, fitness, lambda = length(inds), p.recomb = 0.7,
  slot = "recombine", ...)
```

Arguments

`control`	`[ecr_control]` Control object.
`inds`	`[list]` List of individuals.
`fitness`	`[matrix]` Matrix of fitness values (each column contains the fitness value(s) of one individual).
`lambda`	`[integer(1)]` Number of individuals generated in each generation.
`p.recomb`	`[numeric(1)]` Probability of two parents to perform crossover. Default is 0.7.
`p.mut`	`[numeric(1)]` Probability to apply mutation to a child. Default is 0.1.
`slot`	`[character(1)]` The slot of the control object which contains the registered operator to use. Default is “mutate” for `mutate` and “recombine” for `recombinate`. In most cases there is no need to change this. However, it might be useful if you make use of different mutation operators registerted, e.g., in the slots “mutate1” and “mutate2”.
`...`	[any] Furhter arguments passed down to recombinator/mutator. There parameters will overwrite parameters in `par.list`.

Value

`list` List of individuals.

`generators` *Population generators*

Description

Utility functions to build a set of individuals. The function `gen` expects an R expression and a number `n` in order to create a list of `n` individuals based on the given expression. Functions `genBin`, `genPerm` and `genReal` are shortcuts for initializing populations of binary strings, permutations or real-valued vectors respectively.

Usage

```
gen(expr, n)

genBin(n, n.dim)

genPerm(n, n.dim)

genReal(n, n.dim, lower, upper)
```

Arguments

`expr`	[R expression] Expression to generate a single individual.
`n`	`[integer(1)]` Number of individuals to create.
`n.dim`	`[integer(1)]` Dimension of the decision space.
`lower`	`[numeric]` Vector of minimal values for each parameter of the decision space in case of float encoding.
`upper`	`[numeric]` Vector of maximal values for each parameter of the decision space in case of float encoding.

Value

`list`

`getFront` *Extract fitness values from Pareto archive.*

Description

Get all non-dominated points in objective space, i.e., an (m x n) matrix of fitness with m being the number of objectives and n being the number of non-dominated points in the Pareto archive.

Usage

```
getFront(x)
```

Arguments

`x`	`[ecr_pareto_archive]` Pareto archive.

Value

`matrix`

`getIndividuals` *Extract individuals from Pareto archive.*

Description

Get the non-dominated individuals logged in the Pareto archive.

Usage

```
getIndividuals(x)
```

Arguments

`x`	`[ecr_pareto_archive]` Pareto archive.

Value

`list`

See Also

Other ParetoArchive: `getSize`, `initParetoArchive`, `updateParetoArchive`

`getPopulations` *Access to logged populations.*

Description

Simple getter for the logged populations.

Usage

```
getPopulations(log, trim = TRUE)
```

Arguments

`log`	`[ecr_logger]` The log generated by `initLogger`.
`trim`	`[logical(1)]` Should unused lines in the logged be cut off? Usually one wants this behaviour. Thus the default is `TRUE`.

Details

This functions throws an error if the logger was initialized with `log.pop = FALSE` (see `initLogger`).

Value

`list`	List of populations.

See Also

Other logging: `getStatistics`, `initLogger`, `updateLogger`

`getSize` *Get size of Pareto-archive.*

Description

Returns the number of stored individuals in Pareto archive.

Usage

```
getSize(x)
```

Arguments

`x`	`[ecr_pareto_archive]` Pareto archive.

Value

`integer(1)`

See Also

Other ParetoArchive: `getIndividuals`, `initParetoArchive`, `updateParetoArchive`

`getStatistics` *Access the logged statistics.*

Description

Simple getter for the logged fitness statistics.

Usage

```
getStatistics(log, trim = TRUE)
```

Arguments

`log`	`[ecr_logger]` The log generated by `initLogger`.
`trim`	`[logical(1)]` Should unused lines in the logged be cut off? Usually one wants this behaviour. Thus the default is `TRUE`.

Value

`data.frame` Logged statistics.

See Also

Other logging: `getPopulations`, `initLogger`, `updateLogger`

getSupportedRepresentations

Get supported representations.

Description

Returns the character vector of representation which the operator supports.

Usage

```
getSupportedRepresentations(operator)
```

Arguments

`operator`	`[ecr_operator]` Operator object.

Value

`character` Vector of representation types.

initECRControl *Control object generator.*

Description

The control object keeps information on the objective function and a set of evolutionary components, i.e., operators.

Usage

```
initECRControl(fitness.fun, n.objectives = NULL, minimize = NULL)
```

Arguments

`fitness.fun`	`[function]` The fitness function.
`n.objectives`	`[integer(1)]` Number of objectives of `obj.fun`. Optional if `obj.fun` is a benchmark function from package **smoof**.
`minimize`	`[logical(n.objectives)]` Logical vector with ith entry `TRUE` if the ith objective of `fitness.fun` shall be minimized. If a single logical is passed, it is assumed to be valid for each objective.

Value

`ecr_control`

initLogger *Initialize a log object.*

Description

Logging is a central aspect of each EA. Besides the final solution(s) especially in research often we need to keep track of different aspects of the evolutionary process, e.g., fitness statistics. The logger of ecr keeps track of different user-defined statistics and the population. It may also be used to check stopping conditions (see `makeECRTerminator`). Most important this logger is used internally by the `ecr` black-box interface.

Usage

```
initLogger(control, log.stats = list(fitness = list("min", "mean", "max")),
  log.extras = NULL, log.pop = FALSE, init.size = 1000L)
```

Arguments

`control`	`[ecr_control]` Control object.
`log.stats`	`[list]` List of lists for statistic computation on attributes of the individuals of the population. Each entry should be named by the attribute it should be based on, e.g., fitness, and should contain a list of R functions as a character string or a a list with elements `fun` for the function, and `pars` for additional parameters which shall be passed to the corresponding function. Each function is required to return a scalar numeric value. By default the minimum, mean and maximum of the fitness values is computed. Since fitness statistics are the most important ones these do not have to be stored as attributes, but can be passed as a matrix to the update function.
`log.extras`	`[character]` Possibility to instruct the logger to store additional scalar values in each generation. Named character vector where the names indicate the value to store and the value indicates the corresponding data types. Currently we support all atomic modes of `vector` expect “factor” and “raw”.
`log.pop`	`[logical(1)]` Shall the entire population be saved in each generation? Default is `FALSE`.
`init.size`	`[integer(1)]` Initial number of rows of the slot of the logger, where the fitness statistics are stored. The size of the statistics log is doubled each time an overflow occurs. Default is 1000.

Value

`ecr_logger` An S3 object of class `ecr_logger` with the following components:

log.stats The `log.stats` list.

log.pop The `log.pop` parameter.

init.size Initial size of the log.

env The actual log. This is an R environment which ensures, that in-place modification is possible.

Note

Statistics are logged in a `data.frame`.

See Also

Other logging: `getPopulations`, `getStatistics`, `updateLogger`

Examples

```
control = initECRControl(function(x) sum(x), minimize = TRUE,
  n.objectives = 1L)
control = registerECROperator(control, "mutate", mutBitflip, p = 0.1)
control = registerECROperator(control, "selectForMating", selTournament, k = 2)
control = registerECROperator(control, "selectForSurvival", selGreedy)

log = initLogger(control,
  log.stats = list(
    fitness = list("mean", "myRange" = function(x) max(x) - min(x)),
    age = list("min", "max")
  ), log.pop = TRUE, init.size = 1000L)

 # simply pass stuff down to control object constructor
population = initPopulation(mu = 10L, genBin, n.dim = 10L)
fitness = evaluateFitness(control, population)

# append fitness to individuals and init age
for (i in seq_along(population)) {
  attr(population[[i]], "fitness") = fitness[, i]
  attr(population[[i]], "age") = 1L
}

for (iter in seq_len(10)) {
  # generate offspring
  offspring = generateOffspring(control, population, fitness, lambda = 5)
  fitness.offspring = evaluateFitness(control, offspring)

  # update age of population
  for (i in seq_along(population)) {
    attr(population[[i]], "age") = attr(population[[i]], "age") + 1L
  }

  # set offspring attributes
  for (i in seq_along(offspring)) {
    attr(offspring[[i]], "fitness") = fitness.offspring[, i]
    # update age
    attr(offspring[[i]], "age") = 1L
  }

  sel = replaceMuPlusLambda(control, population, offspring)

  population = sel$population
  fitness = sel$fitness

  # do some logging
  updateLogger(log, population, n.evals = 5)
}
head(getStatistics(log))
```

`initParetoArchive` *Initialize Pareto Archive.*

Description

A Pareto archive is usually used to store all / a part of the non-dominated points stored during a run of an multi-objective evolutionary algorithm.

Usage

```
initParetoArchive(control, max.size = Inf, trunc.fun = NULL)
```

Arguments

`control`	`[ecr_control]` Control object.
`max.size`	`[integer(1)]` Maximum capacity of the Pareto archive, i.e., the maximal number of non-dominated points which can be stored in the archive. Default is `Inf`, i.e., (theoretically) unbounded capacity.
`trunc.fun`	`[function(archive, inds, fitness, ...)]` In case the archive is limited in capacity, i.e., `max.size` is not infinite, this function is called internally if an archive overflow occurs. This function expects the `archive`, a list of individuals `inds`, a matrix of fitness values (each column contains the fitness value(s) of one individual) `fitness` and further optional arguments `...` which may be used by the internals of `trunc.fun`. The function must return a list with components “fitness” and “inds” which shall be the subsets of `fitness` and `inds` respectively, which should be kept by the archive.

Value

`ecr_pareto_archive`

See Also

Other ParetoArchive: `getIndividuals`, `getSize`, `updateParetoArchive`

`initPopulation` *Helper function to build initial population.*

Description

Generates the initial population. Optionally a set of initial solutions can be passed.

Usage

```
initPopulation(mu, gen.fun, initial.solutions = NULL, ...)
```

Arguments

`mu`	`[integer(1)]` Number of individuals in the population.
`gen.fun`	`[function]` Function used to generate initial solutions, e.g., `genBin`.
`initial.solutions`	`[list]` List of individuals which should be placed in the initial population. If the number of passed individuals is lower than `mu`, the population will be filled up by individuals generated by the corresponding generator. Default is `NULL`, i.e., the entire population is generated by the population generator.
`...`	[any] Further parameters passed to `gen.fun`.

Value

`ecr_population`

`is.supported` *Check if ecr operator supports given representation.*

Description

Check if the given operator supportds a certain representation, e.g., "float".

Usage

```
is.supported(operator, representation)
```

Arguments

`operator`	`[ecr_operator]` Object of type `ecr_operator`.
`representation`	`[character(1)]` Representation as a string.

Value

`logical(1)` `TRUE`, if operator supports the representation type.

`isEcrOperator` *Check if given function is an ecr operator.*

Description

Checks if the passed object is of type `ecr_operator`.

Usage

```
isEcrOperator(obj)
```

Arguments

`obj`	[any] Arbitrary R object to check.

Value

`logical(1)`

`makeECRMonitor` *Factory method for monitor objects.*

Description

Monitor objects serve for monitoring the optimization process, e.g., printing some status messages to the console. Each monitor includes the functions `before`, `step` and `after`, each being a function and expecting a log `log` of type `ecr_logger` and `...` as the only parameters. This way the logger has access to the entire log.

Usage

```
makeECRMonitor(before = NULL, step = NULL, after = NULL, ...)
```

Arguments

`before`	`[function]` Function called one time after initialization of the EA.
`step`	`[function]` Function applied after each iteration of the algorithm.
`after`	`[function]` Function applied after the EA terminated.
`...`	[any] Not used.

Value

`ecr_monitor` Monitor object.

`makeMutator` *Construct a mutation operator.*

Description

Helper function which constructs a mutator, i. e., a mutation operator.

Usage

```
makeMutator(mutator, supported = getAvailableRepresentations())
```

Arguments

`mutator`	`[function]` Actual mutation operator.
`supported`	`[character]` Vector of strings/names of supported parameter representations. Possible choices: “permutation”, “float”, “binary” or “custom”.

Value

`ecr_mutator` Mutator object.

`makeOperator` *Construct evolutionary operator.*

Description

Helper function which constructs an evolutionary operator.

Usage

```
makeOperator(operator, supported = getAvailableRepresentations())
```

Arguments

`operator`	`[function]` Actual operator.
`supported`	`[character]` Vector of names of supported parameter representations. Possible choices: “permutation”, “float”, “binary” or “custom”.

Value

`ecr_operator` Operator object.

Note

In general you will not need this function, but rather one of its deriviatives like `makeMutator` or `makeSelector`.

`makeOptimizationTask` *Creates an optimization task.*

Description

An optimization task consists of the fitness/objective function, the number of objectives, the "direction" of optimization, i.e., which objectives should be minimized/maximized and the names of the objectives.

Usage

```
makeOptimizationTask(fun, n.objectives = NULL, minimize = NULL,
  objective.names = NULL)
```

Arguments

`fun`	[`function` | `smoof_function`] Fitness/objective function.
`n.objectives`	[`integer(1)`] Number of objectives. This must be a positive integer value unless `fun` is of type `smoof_function`.
`minimize`	[`logical`] A logical vector indicating which objectives to minimize/maximize. By default all objectives are assumed to be minimized.
`objective.names`	[`character`] Names for the objectuves. Default is `NULL`. In this case the names are set to y1, ..., yn with n equal to `n.objectives` and simply y in the single-objective case.

Value

`ecr_optimization_task`

makeRecombinator *Construct a recombination operator.*

Description

Helper function which constructs a recombinator, i. e., a recombination operator.

Usage

```
makeRecombinator(recombinator, supported = getAvailableRepresentations(),
  n.parents = 2L, n.children = NULL)
```

Arguments

`recombinator`	`[function]` Actual mutation operator.
`supported`	`[character]` Vector of strings/names of supported parameter representations. Possible choices: “permutation”, “float”, “binary” or “custom”.
`n.parents`	`[integer(1)]` Number of parents supported.
`n.children`	`[integer(1)]` How many children does the recombinator produce? Default is `1`.

Value

`ecr_recombinator` Recombinator object.

Note

If a recombinator returns more than one child, the `multiple.children` parameter needs to be `TRUE`, which is the default. In case of multiple children produced these have to be placed within a list.

makeSelector *Construct a selection operator.*

Description

Helper function which defines a selector method, i. e., an operator which takes the population and returns a part of it for mating or survival.

Usage

```
makeSelector(selector, supported = getAvailableRepresentations(),
  supported.objectives, supported.opt.direction = "minimize")
```

Arguments

`selector`	`[function]` Actual selection operator.
`supported`	`[character]` Vector of strings/names of supported parameter representations. Possible choices: “permutation”, “float”, “binary” or “custom”.
`supported.objectives`	`[character]` At least one of “single-objective” or “multi-objective”.
`supported.opt.direction`	`[character(1-2)]` Does the selector work for maximization tasks xor minimization tasks or both? Default is “`minimize`”, which means that the selector selects in favour of low fitness values.

Value

`ecr_selector` Selector object.

`makeTerminator` *Generate stopping condition.*

Description

Wrap a function within a stopping condition object.

Usage

```
makeTerminator(condition.fun, name, message)
```

Arguments

`condition.fun`	`[function]` Function which takes a logger object `log` (see `initLogger`) and returns a single logical.
`name`	`[character(1)]` Identifier for the stopping condition.
`message`	`[character(1)]` Message which should be stored in the termination object, if the stopping condition is met.

Value

`ecr_terminator`

mutBitflip — *Bitplip mutator.*

Description

This operator works only on binary representation and flips each bit with a given probability $p \in (0, 1)$. Usually it is recommended to set $p = \frac{1}{n}$ where n is the number of bits in the representation.

Usage

```
mutBitflip(ind, p = 0.1)
```

Arguments

ind	[binary] Binary vector, i.e., vector with elements 0 and 1 only.
p	[numeric(1)] Probability to flip a single bit. Default is `0.1`.

Value

binary

See Also

Other mutators: `mutGauss`, `mutInsertion`, `mutPolynomial`, `mutScramble`, `mutSwap`, `mutUniform`

mutGauss — *Gaussian mutator.*

Description

Default Gaussian mutation operator known from Evolutionary Algorithms. This mutator is applicable only for `representation="float"`. Given an individual $\mathbf{x} \in R^l$ this mutator adds a Gaussian distributed random value to each component of $\mathbf{x}$, i.~e., $\tilde{\mathbf{x}}_i = \mathbf{x}_i + \sigma\mathcal{N}(0, 1)$.

Usage

```
mutGauss(ind, p = 1L, sdev = 0.05, lower, upper)
```

Arguments

`ind`	`[numeric]` Numeric vector / individual to mutate.
`p`	`[numeric(1)]` Probability of mutation for the gauss mutation operator.
`sdev`	`[numeric(1)` Standard deviance of the Gauss mutation, i. e., the mutation strength.
`lower`	`[numeric]` Vector of minimal values for each parameter of the decision space.
`upper`	`[numeric]` Vector of maximal values for each parameter of the decision space.

Value

`numeric`

See Also

Other mutators: `mutBitflip`, `mutInsertion`, `mutPolynomial`, `mutScramble`, `mutSwap`, `mutUniform`

`mutInsertion` *Insertion mutator.*

Description

The Insertion mutation operator selects a position random and inserts it at a random position.

Usage

```
mutInsertion(ind)
```

Arguments

`ind`	`[integer]` Permutation of integers, i.e., vector of integer values.

Value

`integer`

See Also

Other mutators: `mutBitflip`, `mutGauss`, `mutPolynomial`, `mutScramble`, `mutSwap`, `mutUniform`

`mutInversion` *Inversion mutator.*

Description

The Inversion mutation operator selects two positions within the chromosome at random and inverts the elements inbetween.

Usage

```
mutInversion(ind)
```

Arguments

`ind`	`[integer]` Permutation of integers, i.e., vector of integer values.

Value

`integer`

`mutPolynomial` *Polynomial mutation.*

Description

Performs an polynomial mutation as used in the SMS-EMOA algorithm.

Usage

```
mutPolynomial(ind, p = 0.2, eta = 10, lower, upper)
```

Arguments

`ind`	`[numeric]` Numeric vector / individual to mutate.
`p`	`[numeric(1)]` Probability of mutation of each gene.
`eta`	`[numeric(1)` Distance parameter of the mutation distribution.
`lower`	`[numeric]` Vector of minimal values for each parameter of the decision space.
`upper`	`[numeric]` Vector of maximal values for each parameter of the decision space.

Value

`numeric`

See Also

Other mutators: `mutBitflip`, `mutGauss`, `mutInsertion`, `mutScramble`, `mutSwap`, `mutUniform`

`mutScramble` *Scramble mutator.*

Description

The Scramble mutation operator selects two positions within the chromosome at random and randomly intermixes the subsequence between these positions.

Usage

```
mutScramble(ind)
```

Arguments

`ind`	`[integer]` Permutation of integers, i.e., vector of integer values.

Value

`integer`

See Also

Other mutators: `mutBitflip`, `mutGauss`, `mutInsertion`, `mutPolynomial`, `mutSwap`, `mutUniform`

`mutSwap` *Swap mutator.*

Description

Chooses two positions at random and swaps the genes.

Usage

```
mutSwap(ind)
```

Arguments

`ind`	`[integer]` Permutation of integers, i.e., vector of integer values.

Value

`integer`

See Also

Other mutators: `mutBitflip`, `mutGauss`, `mutInsertion`, `mutPolynomial`, `mutScramble`, `mutUniform`

`mutUniform` *Uniform mutator.*

Description

This mutation operator works on real-valued genotypes only. It selects a position in the solution vector at random and replaced it with a uniformally chosen value within the box constraints of the corresponding parameter. This mutator may proof beneficial in early stages of the optimization process, since it distributes points widely within the box constraints and thus may hinder premature convergence. However, in later stages - when fine tuning is necessary, this feature is disadvantegous.

Usage

```
mutUniform(ind, lower, upper)
```

Arguments

`ind`	`[numeric]` Numeric vector / individual to mutate.
`lower`	`[numeric]` Vector of minimal values for each parameter of the decision space.
`upper`	`[numeric]` Vector of maximal values for each parameter of the decision space.

Value

`numeric`

See Also

Other mutators: `mutBitflip`, `mutGauss`, `mutInsertion`, `mutPolynomial`, `mutScramble`, `mutSwap`

normalizeFront *Normalize points of a set.*

Description

Normalization is done by subtracting the `min.value` for each dimension and dividing by the `max.value` for each dimension by default.

Usage

```
normalizeFront(A, min.value = NULL, max.value = NULL)
```

Arguments

`A`	`[matrix]` Point set (each column corresponds to a point).
`min.value`	`[numeric]` Vector of minimal values of length `nrow(A)`. Default is the row-wise minimum of `A`.
`max.value`	`[numeric]` Vector of maximal values of length `nrow(A)`. Default is the row-wise maximum of `A`.

Value

`matrix` Normalized front.

nsga2 *Implementation of the NSGA-II EMOA algorithm by Deb.*

Description

The NSGA-II merges the current population and the generated offspring and reduces it by means of the following procedure: It first applies the non dominated sorting algorithm to obtain the nondominated fronts. Starting with the first front, it fills the new population until the i-th front does not fit. It then applies the secondary crowding distance criterion to select the missing individuals from the i-th front.

Usage

```
nsga2(fitness.fun, n.objectives = NULL, n.dim = NULL, minimize = NULL,
  lower = NULL, upper = NULL, mu = 100L, lambda = mu,
  mutator = setup(mutPolynomial, eta = 25, p = 0.2, lower = lower, upper =
  upper), recombinator = setup(recSBX, eta = 15, p = 0.7, lower = lower, upper
  = upper), terminators = list(stopOnIters(100L)), ...)
```

Arguments

`fitness.fun`	`[function]` The fitness function.
`n.objectives`	`[integer(1)]` Number of objectives of `obj.fun`. Optional if `obj.fun` is a benchmark function from package **smoof**.
`n.dim`	`[integer(1)]` Dimension of the decision space.
`minimize`	`[logical(n.objectives)]` Logical vector with ith entry `TRUE` if the ith objective of `fitness.fun` shall be minimized. If a single logical is passed, it is assumed to be valid for each objective.
`lower`	`[numeric]` Vector of minimal values for each parameter of the decision space in case of float or permutation encoding. Optional if `obj.fun` is a benchmark function from package **smoof**.
`upper`	`[numeric]` Vector of maximal values for each parameter of the decision space in case of float or permutation encoding. Optional if `obj.fun` is a benchmark function from package **smoof**.
`mu`	`[integer(1)]` Number of individuals in the population. Default is 100.
`lambda`	`[integer(1)]` Offspring size, i.e., number of individuals generated by variation operators in each iteration. Default is 100.
`mutator`	`[ecr_mutator]` Mutation operator of type `ecr_mutator`.
`recombinator`	`[ecr_recombinator]` Recombination operator of type `ecr_recombinator`.
`terminators`	`[list]` List of stopping conditions of type “ecr_terminator”. Default is to stop after 100 iterations.
`...`	[any] Further arguments passed down to fitness function.

Value

`ecr_multi_objective_result`

Note

This is a pure R implementation of the NSGA-II algorithm. It hides the regular ecr interface and offers a more R like interface while still being quite adaptable.

References

Deb, K., Pratap, A., and Agarwal, S. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation, 6 (8) (2002), 182-197.

`plotFront` *Plot Pareto-front.*

Description

Plots a scatterplot of non-dominated points in the objective space utilizing the **ggplot2** package. The function returns a `ggplot` object which can be furhter modified via additional ggplot layers. If the passed object is a `data.frame`, each line is considered to contain the fitness values of one individual. Contrary, if a matrix is passed, it is considered to be passed in ecr2 format, i.e., each column corresponds to one point. The matrix is then transposed and converted to a `data.frame`.

Usage

```
plotFront(x, obj.names = NULL, minimize = TRUE)
```

Arguments

`x`	`[matrix	data.frame]` Object which contains the non-dominated points.
`obj.names`	`[character]` Optional objectives names. Default is `c("f1", "f2")`.
`minimize`	`[logical]` Logical vector with ith entry `TRUE` if the ith objective shall be minimized. If a single logical is passed, it is assumed to be valid for each objective. If the matrix is of type `ecr_fitness_matrix` (this is the case if it is produced by one of ecr2's utility functions, e.g. `evaluateFitness`), the appended `minimize` attribute is the default.

Value

`ggplot` **ggplot** object.

Note

At the moment only two-dimensional objective spaces are supported.

Examples

```
matrix
```

`plotStatistics` *Generate line plot of logged statistics.*

Description

Expects a data.frame of logged statistics, e.g., extracted from a logger object by calling `getStatistics`, and generates a basic line plot. The plot is generated with the **ggplot2** package and the ggplot object is returned.

Usage

```
plotStatistics(x, drop.stats = character(0L))
```

Arguments

`x`	`[ecr_statistics	ecr_logger]` Logger object or statistics data frame from logger object.
`drop.stats`	`[character]` Names of logged statistics to be dropped. Default is the empty character, i.e., not to drop any stats.

`recCrossover` *One-point crossover recombinator.*

Description

The one-point crossover recombinator is defined for float and binary representations. Given two real-valued/binary vectors of length n, the selector samples a random position i between 1 and n-1. In the next step it creates two children. The first part of the first child contains of the subvector from position 1 to position i of the first parent, the second part from position i+1 to n is taken from the second parent. The second child is build analogously. If the parents are list of real-valued/binary vectors, the procedure described above is applied to each element of the list.

Usage

```
recCrossover(inds)
```

Arguments

`inds`	`[list]` Parents, i.e., list of exactly two numeric or binary vectors of equal length.

Value

`list`

See Also

Other recombinators: `recIntermediate`, `recOX`, `recPMX`, `recSBX`, `recUnifCrossover`

`recIntermediate` *Indermediate recombinator.*

Description

Intermediate recombination computes the component-wise mean value of the `k` given parents. It is applicable only for float representation.

Usage

```
recIntermediate(inds)
```

Arguments

`inds`	`[list]` Parents, i.e., list of exactly two numeric vectors of equal length.

Value

`numeric` Single offspring.

See Also

Other recombinators: `recCrossover`, `recOX`, `recPMX`, `recSBX`, `recUnifCrossover`

`recOX` *Ordered-Crossover (OX) recombinator.*

Description

This recombination operator is specifically designed for permutations. The operators chooses two cut-points at random and generates two offspring as follows: a) copy the subsequence of one parent and b) remove the copied node indizes from the entire sequence of the second parent from the sescond cut point and b) fill the remaining gaps with this trimmed sequence.

Usage

```
recOX(inds)
```

Arguments

`inds`	`[list]` Parents, i.e., list of exactly two permutations (vectors of integer values) of equal length.

Value

`list`

See Also

Other recombinators: `recCrossover`, `recIntermediate`, `recPMX`, `recSBX`, `recUnifCrossover`

`recPMX` *Partially-Mapped-Crossover (PMX) recombinator.*

Description

This recombination operator is specifically designed for permutations. The operators chooses two cut-points at random and generates two offspring as follows: a) copy the subsequence of one parent and b) fill the remaining positions while preserving the order and position of as many genes as possible.

Usage

```
recPMX(inds)
```

Arguments

`inds`	`[numeric]` Parents, i.e., list of exactly two permutations of equal length.

Value

`ecr_recombinator`

See Also

Other recombinators: `recCrossover`, `recIntermediate`, `recOX`, `recSBX`, `recUnifCrossover`

`recSBX` *Simulated Binary Crossover (SBX) recombinator.*

Description

The Simulated Binary Crossover was first proposed by [1]. It i suited for float representation only and creates two offspring. Given parents p_1, p_2 the offspring are generated as $c_{1/2} = \bar{x} \pm \frac{1}{2}\beta(p_2 - p_1)$ where $\bar{x} = \frac{1}{2}(p_1 + p_2), p_2 > p_1$. This way $\bar{c} = \bar{x}$ is assured.

Usage

```
recSBX(inds, eta = 5, p = 1, lower, upper)
```

Arguments

`inds`	`[numeric]` Parents, i.e., list of exactly two numeric vectors of equal length.
`eta`	`[numeric(1)]` Parameter eta, i.e., the distance parameters of the crossover distribution.
`p`	`[numeric(1)]` Crossover probability for each gene. Default is `1.0`.
`lower`	`[numeric]` Vector of minimal values for each parameter of the decision space.
`upper`	`[numeric]` Vector of maximal values for each parameter of the decision space.

Value

`ecr_recombinator`

Note

This is the default recombination operator used in the NSGA-II EMOA (see `nsga2`).

References

[1] Deb, K. and Agrawal, R. B. (1995). Simulated binary crossover for continuous search space. Complex Systems 9(2), 115-148.

See Also

Other recombinators: `recCrossover`, `recIntermediate`, `recOX`, `recPMX`, `recUnifCrossover`

`recUnifCrossover` *Uniform crossover recombinator.*

Description

Produces two child individuals. The i-th gene is from parent1 with probability `p` and from parent2 with probability `1-p`.

Usage

```
recUnifCrossover(inds, p = 0.5)
```

Arguments

`inds`	`[list]` Parents, i.e., list of exactly two numeric or binary vectors of equal length.
`p`	`[numeric(1)]` Probability to select gene from parent1.

Value

`list`

See Also

Other recombinators: `recCrossover`, `recIntermediate`, `recOX`, `recPMX`, `recSBX`

`registerECROperator` *Register operators to control object.*

Description

In ecr the control object stores information on the fitness function and serves as a storage for evolutionary components used by your evluationary algorithm. This function handles the registration process.

Usage

```
registerECROperator(control, slot, fun, ...)
```

Arguments

`control`	`[ecr_control]` Control object.
`slot`	`[character(1)]` Name of the field in the control object where to store the operator.
`fun`	`[function]` Actual operator. In order to use the various helper functions of ecr one needs to stick to a simple convention: The first argument of `function` should be the individual to mutate, a list of individuals for recombination or a matrix of fitness values for recombination. If one does not want to use the corresponding helpers, e.g., `mutate`, the signature of the function does not matter. However, in this case you are responsable to pass arguments correctly.
`...`	[any] Further arguments for `fun`. These arguments are stored in the control object and passed on to `fun`.

Value

`ecr_control`

replace *(mu + lambda) selection*

Description

Takes a population of mu individuals and another set of lambda offspring individuals and selects mu individuals out of the union set according to the survival selection strategy stored in the control object.

Usage

```
replaceMuPlusLambda(control, population, offspring, fitness = NULL,
  fitness.offspring = NULL)

replaceMuCommaLambda(control, population, offspring, fitness = NULL,
  fitness.offspring = NULL, n.elite = base::max(ceiling(length(population *
  0.1)), 1L))
```

Arguments

`control`	`[ecr_control]` Control object.
`population`	`[list]` Current set of individuals.
`offspring`	`[list]` Another set of individuals.
`fitness`	`[matrix]` Matrix of fitness values for the individuals from `population`. This is only optional in the case that each individual in `population` has an attribute “fitness”.
`fitness.offspring`	`[matrix]` Matrix of fitness values for the individuals from `offspring`. This is only optional in the case that each individual in `offspring` has an attribute “fitness”.
`n.elite`	`[integer(1)]` Number of fittest individuals of the current generation that shall be copied to the next generation without changing. Keep in mind, that the algorithm does not care about this option if the `survival.strategy` is set to ’plus’. Default is 0.

Value

`list` List with selected population and corresponding fitness matrix.

`selDomHV` *Dominated Hypervolume selector.*

Description

Performs nondominated sorting and drops the individual from the last front with minimal hypervolume contribution.

Usage

```
selDomHV(fitness, n.select, ref.point)
```

Arguments

`fitness`	`[matrix]` Matrix of fitness values (each column contains the fitness value(s) of one individual).
`n.select`	`[integer(1)]` Number of elements to select.
`ref.point`	`[numeric]` Reference point for hypervolume computation.

Value

`integer` Vector of survivor indizes.

See Also

Other selectors: `selGreedy`, `selNondom`, `selRoulette`, `selSimple`, `selTournament`

`select` *Select individuals.*

Description

This utility functions expect a control object, a matrix of fitness values - each column containing the fitness value(s) of one individual - and the number of individuals to select. The corresponding selector, i.e., mating selector for `selectForMating` or survival selector for `selectForSurvival` is than called internally and a vector of indizes of selected individuals is returned.

Usage

```
selectForMating(control, fitness, n.select)

selectForSurvival(control, fitness, n.select)
```

Arguments

`control`	`[ecr_control]` Control object.
`fitness`	`[matrix]` Matrix of fitness values (each column contains the fitness value(s) of one individual).
`n.select`	`[integer(1)]` Number of individuals to select.

Details

Both functions check the optimization directions stored in the task inside the control object, i.e., whether to minimize or maximize each objective, and transparently prepare/transform the `fitness` matrix for the selector.

Value

`integer` Integer vector with the indizes of selected individuals.

`selGreedy` *Simple selector.*

Description

Sorts the individuals according to their fitness value in increasing order and selects the best ones.

Usage

```
selGreedy(fitness, n.select)
```

Arguments

`fitness`	`[matrix]` Matrix of fitness values (each column contains the fitness value(s) of one individual).
`n.select`	`[integer(1)]` Number of elements to select.

Value

`integer` Vector of survivor indizes.

See Also

Other selectors: `selDomHV`, `selNondom`, `selRoulette`, `selSimple`, `selTournament`

selNondom — *Non-dominated sorting selector.*

Description

Applies nondominated sorting of the objective and subsequent crowding distance criterion to select a subset of individuals. This is the selector used by the NSGA-II EMOA (see `nsga2`).

Usage

```
selNondom(fitness, n.select)
```

Arguments

`fitness`	`[matrix]` Matrix of fitness values (each column contains the fitness value(s) of one individual).
`n.select`	`[integer(1)]` Number of elements to select.

Value

`setOfIndividuals`

See Also

Other selectors: `selDomHV`, `selGreedy`, `selRoulette`, `selSimple`, `selTournament`

selRoulette — *Roulette-wheel / fitness-proportional selector.*

Description

The chance of an individual to get selected is proportional to its fitness, i.e., better individuals get a higher chance to take part in the reproduction process. Low-fitness individuals however, have a positive fitness as well.

Usage

```
selRoulette(fitness, n.select, offset = 0.1)
```

Arguments

`fitness`	`[matrix]` Matrix of fitness values (each column contains the fitness value(s) of one individual).
`n.select`	`[integer(1)]` Number of elements to select.
`offset`	`[numeric(1)]` In case of negative fitness values all values are shifted towards positive values by adding the negative of the minimal fitness value. However, in this case the minimal fitness value after the shifting process is zero. The `offset` is a positive numeric value which is added additionally to each shifted fitness value. This way even the individual with the smallest fitness value has a positive porbability to be selected. Default is `0.1`.

Details

Fitness proportional selection can be naturally applied to single objective maximization problems. However, negative fitness values can are problematic. The Roulette-Wheel selector thus works with the following heuristic: if negative values occur, the negative of the smallest fitness value is added to each fitness value. In this case to avoid the smallest shifted fitness value to be zero and thus have a zero probability of being selected an additional positive constant `offset` is added (see parameters).

Value

`setOfIndividuals`

See Also

Other selectors: `selDomHV`, `selGreedy`, `selNondom`, `selSimple`, `selTournament`

`selSimple` *Simple (naive) selector.*

Description

Just for testing. Actually does not really select, but instead returns a random sample of `ncol(fitness)` indizes.

Usage

```
selSimple(fitness, n.select)
```

Arguments

`fitness`	`[matrix]` Matrix of fitness values (each column contains the fitness value(s) of one individual).
`n.select`	`[integer(1)]` Number of elements to select.

Value

`setOfIndividuals`

See Also

Other selectors: `selDomHV`, `selGreedy`, `selNondom`, `selRoulette`, `selTournament`

`selTournament` *k-Tournament selector.*

Description

k individuals from the population are chosen randomly and the best one is selected to be included into the mating pool. This process is repeated until the desired number of individuals for the mating pool is reached.

Usage

```
selTournament(fitness, n.select, k = 3L)
```

Arguments

`fitness`	`[matrix]` Matrix of fitness values (each column contains the fitness value(s) of one individual).
`n.select`	`[integer(1)]` Number of elements to select.
`k`	`[integer(1)]` Number of individuals to participate in each tournament. Default is `2L`.

Value

`integer` Vector of survivor indizes.

See Also

Other selectors: `selDomHV`, `selGreedy`, `selNondom`, `selRoulette`, `selSimple`

setup — *Set up parameters for evolutionary operator.*

Description

This function builds a simple wrapper around an evolutionary operator, i.e., mutator, recombinator or selector and defines its parameters. The result is a function that does not longer depend on the parameters. E.g., `fun = setup(mutBitflip, p = 0.3)` initializes a bitflip mutator with mutation probability 0.3. Thus, the following calls have the same behaviour: `fun(c(1, 0, 0))` and `mutBitflip(fun(c(1, 0, 0), p = 0.3)`. Basically, this type of preinitialization is only neccessary if operators with additional parameters shall be initialized in order to use the black-box `ecr`.

Usage

```
setup(operator, ...)
```

Arguments

`operator`	[ecr_operator] Evolutionary operator.
`...`	[any] Furhter parameters for `operator`.

Value

`function` Wrapper evolutionary operator with parameters `x` and `...`.

Examples

```
# initialize bitflip mutator with p = 0.3
bf = setup(mutBitflip, p = 0.3)
# sample binary string
x = sample(c(0, 1), 100, replace = TRUE)

set.seed(1)
# apply preinitialized function
print(bf(x))

set.seed(1)
# apply raw function
print(mutBitflip(x, p = 0.3))

# overwrite preinitialized values with mutate
ctrl = initECRControl(fitness.fun = function(x) sum(x), n.objectives = 1L)
# here we define a mutation probability of 0.3
ctrl = registerECROperator(ctrl, "mutate", setup(mutBitflip, p = 0.3))
# here we overwrite with 1, i.e., each bit is flipped
print(x)
print(mutate(ctrl, list(x), p.mut = 1, p = 1)[[1]])
```

setupECRDefaultMonitor *Default monitor.*

Description

Default monitor object that outputs messages to the console based on a default logger (see `initLogger`).

Usage

```
setupECRDefaultMonitor(step = 10L)
```

Arguments

`step`	`[integer(1)]` Number of steps of the EA between monitoring. Default is 10.

Value

`ecr_monitor`

smsemoa *Implementation of the SMS-EMOA by Emmerich et al.*

Description

Pure R implementation of the SMS-EMOA. This algorithm belongs to the group of indicator based multi-objective evolutionary algorithms. In each generation, the SMS-EMOA selects two parents uniformly at, applies recombination and mutation and finally selects the best subset of individuals among all subsets by maximizing the Hypervolume indicator.

Usage

```
smsemoa(fitness.fun, n.objectives = NULL, n.dim = NULL, minimize = NULL,
  lower = NULL, upper = NULL, mu = 100L, ref.point = NULL,
  mutator = setup(mutPolynomial, eta = 25, p = 0.2, lower = lower, upper =
  upper), recombinator = setup(recSBX, eta = 15, p = 0.7, lower = lower, upper
  = upper), terminators = list(stopOnIters(100L)), ...)
```

Arguments

`fitness.fun`	`[function]` The fitness function.
`n.objectives`	`[integer(1)]` Number of objectives of `obj.fun`. Optional if `obj.fun` is a benchmark function from package **smoof**.
`n.dim`	`[integer(1)]` Dimension of the decision space.
`minimize`	`[logical(n.objectives)]` Logical vector with ith entry `TRUE` if the ith objective of `fitness.fun` shall be minimized. If a single logical is passed, it is assumed to be valid for each objective.
`lower`	`[numeric]` Vector of minimal values for each parameter of the decision space in case of float or permutation encoding. Optional if `obj.fun` is a benchmark function from package **smoof**.
`upper`	`[numeric]` Vector of maximal values for each parameter of the decision space in case of float or permutation encoding. Optional if `obj.fun` is a benchmark function from package **smoof**.
`mu`	`[integer(1)]` Number of individuals in the population. Default is 100.
`ref.point`	`[numeric]` Reference point for the hypervolume computation. Default is (11, ..., 11)' with the corresponding dimension.
`mutator`	`[ecr_mutator]` Mutation operator of type `ecr_mutator`.
`recombinator`	`[ecr_recombinator]` Recombination operator of type `ecr_recombinator`.
`terminators`	`[list]` List of stopping conditions of type “ecr_terminator”. Default is to stop after 100 iterations.
`...`	[any] Further arguments passed down to fitness function.

Value

`ecr_multi_objective_result`

Note

This helper function hides the regular ecr interface and offers a more R like interface of this state of the art EMOA.

References

Beume, N., Naujoks, B., Emmerich, M., SMS-EMOA: Multiobjective selection based on dominated hypervolume, European Journal of Operational Research, Volume 181, Issue 3, 16 September 2007, Pages 1653-1669.

`stoppingConditions` *Stopping conditions*

Description

Stop the EA after a fixed number of fitness function evaluations, after a predefined number of generations/itereations or if the known optimal function value is approximated (only for single-objective optimization).

Usage

```
stopOnEvals(max.evals = NULL)

stopOnIters(max.iter = NULL)

stopOnOptY(opt.y, eps)
```

Arguments

`max.evals`	`[integer(1)]` Maximal number of function evaluations. Default ist `Inf`.
`max.iter`	`[integer(1)]` Maximal number of iterations. Default ist `Inf`.
`opt.y`	`[numeric(1)]` Optimal scalar fitness function value.
`eps`	`[numeric(1)]` Stop if absolute deviation from `opt.y` is lower than `eps`.

Value

`ecr_terminator`

toGG *Transform to long format.*

Description

Transform the data.frame of logged statistics from wide to **ggplot2**-friendly long format.

Usage

```
toGG(x, drop.stats = character(0L))
```

Arguments

`x`	`[ecr_statistics	ecr_logger]` Logger object or statistics data frame from logger object.
`drop.stats`	`[character]` Names of logged statistics to be dropped. Default is the empty character, i.e., not to drop any stats.

Value

`data.frame`

updateLogger *Update the log.*

Description

This function modifies the log in-place, i.e., without making copies.

Usage

```
updateLogger(log, population, fitness = NULL, n.evals, extras = NULL, ...)
```

Arguments

`log`	`[ecr_logger]` The log generated by `initLogger`.
`population`	`[list]` List of individuals.
`fitness`	`[matrix]` Optional matrix of fitness values (each column contains the fitness value(s) for one individual) of `population`. If no matrix is passed and the log shall store information of the fitness, each individual needs to have an attribute fitness.

`n.evals`	[integer(1)] Number of fitness function evaluations performed in the last generation.
`extras`	[list] Optional named list of additional scalar values to log. See `log.extras` argument of `initLogger` for details.
`...`	[any] Furhter arguments. Not used at the moment.

See Also

Other logging: `getPopulations`, `getStatistics`, `initLogger`

---

updateParetoArchive *Update Pareto Archive.*

Description

This function updates a Pareto archive, i.e., an archive of non-dominated points. It expects the archive, a set of individuals, a matrix of fitness values (each column corresponds to the fitness vector of one individual) and updates the archive "in-place". If the archive has unlimited capacity all non-dominated points of the union of archive and passed individuals are stored. Otherwise, i.e., in case the archive is limited in capacity (argument `max.size` of `initParetoArchive` was set to an integer value greater zero), the `trunc.fun` function passed to `initParetoArchive` is applied to all non-dominated points to determine which points should be dropped.

Usage

```
updateParetoArchive(archive, inds, fitness, ...)
```

Arguments

`archive`	[ecr_pareto_archive] The archive generated by `initParetoArchive`.
`inds`	[list] List of individuals.
`fitness`	[matrix] Matrix of fitness values (each column contains the fitness value(s) for one individual) of `inds`.
`...`	[any] Furhter arguments passed down to `trunc.fun` (set via `initParetoArchive`).

See Also

Other ParetoArchive: `getIndividuals`, `getSize`, `initParetoArchive`

which.dominated *Determine which points of a set are (non)dominated.*

Description

Given a matrix with one point per column `which.dominated` returns the column numbers of the dominated points and `which.nondominated` the column numbers of the nondominated points. Function `isMaximallyDominated` returns a logical vector with `TRUE` for each point which does not dominate any other point.

Usage

```
which.dominated(x)

which.nondominated(x)

isMaximallyDominated(x)
```

Arguments

`x`	`[matrix]` Numeric (n x d) matrix where n is the number of points and d is the number of objectives.

Value

`integer`

Examples

```
data(mtcars)
# assume we want to maximize horsepower and minimize gas consumption
cars = mtcars[, c("mpg", "hp")]
cars$hp = -cars$hp
idxs = which.nondominated(as.matrix(cars))
print(mtcars[idxs, ])
```

wrapChildren *Wrap the individuals constructed by a recombination operator.*

Description

Should be used if the recombinator returns multiple children.

Usage

```
wrapChildren(...)
```

Arguments

...	[any] Individuals.

Value

`list` List of individuals.

Index