Package ‘MOEADr’

October 24, 2017

Type Package

Title Component-Wise MOEA/D Implementation

Description Modular implementation of Multiobjective Evolutionary Algorithms based on Decomposition (MOEA/D) [Zhang and Li (2007), <DOI:10.1109/TEVC.2007.892759>] for quick assembling and testing of new algorithmic components, as well as easy replication of published MOEA/D proposals.

Version 1.1.0

Date 2017-10-21

Imports FNN, assertthat

Suggests smoof, scatterplot3d, MASS, grDevices, irace, testthat, knitr, rmarkdown, emoa, ggplot2, reshape2

Depends R (>= 3.4.0)

Author Felipe Campelo [aut, cre],
Lucas Batista [com],
Claus Aranha [aut]

Maintainer Felipe Campelo <`fcampelo@ufmg.br`>

License GPL-2

LazyData TRUE

Encoding UTF-8

RoxygenNote 6.0.1

VignetteBuilder knitr

URL `https://github.com/fcampelo/MOEADr`

NeedsCompilation no

Repository CRAN

Date/Publication 2017-10-24 07:14:58 UTC

R topics documented:

box_constraints *Box constraints routine*

Description

Calculates the constraint values and violations when only box constraints are present.

Usage

```
box_constraints(X, ...)
```

Arguments

`X`	Population matrix of the MOEA/D (each row is a candidate solution). If `NULL` the function searches for `X` in the calling environment.
`...`	other parameters (unused, included for compatibility with generic call)

Details

This routine calculates the constraint values and violations for a population matrix in the MOEA/D. Each row of the matrix is considered as a candidate solution. This routine expects the candidate solutions to be standardized, i.e., that the variable limits given in `problem$xmin` and `problem$xmax` are mapped to `0` and `1`, respectively.

Value

List objective containing a matrix of constraint values `Cmatrix`, a matrix of individual constraint violations `Vmatrix`, and a vector of total constraint violations `v`.

`calcIGD` *Inverted Generational Distance*

Description

Calculate IGD

Usage

```
calcIGD(Y, Yref)
```

Arguments

`Y`	Matrix of points in the objective space
`Yref`	Matrix of Pareto-optimal reference points

Value

igd value (scalar)

`check_stop_criteria` *Stop criteria for MOEA/D*

Description

Verifies stop criteria for the MOEADr package.

Usage

```
check_stop_criteria(stopcrit, call.env)
```

Arguments

`stopcrit`	list containing the parameters defining the stop handling method. See Section `Constraint Handling` of the `moead()` documentation for details.
`call.env`	List vector containing the stop criteria to be used. See `moead()` for details.

Details

This routine is intended to be used internally by `moead()`, and should not be called directly by the user.

Value

Flag `keep.running`, indicating whether the algorithm should continue (`TRUE`) or terminate (`FALSE`).

`constraint_none` *NULL constraint handling method for MOEA/D*

Description

Construct the preference index matrix based only on performance values.

Usage

```
constraint_none(B, bigZ, bigV, ...)
```

Arguments

`B`	Matrix of neighborhoods (generated by `define_neighborhood(...)`))
`bigZ`	Matrix of scalarized objective values for each neighborhood and the incumbent solution (generated by `scalarize_values`)
`bigV`	Matrix of violation values for each neighborhood and the incumbent solution
`...`	other parameters (unused, included for compatibility with generic call)

Details

This function ignores the violation values when constructing the preference index matrix, using only the scalarized performance values.

Value

`[ N x (T+1) ]` matrix of preference indices. Each row `i` contains a permutation of `{1, 2, ..., (T+1)}`, where `1,...,T` correspond to the solutions contained in the neighborhood of the i-th subproblem, `B[i, ]`, and `T+1` corresponds to the incumbent solution for that subproblem. The order of the permutation is defined by the increasing values of `f(xk)`, where `f(xk)` is the aggregation function value of the k-th solution being compared.

`constraint_penalty` *"Penalty" constraint handling method for MOEA/D*

Description

Uses the Penalty Function constraint handling method to generate a preference index for the MOEADr framework.

Usage

```
constraint_penalty(B, bigZ, bigV, beta, ...)
```

Arguments

`B`	Matrix of neighborhoods (generated by `define_neighborhood()`$B)
`bigZ`	Matrix of scalarized objective values for each neighborhood and the incumbent solution (generated by `scalarize_values()`)
`bigV`	Matrix of violation values for each neighborhood and the incumbent solution (generated in `order_neighborhood()`)
`beta`	Penalization constant (non-negative value)
`...`	other parameters (unused, included for compatibility with generic call)

Details

This function calculates the preference index of a set of neighborhoods based on the "penalty" constraint handling method. Please see `order_neighborhood()` for more information on the preference index matrix.

Value

`[ N x (T+1) ]` matrix of preference indices. Each row `i` contains a permutation of `{1, 2, ..., (T+1)}`, where `1,...,T` correspond to the solutions contained in the neighborhood of the i-th subproblem, `B[i, ]`, and `T+1` corresponds to the incumbent solution for that subproblem. The order of the permutation is defined by the increasing values of `f(xk) + beta * v(xk)`, where `f(xk)` is the aggregation function value of the k-th solution being compared, and v(xk) is its total constraint violation (calculated in `evaluate_population()`$V$v).

`constraint_vbr` *"Violation-based Ranking" constraint handling method for MOEA/D*

Description

Uses the Violation-based Ranking handling method to generate a preference index for the MOEADr framework.

Usage

```
constraint_vbr(bigZ, bigV, type = c("ts", "sr", "vt"), pf = NULL, ...)
```

Arguments

`bigZ`	Matrix of scalarized objective values for each neighborhood and the incumbent solution (generated by `scalarize_values()`)
`bigV`	Matrix of violation values for each neighborhood and the incumbent solution (generated in `order_neighborhood()`)
`type`	type of `c(x)` function to use (see `c(x) Criteria` for details).
`pf`	probability parameter for type = "sr" (ignored in other modes).
`...`	other parameters (unused, included for compatibility with generic call)

Details

This function calculates the preference index of a set of neighborhoods based on the "violation-based ranking" (VBR) constraint handling method. Please see `order_neighborhood()` for more information on the preference index matrix.

The VBR strategy generalizes some well-known methods for handling constraints in population-based metaheuristics (see Section `c(x) Criteria`). This strategy essentially ranks points within for a given subproblem based on their aggregated function value (`f^{agg}(x|w_i)`) or their total constraint violation (`v(x)`). Specific variations of this strategy differ on the criteria for using one or the other.

The value used for ranking a given point `x` can be summarized as:

Violation	c(x) criterion	Rank using:
`v(x) = 0`	`c(x) = *`	`f^{agg}(x|w_i)`
`v(x) > 0`	`c(x) == TRUE`	`f^{agg}(x|w_i)`
`v(x) > 0`	`c(x) == FALSE`	`v(x)`

Points compared according to their `f^{agg}(x|w_i)` values (i.e., feasible points and those for which `c(x) = TRUE`) are ranked first (i.e., receive ranks between `1` and `n_{feas}`, where `n_{feas}` is the number of feasible points in the i-th neighborhood), with points that are compared according to their `v(x)` values receiving ranks between `(n_{feas} + 1)` and `T + 1` (`T` being the size of the neighborhood. The `+1` comes from including the incumbent solution in the comparison).

Value

`[ N x (T+1) ]` matrix of preference indices. Each row `i` contains a permutation of `{1, 2, ..., (T+1)}`, where `1,...,T` correspond to the solutions contained in the neighborhood of the i-th subproblem, `B[i, ]`, and `T+1` corresponds to the incumbent solution for that subproblem. The order of the permutation is defined by the specific strategy defined by the input variable `type`).

c(x) Criteria

Specific variations of the VBR differ on how the criterion c(x) is implemented. Three variants are currently implemented in the MOEADr package:

Method	ID	c(x)
Tournament Selection `[Deb2000]`	`$type = "ts"`	`FALSE`
Stochastic Ranking `[Runarsson2000]`	`$type = "sr"`	`runif() < pf`
Violation Threshold `[Asafuddoula2014]`	`$type = "vt"`	`v(x) < eps_v^i`

where $pf \in [0, 1]$ is a user-defined parameter for the "sr" method, and `eps_v^i` is subproblem-dependent, adaptive quantity calculated internally in the routine (see `[Asafuddoula2014]` and `[Campelo2017]` for details).

Using an External Archive

For types "sr" and "vt", it is possible for the algorithm to lose feasible solutions during its update step, since there is a non-zero probability of unfeasible solutions replacing feasible ones. In these cases, it is recommended to set the `moead()` parameter `update$UseArchive = TRUE`, so that an external archive is built with the best feasible solutions found for each subproblem.

References

`[Deb2000]` K. Deb, "An efficient constraint handling method for genetic algorithm", Computer Methods in Applied Mechanics and Engineering 186(2–4):311–338, 2000.

`[Runarsson2000]` T. Runarsson, X. Yao, "Stochastic ranking for constrained evolutionary optimization", IEEE Transactions on Evolutionary Computation4(3):284–294, 2000.

`[Asafuddoula2014]` M. Asafuddoula, T. Ray, R. Sarker, K. Alam, "An adaptive constraint handling approach embedded MOEA/D," 2012 IEEE Congress on Evolutionary Computation (CEC).

`[Campelo2017]` F. Campelo, L.S. Batista, C. Aranha, "A Component-Wise Perspective on Multi-objective Evolutionary Algorithms based on Decomposition". In preparation, 2017.

`create_population` *Create population*

Description

Create a population for the MOEADr package

Usage

```
create_population(N, problem)
```

Arguments

`N`	population size
`problem`	list of named problem parameters. See Section `Problem Description` of the `moead()` documentation for details.

Details

This routine creates a population matrix for the MOEA/D. Currently only a multivariate uniform distribution is implemented. All points are created within the standardized space $0 \leq x_i \leq 1, i = 1, ..., n_v$.

Value

A population matrix X for the MOEA/D.

Examples

```
ex.problem <- list(name = "example_problem",
                   xmin = rep(-1, 5),
                   xmax = rep(1, 5),
                   m    = 2)
X <- create_population(20, ex.problem)
```

---

decomposition_msld *Problem Decomposition using Multi-layered Simplex-lattice Design*

Description

Problem Decomposition using Multi-layered Simplex-lattice Design for MOEADr package

Usage

```
decomposition_msld(decomp, ...)
```

Arguments

`decomp`	list containing the relevant decomposition parameters. Besides `decomp$name = "msld"`, this method requires the definition of the following key-value pairs in `decomp`: • `decomp$H`: array of positive integers representing the `H` values to be used by the SLD decomposition at each layer (see `decomposition_sld()` for details). • `decomp$tau`: array of scale multipliers for each layer, $0 < \tau_i \leq 1$, $\tau_i ! = \tau_j$ for all $i! = j$. Must have the same length as `decomp$H`. • `decomp$.nobj`: integer value, `decomp$.nobj > 1`. Number of objectives of the problem.
`...`	other parameters (included for compatibility with generic call)

Details

This routine calculates the weight vectors for the MOEA/D using the Multi-layered Simplex-lattice Design.

References

K. Li et al. (2014), "An Evolutionary Many-Objective Optimization Algorithm Based on Dominance and Decomposition", IEEE Trans. Evol. Comp. 19(5):694-716, 2015. DOI: 10.1109/TEVC.2014.2373386

Examples

```
decomp <- list(name = "msld", H = c(5, 3), tau = c(.9, .5), .nobj = 4)
W <- decomposition_msld(decomp)
```

decomposition_sld *Problem Decomposition using Simplex-lattice Design*

Description

Problem Decomposition using Simplex-lattice Design for MOEADr package

Usage

```
decomposition_sld(decomp, ...)
```

Arguments

`decomp`	list containing the relevant decomposition parameters. Besides `decomp$name = "sld"`, this method requires the definition of the following key-value pairs: • `decomp$H`, decomposition constant. Suggested values for `decomp$H` are (use with caution): `m	H	N` `2	99	100` `3	12	91` `5	6	210` It is important to highlight that the number of vectors generated (`N`) must be greater than the number of neighbors declared in `neighbors$T` (see `moead()` for details). • `decomp$.nobj`: integer value, `decomp$.nobj > 1`. Number of objectives of the problem.
`...`	other parameters (included for compatibility with generic call)

Details

This routine calculates the weight vectors for the MOEA/D using the Simplex-lattice Design.

References

I. Das, J. Dennis (1998), "Normal Boundary Intersection - A New Method for Generating the Pareto Surface in Nonlinear Multicriteria Optimization Problems", SIAM J. Optim., 8(3), 631-657. DOI: 10.1137/S1052623496307510

Examples

```
decomp <- list(name = "sld", H = 99, .nobj = 2)
W <- decomposition_sld(decomp)
```

`decomposition_uniform` *Problem Decomposition using Uniform Design*

Description

Problem Decomposition using Uniform Design for MOEADr package

Usage

```
decomposition_uniform(decomp, ...)
```

Arguments

`decomp`	list containing the relevant decomposition parameters. Besides `decomp$name = "uniform"`, this method requires the definition of the following key-value pairs: • `decomp$N`, number of subproblems to generate. It is important to highlight that the number of subproblems must be greater than the number of neighbors declared in `neighbors$T` (see `moead()` for details). • `decomp$.nobj`: integer value, `decomp$.nobj > 1`. Number of objectives of the problem.
`...`	other parameters (included for compatibility with generic call)

Details

This routine calculates the weight vectors for the MOEA/D using the Uniform Design:

References

R. Wang, T. Zhang, B. Guo, "An enhanced MOEA/D using uniform directions and a pre-organization procedure". Proc. IEEE Congress on Evolutionary Computation, Cancun, Mexico, 2013, pp. 2390–2397.

Examples

```
decomp <- list(name = "uniform", N = 50, .nobj = 3)
W <- decomposition_uniform(decomp)
```

`define_neighborhood` *Calculate neighborhood relations*

Description

Calculates neighborhood relations for the MOEADr package

Usage

```
define_neighborhood(neighbors, v.matrix, iter)
```

Arguments

`neighbors`	List containing the decomposition method parameters. This list must contain the following key-value pairs: • `neighbors$name`, type of neighborhood to use. The following types are currently available: – `neighbors$name = "lambda"`: defines the neighborhood using the distance matrix for the weight vectors. The calculation is performed only once for the entire run. – `neighbors$name = "x"`: defines the neighborhood using the distance matrix for the incumbent solution associated with each subproblem. In this case the calculation is performed at each iteration. • `neighbors$T`: Neighborhood size. The value of `neighbors$T` must be smaller than the number of subproblems. • `neighbors$delta.p`: Probability of sampling from the neighborhood when performing variation. Must be a scalar value between 0 and 1.
`v.matrix`	matrix of vectors to be used for defining the neighborhoods.
`iter`	iteration counter of the MOEA/D

Details

This routine calculates the neighborhood relations for the MOEA/D.

Warning: this routine may access (but not directly modify) variables from the calling environment.

Value

List containing the matrix of selection probabilities (`P`) and the matrix of neighborhoods (`B`).

`evaluate_population` *Evaluate population*

Description

Evaluate a population matrix on the objective functions for the MOEADr package

Usage

```
evaluate_population(X, problem, nfe)
```

Arguments

`X`	Population matrix of the MOEA/D (each row is a candidate solution).
`problem`	list of named problem parameters. See Section `Problem Description` of the `moead()` documentation for details.
`nfe`	counter of function evaluations from the `moead()` routine.

Details

This routine evaluates a population matrix for the MOEA/D. Each row of the matrix is considered as a candidate solution. This routine expects the candidate solutions to be standardized, i.e., that the variable limits given in `problem$xmin` and `problem$xmax` are mapped to `0` and `1`, respectively.

Value

List object containing the matrix of objective function values, a list object containing information about the constraint violations (a matrix of constraint values `Cmatrix`, a matrix of constraint violations `Vmatrix`, and a vector of total violations `v`), and the updated counter `nfe`.

Examples

```
ex.problem <- list(name = "example_problem",
                   xmin = rep(-1, 5),
                   xmax = rep(1, 5),
                   m    = 2)
X <- create_population(20, ex.problem)
Y <- evaluate_population(X, ex.problem, nfe = 0)
```

`example_problem` *Example problem*

Description

Example problem - minimization of shifted sphere and rastrigin functions.

Usage

```
example_problem(X)
```

Arguments

`X`	population matrix (see `moead()` for details)

Value

Matrix of objective function values

`find_nondominated_points` *Find non-dominated points*

Description

Non-dominated point finding for **minimization** problems

Usage

```
find_nondominated_points(Y)
```

Arguments

`Y`	row matrix of points in the space of objectives.

Details

Non-dominated point finding, based on portions of function *fastNonDominatedSorting* from package NSGA2R (`https://CRAN.R-project.org/package=nsga2R`)

Value

logical vector of length `nrow(Y)` indicating the nondominated points as `TRUE`.

Examples

```
Y  <- matrix(runif(200), ncol = 2)
nd <- find_nondominated_points(Y)
plot(Y[, 1], Y[, 2], type = "p", pch = 20, las = 1)
points(Y[nd, 1], Y[nd, 2], type = "p", pch = 16, col = 2, cex = 1.5)
```

`generate_weights` *Calculate weight vectors*

Description

Calculates weight vectors for the MOEADr package

Usage

```
generate_weights(decomp, m, ...)
```

Arguments

`decomp`	List containing the decomposition method parameters. See `moead()` for details.
`m`	Number of objectives ($m \geq 2$)
`...`	other parameters (included for compatibility with generic call)

Details

This routine calculates the weight vectors for the MOEA/D. The list of available methods for generating the weights, as well as information about their specific parameters, can be generated using `get_decomposition_methods()`.

Value

Weight matrix `W`

Examples

```
decomp <- list(name = "sld", H = 99)
W <- generate_weights(decomp, m = 2)
```

`get_constraint_methods` *Print available constraint methods*

Description

Prints the constraint handling methods available in the MOEADr package

Usage

```
get_constraint_methods()
```

Details

This routine prints the names of the constraint handling methods available in the MOEADr package, to be used as the `constraint$name` parameter in the `moead(...)` call. Instructions for obtaining more info on each operator are also returned.

Value

Formatted data frame containing reference name (for `constraint$name`) and instructions for More Info about each method.

Examples

```
get_constraint_methods()
```

`get_decomposition_methods` *Print available decomposition methods*

Description

Prints the decomposition methods available in the MOEADr package

Usage

```
get_decomposition_methods()
```

Details

This routine prints the names of the decomposition methods available in the MOEADr package, to be used as the `decomp$name` parameter in the `moead(...)` call. Instructions for obtaining more info on each operator are also returned.

Value

Formatted data frame containing reference name (for `decomp$name`) and instructions for More Info about each method.

Examples

```
get_decomposition_methods()
```

`get_localsearch_methods`
Print available local search methods

Description

Prints the local search methods available in the MOEADr package

Usage

```
get_localsearch_methods()
```

Details

This routine prints the names of the local search methods available in the MOEADr package, to be used as the `aggfun$name` parameter in the `moead(...)` call. Instructions for obtaining more info on each operator are also returned.

Value

Formatted data frame containing reference name (for `variation$localsearch$type`) and instructions for More Info about each method.

Examples

```
get_localsearch_methods()
```

`get_scalarization_methods` *Print available scalarization methods*

Description

Prints the scalarization methods available in the MOEADr package

Usage

```
get_scalarization_methods()
```

Details

This routine prints the names of the scalarization methods available in the MOEADr package, to be used as the `aggfun$name` parameter in the `moead(...)` call. Instructions for obtaining more info on each operator are also returned.

Value

Formatted data frame containing reference name (for `aggfun$name`) and instructions for More Info about each method.

Examples

```
get_scalarization_methods()
```

`get_stop_criteria` *Print available stop criteria*

Description

Prints the stop criteria available in the MOEADr package

Usage

```
get_stop_criteria()
```

Details

This routine prints the names of the stop criteria available in the MOEADr package, to be used as the `stopcrit[[i]]$name` parameter in the `moead(...)` call. Instructions for obtaining more info on each criterion are also returned.

Value

Formatted data frame containing reference name (for `stopcrit[[i]]$name`) and instructions for More Info about each criterion.

Examples

```
get_stop_criteria()
```

`get_update_methods` *Print available update methods*

Description

Prints the update methods available in the MOEADr package

Usage

```
get_update_methods()
```

Details

This routine prints the names of the update methods available in the MOEADr package, to be used as the `update$name` parameter in the `moead(...)` call. Instructions for obtaining more info on each operator are also returned.

Value

Formatted data frame containing reference name (for `update$name`) and instructions for More Info about each method.

Examples

```
get_update_methods()
```

get_variation_operators — *Print available variation operators*

Description

Prints the variation operators available in the MOEADr package

Usage

```
get_variation_operators()
```

Details

This routine prints the names of the variation operators available in the MOEADr package, to be used as the `variation$name` parameter in the `moead(...)` call. Instructions for obtaining more info on each operator are also returned.

Value

Formatted data frame containing reference name (for `variation$name`) and instructions for More Info about each operator.

Examples

```
get_variation_operators()
```

ls_dvls — *Differential vector-based local search*

Description

Differential vector-based local search (DVLS) implementation for the MOEA/D

Usage

```
ls_dvls(Xt, Yt, Vt, B, W, which.x, trunc.x, problem, scaling, aggfun,
  constraint, ...)
```

Arguments

`Xt`	Matrix of incumbent solutions
`Yt`	Matrix of objective function values for Xt
`Vt`	List object containing information about the constraint violations of the *incumbent solutions*, generated by `evaluate_population()`
`B`	Neighborhood matrix, generated by `define_neighborhood()`.
`W`	matrix of weights (generated by `generate_weights()`).
`which.x`	logical vector indicating which subproblems should undergo local search
`trunc.x`	logical flag indicating whether candidate solutions generated by local search should be truncated to the variable limits of the problem.
`problem`	list of named problem parameters. See Section `Problem Description` of the `moead()` documentation for details.
`scaling`	list containing the scaling parameters (see `moead()` for details).
`aggfun`	List containing the aggregation function parameters. See Section `Scalar Aggregation Functions` of the `moead()` documentation for details.
`constraint`	list containing the parameters defining the constraint handling method. See Section `Constraint Handling` of the `moead()` documentation for details.
`...`	other parameters (included for compatibility with generic call)

Details

This routine implements the differential vector-based local search for the MOEADr package. Check the references for details.

This routine is intended to be used internally by `variation_localsearch()`, and should not be called directly by the user.

Value

List object with fields `X` (matrix containing the modified points, with points that did not undergo local search indicated as NA) and `nfe` (integer value informing how many additional function evaluations were performed).

References

B. Chen, W. Zeng, Y. Lin, D. Zhang, "A new local search-based multiobjective optimization algorithm", IEEE Trans. Evolutionary Computation 19(1):50-73, 2015.

`ls_tpqa` *Three-point quadratic approximation local search*

Description

Three-point quadratic approximation (TPQA) local search implementation for the MOEA/D

Usage

```
ls_tpqa(Xt, Yt, W, B, Vt, scaling, aggfun, constraint, epsilon = 1e-06,
  which.x, ...)
```

Arguments

`Xt`	Matrix of incumbent solutions
`Yt`	Matrix of objective function values for Xt
`W`	matrix of weights (generated by `generate_weights()`).
`B`	Neighborhood matrix, generated by `define_neighborhood()`.
`Vt`	List object containing information about the constraint violations of the *incumbent solutions*, generated by `evaluate_population()`
`scaling`	list containing the scaling parameters (see `moead()` for details).
`aggfun`	List containing the aggregation function parameters. See Section `Scalar Aggregation Functions` of the `moead()` documentation for details.
`constraint`	list containing the parameters defining the constraint handling method. See Section `Constraint Handling` of the `moead()` documentation for details.
`epsilon`	threshold for using the quadratic approximation value
`which.x`	logical vector indicating which subproblems should undergo local search
`...`	other parameters (included for compatibility with generic call)

Details

This routine implements the 3-point quadratic approximation local search for the MOEADr package. Check the references for details.

This routine is intended to be used internally by `variation_localsearch()`, and should not be called directly by the user.

Value

Matrix `X`' containing the modified population

References

Y. Tan, Y. Jiao, H. Li, X. Wang, "A modification to MOEA/D-DE for multiobjective optimization problems with complicated Pareto sets", Information Sciences 213(1):14-38, 2012.

Y.-C. Jiao, C. Dang, Y. Leung, Y. Hao, "A modification to the new version of the prices algorithm for continuous global optimization problems", J. Global Optimization 36(4):609-626, 2006.

`make_vectorized_smoof` *Make vectorized smoof function*

Description

Make a vectorized version of test functions available in package "smoof".

Usage

```
make_vectorized_smoof(prob.name, ...)
```

Arguments

`prob.name`	name of the problem to build
`...`	other parameters passed to each specific function

Details

This routine builds MOEADr-compliant versions of the classic multiobjective test functions available in package smoof. The most commonly used ones are:

- `prob.name = ZDT1, ... , ZDT6`, in which case the function requires additional parameter `dimensions` (positive integer)
- `prob.name = DTLZ1, ..., DTLZ7`, in which case the function requires additional parameters `dimensions` (positive integer), `n.objectives` (= 2 or 3) and, for DTLZ4, `alpha` (positive integer, defaults to 100).
- `prob.name = UF`, in which case the function requires additional parameters `dimensions` (positive integer) and `id` (= 1, ..., 10).

Examples

```
## Not run:
  library(smoof)
  DTLZ2 <- make_vectorized_smoof(prob.name    = "DTLZ2",
                                 dimensions   = 10,
                                 n.objectives = 2)
  DTLZ2(X = matrix(runif(100), ncol = 10))

## End(Not run)
```

moead *MOEA/D*

Description

MOEA/D implementation in R

Usage

```
moead(preset = NULL, problem = NULL, decomp = NULL, aggfun = NULL,
  neighbors = NULL, variation = NULL, update = NULL, constraint = NULL,
  scaling = NULL, stopcrit = NULL, showpars = NULL, seed = NULL, ...)
```

Arguments

`preset`	List object containing preset values for one or more of the other parameters of the `moead` function. Values provided in the `preset` list will override any other value provided. Presets should be generated by the `preset_moead()` function.
`problem`	List containing the problem parameters. See `Problem Description` for details.
`decomp`	List containing the decomposition method parameters See `Decomposition methods` for details.
`aggfun`	List containing the aggregation function parameters See `Scalarization methods` for details.
`neighbors`	List containing the decomposition method parameters See `Neighborhood strategies` for details.
`variation`	List containing the variation operator parameters See `Variation operators` for details.
`update`	List containing the population update parameters See `Update strategies` for details.
`constraint`	List containing the constraint handing parameters See `Constraint operators` for details.
`scaling`	List containing the objective scaling parameters See `Objective scaling` for details.
`stopcrit`	list containing the stop criteria parameters. See `Stop criteria` for details.
`showpars`	list containing the echoing behavior parameters. See `print_progress()` for details.
`seed`	seed for the pseudorandom number generator. Defaults to NULL, in which case `as.integer(Sys.time())` is used for the definition.
`...`	Other parameters (useful for development and debugging, not necessary in regular use)

Details

Component-wise implementation of the Multiobjective Evolutionary Algorithm based on decomposition - MOEA/D.

Value

List object of class *moead* containing:

- information on the final population (`X`), its objective values (`Y`) and constraint information list (`V`) (see `evaluate_population()` for details);
- Archive population list containing its corresponding `X`, `Y` and `V` fields (only if `update$UseArchive = TRUE`).
- Estimates of the *ideal* and *nadir* points, calculated for the final population;
- Number of function evaluations, iterations, and total execution time;
- Random seed employed in the run, for reproducibility

Problem Description

The `problem` parameter consists of a list with all necessary definitions for the multiobjective optimization problem to be solved. `problem` must contain at least the following fields:

- `problem$name`: name of the problem instance function, that is, a routine that calculates $\mathbf{Y} = \mathbf{f}(\mathbf{X})$;
- `problem$xmin`: vector of lower bounds of each variable
- `problem$xmax`: vector of upper bounds of each variable
- `problem$m`: integer indicating the number of objectives

Besides these fields, `problem` should contain any other relevant inputs for the routine listed in `$name`. `problem` may also contain the (optional) field `problem$constraints`, which is a list object containing information about the problem constraints. If present, this list must have the following fields:

- `problem$constraints$name` - (required) name of the function that calculates the constraint values (see below for details)
- `problem$constraints$epsilon` - (optional) a small non-negative value indicating the tolerance to be considered for equality constraints. Defaults to zero.

Besides these fields, `problem$constraint` should contain any other relevant inputs for the routine listed in `problem$constraint$name`.

Detailed instructions for defining the routines for calculating the objective and constraint functions are provided in the vignette *Defining Problems in the MOEADr Package*. Check that documentation for details.

Decomposition Methods

The `decomp` parameter is a list that defines the method to be used for the generation of the weight vectors. `decomp` must have at least the `$name` parameter. Currently available methods can be verified using `get_decomposition_methods()`. Check `generate_weights()` and the information provided by `get_decomposition_methods()` for more details.

Neighborhood Strategies

The `neighbors` parameter is a list that defines the method for defining the neighborhood relations among subproblems. `neighbors` must have at least three parameters:

- `neighbors$name`, name of the strategy used to define the neighborhoods. Currently available methods are: - `$name = "lambda"`: uses the distances between weight vectors. The calculation is performed only once for the entire run, since the weight vectors are assumed static. - `$name = "x"`: uses the distances between the incumbent solutions associated with each subproblem. In this case the calculation is performed at each iteration, since incumbent solutions may change.
- `neighbors$T`: defines the neighborhood size. This parameter must receive a value smaller than the number of subproblems defined for the MOEA/D.
- `neighbors$delta.p`: parameter that defines the probability of sampling from the neighborhood when performing variation.

Check `define_neighborhood()` for more details.

Variation Operators

The `variation` parameter consists of a list vector, in which each sublist defines a variation operator to be used as part of the variation block. Each sublist must have at least a field `$name`, containing the name of the `i`-th variation operator to be applied. Use `get_variation_operators()` to generate a list of available operators, and consult the vignette `Variation Stack in the MOEADr Package` for more details.

Scalar Aggregation Functions

The `aggfun` parameter is a list that defines the scalar aggregation function to be used. `aggfun` must have at least the `$name` parameter. Currently available methods can be verified using `get_scalarization_methods()`. Check `scalarize_values()` and the information provided by `get_scalarization_methods()` for more details.

Update Methods

The `update` parameter is a list that defines the population update strategy to be used. `update` must have at least the `$name` parameter. Currently available methods can be verified using `get_update_methods()`. Check `update_population()` and the information provided by `get_update_methods()` for more details.

Another (optional) field of the `update` parameter is `update$UseArchive`, which is a binary flag defining whether the algorithm should keep an external solution archive (`TRUE`) or not (`FALSE`). Since it adds to the computational burden and memory requirements of the algorithm, the use of an archive population is recommended only in the case of constrained problems with constraint handling method that can occasionally accept unfeasible solutions, leading to the potential loss of feasible efficient solutions for certain subproblems (e.g., `constraint_vbr()` with type = "sr" or "vt").

Constraint Handling Methods

The `constraint` parameter is a list that defines the constraint-handling technique to be used. `constraint` must have at least the `$name` parameter. Currently available methods can be verified using `get_constraint_methods()`. Check `update_population()` and the information provided by `get_constraint_methods()` for more details.

Objective Scaling

Objective scaling refers to the re-scaling of the objective values at each iteration, which is generally considered to prevent problems arising from differently-scaled objective functions. `scaling` is a list that must have at least the `$name` parameter. Currently available options are `$name = "none"`, which does not perform any scaling, and `$name = "simple"`, which performs a simple linear scaling of the objectives to the interval `[0, 1]`.

Stop Criteria

The `stopcrit` parameter consists of a list vector, in which each sublist defines a termination criterion to be used for the MOEA/D. Each sublist must have at least a field `$name`, containing the name of the `i`-th criterion to be verified. The iterative cycle of the MOEA/D is terminated whenever any criterion is met. Use `get_stop_criteria()` to generate a list of available criteria, and check the information provided by that function for more details.

Echoing Options

The `showpars` parameter is a list that defines the echoing options of the MOEA/D. `showpars` must contain two fields:

- `showpars$show.iters`, defining the type of echoing output. `$show.iters` can be set as `"none"`, `"numbers"`, or `"dots"`.
- `showpars$showevery`, defining the period of echoing (in iterations). `$showevery` must be a positive integer.

References

F. Campelo, L.S. Batista, C. Aranha: "The MOEADr Package - A Component-Based Framework for Multiobjective Evolutionary Algorithms Based on Decomposition". In preparation, 2017.

Examples

```
## Prepare a test problem composed of minimization of the (shifted)
## sphere and Rastrigin functions
sphere     <- function(x){sum((x + seq_along(x) * 0.1) ^ 2)}
rastringin <- function(x){
                x.shift <- x - seq_along(x) * 0.1
                sum((x.shift) ^ 2 - 10 * cos(2 * pi * x.shift) + 10)}
problem.sr <- function(X){
                t(apply(X, MARGIN = 1,
                FUN = function(X){c(sphere(X), rastringin(X))}))}
```

```
## Set the input parameters for the moead() routine
## This reproduces the Original MOEA/D of Zhang and Li (2007)
## (with a few changes in the computational budget, to make it run faster)
problem   <- list(name       = "problem.sr",
                  xmin       = rep(-1, 30),
                  xmax       = rep(1, 30),
                  m          = 2)
decomp    <- list(name       = "SLD", H = 49) # <-- H = 99 in the original
neighbors <- list(name       = "lambda",
                  T          = 20,
                  delta.p    = 1)
aggfun    <- list(name       = "wt")
variation <- list(list(name  = "sbx",
                       etax  = 20, pc = 1),
                  list(name  = "polymut",
                       etam  = 20, pm = 0.1),
                  list(name  = "truncate"))
update    <- list(name       = "standard", UseArchive = FALSE)
scaling   <- list(name       = "none")
constraint<- list(name       = "none")
stopcrit  <- list(list(name  = "maxiter",
                    maxiter  = 50))      # <-- maxiter = 200 in the original
showpars  <- list(show.iters = "dots",
                  showevery  = 10)
seed      <- 42

## Run MOEA/D
out1 <- moead(preset = NULL,
              problem, decomp, aggfun, neighbors, variation, update,
              constraint, scaling, stopcrit, showpars, seed)

## Examine the output:
summary(out1)

## Alternatively, the standard MOEA/D could also be set up using the
## preset_moead() function. The code below runs the original MOEA/D with
## exactly the same configurations as in Zhang and Li (2007).
## Not run:
  out2 <- moead(preset   = preset_moead("original"),
                problem  = problem,
                showpars = showpars,
                seed     = 42)

  ## Examine the output:
  summary(out2)
  plot(out2, suppress.pause = TRUE)

## End(Not run)

# Rerun with MOEA/D-DE configuration and AWT scalarization
out3 <- moead(preset   = preset_moead("moead.de"),
              problem  = problem,
              aggfun   = list(name = "awt"),
              stopcrit = list(list(name    = "maxiter",
                                   maxiter = 50)),
              seed    = seed)
plot(out3, suppress.pause = TRUE)
```

`order_neighborhood` *Order Neighborhood for MOEA/D*

Description

Calculates the ordering of competing solutions for each subproblem in the MOEA/D, based on their scalarized performance and violation values.

Usage

```
order_neighborhood(bigZ, B, V, Vt, constraint)
```

Arguments

`bigZ`	Matrix of scalarized performance values by neighborhood, generated by `scalarize_values()`
`B`	Neighborhood matrix, generated by `define_neighborhood()`.
`V`	List object containing information about the constraint violations of the *candidate solutions*, generated by `evaluate_population()`
`Vt`	List object containing information about the constraint violations of the *incumbent solutions*, generated by `evaluate_population()`
`constraint`	list containing the parameters defining the constraint handling method. See Section `Constraint Handling` of the `moead()` documentation for details.

Details

This routine receives a matrix of scalarized performance values (returned by `scalarize_values()`), a neighborhood matrix, and the list of violation values for the candidate and incumbent populations. It calculates the preference order of the candidates for each neighborhood based on the performance values and constraint handling method.

The list of available constraint handling methods can be generated using `get_constraint_methods()`.

Value

`[N x (T+1)]` matrix of preference indexes. Each row contains the T indexes of the candidate solutions in the neighborhood of a given subproblem, plus a value (column T+1) for the incumbent solution of that subproblem, in an order defined by the constraint handling method specified in `moead.env$constraint`.

perform_variation *Run variation operators*

Description

Sequentially apply variation operators for the MOEADr package

Usage

```
perform_variation(variation, X, iter, ...)
```

Arguments

`variation`	List vector containing the variation operators to be used. See `moead()` for details.
`X`	Population matrix of the MOEA/D (each row is a candidate solution).
`iter`	iterations counter of the `moead()` function.
`...`	other parameters to be passed down to the individual variation operators (see documentation of the specific `variation_`**xyz**`()` functions for details)

Details

This routine performs the variation block for the MOEA/D. The list of available variation operators can be generated using `get_variation_operators()`.

If the `localsearch` operator is included, it is executed whenever its conditions (period of occurrence or probability of occurrence) are verified. See `variation_localsearch()` for details.

Value

List object containing a modified population matrix `X`, a local search argument list `ls.arg`, and the number of function evaluations used by the variation operators, `var.nfe`.

plot.moead *plot.moead*

Description

S3 method for plotting *moead* objects (the output of `moead()`).

Usage

```
## S3 method for class 'moead'
plot(x, ..., useArchive = FALSE, feasible.only = TRUE,
  viol.threshold = 1e-06, nondominated.only = TRUE, plot.weights = FALSE,
  which.objectives = NULL, suppress.pause = FALSE, color.by.obj = 1)
```

Arguments

`x`	list object of class *moead* (generated by `moead()`)
`...`	other parameters to be passed down to specific plotting functions (currently unused)
`useArchive`	logical flag to use information from `x$Archive`. Only used if x$Archive is not `NULL`.
`feasible.only`	logical flag to use only feasible points in the plots.
`viol.threshold`	threshold of tolerated constraint violation, used to determine feasibility if `feasible.only == TRUE`.
`nondominated.only`	logical flag to use only nondominated points in the plots.
`plot.weights`	logical flag to plot the weight vectors for 2 and 3-objective problems.
`which.objectives`	integer vector of which objectives to plot. Defaults to `NULL` (use all objectives)
`suppress.pause`	logical flag to prevent pause messages from being show after every image. Defaults to `FALSE` (show pause messages)
`color.by.obj`	integer, determines which objective is used as the basis for coloring the parallel coordinates plot.

Examples

```
problem.1 <- list(name = "example_problem",
                  xmin = rep(-1,30),
                  xmax = rep(1,30),
                  m    = 2)
out <- moead(preset    = preset_moead("original2"),
             problem   = problem.1,
             stopcrit  = list(list(name = "maxiter",
                                   maxiter = 100)),
             showpars  = list(show.iters = "dots",
                              showevery  = 10))
plot(out, suppress.pause = TRUE)
```

`preset_moead` *preset_moead*

Description

Generate a preset configuration for moead()].

Usage

```
preset_moead(name = NULL)
```

Arguments

`name`	name of the preset to be generated. Use `preset_moead()` to obtain the list of available options.

Details

This function returns a list of configuration presets taken from the literature to be used with the `moead()` function in package `MOEADr`.

Use these configurations as a starting point. We strongly recommend that you play around with the particular configurations (see example).

Value

List object containing the preset, to be used as an input to `moead()`; or, if `name == NULL` (the default), returns a logical flag invisibly.

Examples

```
# Generate list of available presets
preset_moead(name = NULL)

## Not run:
  library(smoof) # < Install package smoof if needed
  ZDT1 <- make_vectorized_smoof(prob.name  = "ZDT1",
                                dimensions = 30)
                                problem   <- list(name       = "ZDT1",
                                                  xmin       = rep(0, 30),
                                                  xmax       = rep(1, 30),
                                                  m          = 2)

  # Get preset configuration for original MOEA/D
  configuration <- preset_moead("original")

  # Modify whatever you fancy:
  stopcrit <- list(list(name = "maxiter", maxiter = 50))
  showpars <- list(show.iters = "dots", showevery  = 10)
  seed     <- 42

  output <- moead(problem  = problem,
                  preset   = configuration,
                  showpars = showpars,
                  stopcrit = stopcrit,
                  seed     = seed)

## End(Not run)
```

print.moead *print.moead*

Description

S3 method for printing *moead* objects (the output of `moead()`).

Usage

```
## S3 method for class 'moead'
print(x, ...)
```

Arguments

`x`	list object of class *moead* (generated by `moead()`)
`...`	other parameters to be passed down to specific summary functions (currently unused)

Examples

```
problem.1 <- list(name = "example_problem",
                  xmin = rep(-1,30),
                  xmax = rep(1,30),
                  m    = 2)
out <- moead(preset    = preset_moead("original2"),
             problem   = problem.1,
             stopcrit  = list(list(name = "maxiter",
                                   maxiter = 100)),
             showpars  = list(show.iters = "dots",
                              showevery  = 10))
print(out)
```

print_progress *Print progress of MOEA/D*

Description

Echoes progress of MOEA/D to the terminal for the MOEADr package

Usage

```
print_progress(iter.times, showpars)
```

Arguments

`iter.times`	vector of iteration times of the `moead()` routine.
`showpars`	list object containing parameters that control the printed output of `moead()`. Parameter `showpars` can have the following key-value pairs: • `$show.iters`: type of output ("dots", "numbers", or "none"). If not present in `showpars`, it defaults to "numbers"; • `$showevery`: positive integer that determines how frequently the routine echoes something to the terminal. If not present in `showpars`, it defaults to `10`.

`scalarization_awt` *Adjusted Weighted Tchebycheff Scalarization*

Description

Perform Adjusted Weighted Tchebycheff Scalarization for the MOEADr package.

Usage

```
scalarization_awt(Y, W, minP, eps = 1e-16, ...)
```

Arguments

`Y`	matrix of objective function values
`W`	matrix of weights.
`minP`	numeric vector containing estimated ideal point
`eps`	tolerance value for avoiding divisions by zero.
`...`	other parameters (included for compatibility with generic call)

Details

This routine calculates the scalarized performance values for the MOEA/D using the Adjusted Weighted Tchebycheff method.

Value

Vector of scalarized performance values.

References

Y. Qi, X. Ma, F. Liu, L. Jiao, J. Sun, and J. Wu, “MOEA/D with adaptive weight adjustment,” Evolutionary Computation, vol. 22, no. 2, pp. 231–264, 2013.

R. Wang, T. Zhang, and B. Guo, “An enhanced MOEA/D using uniform directions and a pre-organization procedure,” in IEEE Congress on Evolutionary Computation, Cancun, Mexico, 2013, pp. 2390–2397.

Examples

```
W    <- generate_weights(decomp = list(name = "sld", H = 19), m = 2)
Y    <- matrix(runif(40), ncol = 2)
minP <- apply(Y, 2, min)
Z    <- scalarization_awt(Y, W, minP)
```

scalarization_ipbi *Inverted Penalty-based Boundary Intersection Scalarization*

Description

Perform inverted PBI Scalarization for the MOEADr package.

Usage

```
scalarization_ipbi(Y, W, maxP, aggfun, eps = 1e-16, ...)
```

Arguments

Y	matrix of objective function values
W	matrix of weights.
maxP	numeric vector containing estimated ideal point
aggfun	list containing parameters for the aggregation function. Must contain the non-negative numeric constant aggfun$theta.
eps	tolerance value for avoiding divisions by zero.
...	other parameters (included for compatibility with generic call)

Details

This routine calculates the scalarized performance values for the MOEA/D using the inverted PBI method.

Value

Vector of scalarized performance values.

References

H. Sato, "Inverted PBI in MOEA/D and its impact on the search performance on multi and many-objective optimization." Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation (GECCO), 2014.

H. Sato, "Analysis of inverted PBI and comparison with other scalarizing functions in decomposition based MOEAs." Journal of Heuristics 21(6):819-849, 2015

Examples

```
W      <- generate_weights(decomp = list(name = "sld", H = 19), m = 2)
Y      <- matrix(runif(40), ncol = 2)
minP   <- apply(Y, 2, min)
aggfun <- aggfun    <- list(name = "ipbi", theta = 5)
Z      <- scalarization_ipbi(Y, W, minP, aggfun)
```

`scalarization_pbi` *Penalty-based Boundary Intersection Scalarization*

Description

Perform PBI Scalarization for the MOEADr package.

Usage

```
scalarization_pbi(Y, W, minP, aggfun, eps = 1e-16, ...)
```

Arguments

`Y`	matrix of objective function values
`W`	matrix of weights.
`minP`	numeric vector containing estimated ideal point
`aggfun`	list containing parameters for the aggregation function. Must contain the non-negative numeric constant `aggfun$theta`.
`eps`	tolerance value for avoiding divisions by zero.
`...`	other parameters (included for compatibility with generic call)

Details

This routine calculates the scalarized performance values for the MOEA/D using the PBI method.

Value

Vector of scalarized performance values.

References

Q. Zhang and H. Li, "MOEA/D: A Multiobjective Evolutionary Algorithm

H. Li, Q. Zhang, "Multiobjective Optimization Problems With Complicated Pareto Sets, MOEA/D and NSGA-II", IEEE. Trans. Evol. Comp. 12(2):284-302, 2009.

Examples

```
W      <- generate_weights(decomp = list(name = "sld", H = 19), m = 2)
Y      <- matrix(runif(40), ncol = 2)
minP   <- apply(Y, 2, min)
aggfun <- aggfun    <- list(name = "pbi", theta = 5)
Z      <- scalarization_pbi(Y, W, minP, aggfun)
```

scalarization_ws *Weighted Sum Scalarization*

Description

Perform Weighted Sum Scalarization for the MOEADr package.

Usage

```
scalarization_ws(Y, W, minP, eps = 1e-16, ...)
```

Arguments

Y	matrix of objective function values
W	matrix of weights.
minP	numeric vector containing estimated ideal point
eps	tolerance value for avoiding divisions by zero.
...	other parameters (included for compatibility with generic call)

Details

This routine calculates the scalarized performance values for the MOEA/D using the Weighted Sum method.

Value

vector of scalarized performance values.

References

Q. Zhang and H. Li, "MOEA/D: A Multiobjective Evolutionary Algorithm

H. Li, Q. Zhang, "Multiobjective Optimization Problems With Complicated Pareto Sets, MOEA/D and NSGA-II", IEEE. Trans. Evol. Comp. 12(2):284-302, 2009.

Examples

```
W    <- generate_weights(decomp = list(name = "sld", H = 19), m = 2)
Y    <- matrix(runif(40), ncol = 2)
minP <- apply(Y, 2, min)
Z    <- scalarization_ws(Y, W, minP)
```

scalarization_wt *Weighted Tchebycheff Scalarization*

Description

Perform Weighted Tchebycheff Scalarization for the MOEADr package.

Usage

```
scalarization_wt(Y, W, minP, eps = 1e-16, ...)
```

Arguments

`Y`	matrix of objective function values
`W`	matrix of weights.
`minP`	numeric vector containing estimated ideal point
`eps`	tolerance value for avoiding divisions by zero.
`...`	other parameters (included for compatibility with generic call)

Details

This routine calculates the scalarized performance values for the MOEA/D using the Weighted Tchebycheff method.

Value

Vector of scalarized performance values.

References

Q. Zhang and H. Li, "MOEA/D: A Multiobjective Evolutionary Algorithm

H. Li, Q. Zhang, "Multiobjective Optimization Problems With Complicated Pareto Sets, MOEA/D and NSGA-II", IEEE. Trans. Evol. Comp. 12(2):284-302, 2009.

Examples

```
W    <- generate_weights(decomp = list(name = "sld", H = 19), m = 2)
Y    <- matrix(runif(40), ncol = 2)
minP <- apply(Y, 2, min)
Z    <- scalarization_wt(Y, W, minP)
```

`scalarize_values` *Scalarize values for MOEA/D*

Description

Perform scalarization for the MOEADr package.

Usage

```
scalarize_values(normYs, W, B, aggfun)
```

Arguments

`normYs`	List generated by `scale_objectives()`, containing two matrices of scaled objective values (`normYs$Y` and `normYs$Yt`) and two vectors, containing the current estimates of the ideal (`normYs$minP`) and nadir (`normYs$maxP`) points. See `scale_objectives()` for details.
`W`	matrix of weights, generated by `generate_weights()`.
`B`	neighborhood matrix, generated by `define_neighborhood()`.
`aggfun`	List containing the aggregation function parameters. See Section `Scalar Aggregation Functions` of the `moead()` documentation for details.

Details

This routine calculates the scalarized performance values for the MOEA/D.

The list of available scalarization methods can be generated using `get_scalarization_methods()`

Value

`[ (T+1) x N ]` matrix of scalarized performance values. Each column contains the T scalarized performances of the candidate solutions in the neighborhood of a given subproblem, plus the scalarized performance value for the incumbent solution for that subproblem.

`scale_objectives` *Scaling of the objective function values*

Description

Performs scaling of the objective function values for the MOEADr package

Usage

```
scale_objectives(Y, Yt, scaling, eps = 1e-16, ...)
```

Arguments

`Y`	matrix of objective function values for the incumbent solutions
`Yt`	matrix of objective function values for the candidate solutions
`scaling`	list containing the scaling parameters (see `moead()` for details).
`eps`	tolerance value for avoiding divisions by zero.
`...`	other parameters (included for compatibility with generic call)

Details

This routine scales the matrices of objective function values for the current (`Yt`) and candidate (`Y`) solutions. The following methods are currently available:

- `scaling$name = "none"`: no scaling
- `scaling$name = "simple"`: simple linear scaling between estimated ideal and nadir points, calculated from the available points in `Y` and `Yt` at each iteration.

Value

List object containing scaled objective function value matrices `Y` and `Yt`, as well as estimates of the "ideal" point `minP`` and "nadir" pointmaxP`‘.

`stop_maxeval` *Stop criterion: maximum number of evaluations*

Description

Verifies stop criterion "maximum number of evaluations" for the MOEADr package. For internal use only, not to be called directly by the user.

Usage

```
stop_maxeval(stopcrit, nfe, ...)
```

Arguments

`stopcrit`	list containing the parameters defining the stop handling method. See Section `Constraint Handling` of the `moead()` documentation for details.
`nfe`	evaluations counter of `moead()`.
`...`	other parameters (included for compatibility with generic call)

Details

When this stop criterion is used, one element of the `stopcrit` parameter (see `moead()`) must have the following structure:

- `stopcrit$name = "maxeval"`
- `stopcrit$maxeval`, containing a positive integer representing the desired maximum number of evaluations.

Value

boolean value: `TRUE` if this criterion has been met, `FALSE` otherwise.

`stop_maxiter` *Stop criterion: maximum number of iterations*

Description

Verifies stop criterion "maximum number of iterations" for the MOEADr package. For internal use only, not to be called directly by the user.

Usage

```
stop_maxiter(stopcrit, iter, ...)
```

Arguments

`stopcrit`	list containing the parameters defining the stop handling method. See Section `Constraint Handling` of the `moead()` documentation for details.
`iter`	iterations counter of `moead()`.
`...`	other parameters (included for compatibility with generic call)

Details

When this stop criterion is used, one element of the `stopcrit` parameter (see `moead()`) must have the following structure:

- `stopcrit$name = "maxiter"`
- `stopcrit$maxiter`, containing a positive integer representing the desired maximum number of iterations.

Value

boolean value: `TRUE` if this criterion has been met, `FALSE` otherwise.

`stop_maxtime` *Stop criterion: maximum runtime*

Description

Verifies stop criterion "run time limit" for the MOEADr package. For internal use only, not to be called directly by the user.

Usage

```
stop_maxtime(stopcrit, iter.times, ...)
```

Arguments

`stopcrit`	list containing the parameters defining the stop handling method. See Section `Constraint Handling` of the `moead()` documentation for details.
`iter.times`	vector containing the times spent by each iteration of the moead() routine, up to the current one.
`...`	other parameters (included for compatibility with generic call)

Details

When this stop criterion is used, one element of the `stopcrit` parameter (see `moead()`) must have the following structure:

- `stopcrit$name = "maxtime"`
- `stopcrit$maxtime`, containing a positive integer representing the desired time limit (in seconds).

Value

boolean value: `TRUE` if this criterion has been met, `FALSE` otherwise.

Warning

This function uses Sys.time() for verifying the total run time, i.e., it counts wall-clock time, not CPU time.

`summary.moead` *summary.moead*

Description

S3 method for summarizing *moead* objects (the output of `moead()`).

Usage

```
## S3 method for class 'moead'
summary(object, ..., useArchive = FALSE,
  viol.threshold = 1e-06, ndigits = 3, ref.point = NULL,
  ref.front = NULL)
```

Arguments

`object`	list object of class *moead* (generated by `moead()`)
`...`	other parameters to be passed down to specific summary functions (currently unused)
`useArchive`	logical flag to use information from `object$Archive`. Only used if object$Archive is not `NULL`.
`viol.threshold`	threshold of tolerated constraint violation, used to determine feasibility of points in `object`.
`ndigits`	number of decimal places to use for the ideal and nadir estimates
`ref.point`	reference point for calculating the dominated hypervolume (only if package `emoa` is available). If `NULL` the estimated nadir point is used instead.
`ref.front`	`Np x Nobj` matrix containing a sample of the true Pareto-optimal front, for calculating IGD.

Examples

```
problem.1 <- list(name = "example_problem",
                  xmin = rep(-1,30),
                  xmax = rep(1,30),
                  m    = 2)
out <- moead(preset    = preset_moead("original2"),
             problem   = problem.1,
             stopcrit  = list(list(name = "maxiter",
                                   maxiter = 100)),
             showpars  = list(show.iters = "dots",
                              showevery  = 10))
summary(out)
```

`unitary_constraints` *Unitary constraints routine*

Description

Calculates the constraint values and violations when only unitary constraints (i.e., the sum of all variables equals one) are present.

Usage

```
unitary_constraints(X, epsilon = 0, ...)
```

Arguments

`X`	Population matrix of the MOEA/D (each row is a candidate solution). If `NULL` the function searches for `X` in the calling environment.
`epsilon`	small non-negative value indicating the tolerance to be considered for the equality constraint. Defaults to zero.
`...`	other parameters (unused, included for compatibility with generic call)

Details

This routine calculates the constraint values and violations for a population matrix in the MOEA/D. Each row of the matrix is considered as a candidate solution. This routine expects the candidate solutions to be standardized, i.e., that the variable limits given in `problem$xmin` and `problem$xmax` are mapped to `0` and `1`, respectively.

Value

List objective containing a matrix of constraint values `Cmatrix`, a matrix of individual constraint violations `Vmatrix`, and a vector of total constraint violations `v`.

`update_population` *Update population*

Description

Selection and population update procedures for the MOEA/D

Usage

```
update_population(update, ...)
```

Arguments

`update`	List containing the population update parameters. See Section `Update Strategies` of the `moead()` documentation for details.
`...`	other parameters to be passed down to the specific `updt_`**xyz**`()` routines.

Details

This update routine is intended to be used internally by the main `moead()` function, and should not be called directly by the user. The list of available update methods can be generated using `get_update_methods()`.

Value

List object containing the updated values of the population matrix `X`, objective function matrix `Y`, and constraint values list `V`, as well as an updated Archive list containing its corresponding components `X`, `Y` and `V`.

`updt_best` *Best Neighborhood Replacement Update for MOEA/D*

Description

Population update using the best neighborhood replacement method for the MOEADr package.

Usage

```
updt_best(update, X, Xt, Y, Yt, V, Vt, normYs, W, BP, constraint, aggfun, ...)
```

Arguments

`update`	List containing the population update parameters. See Section `Update Strategies` of the `moead()` documentation for details. `update` must have the following key-value pairs: • `update$Tr`: positive integer, neighborhood size for the update operation • `update$nr`: positive integer, maximum number of copies of a given candidate solution.
`X`	Matrix of candidate solutions
`Xt`	Matrix of incumbent solutions
`Y`	Matrix of objective function values of `X`
`Yt`	Matrix of objective function values of `Xt`
`V`	List object containing information about the constraint violations of the candidate solutions, generated by `evaluate_population()`
`Vt`	List object containing information about the constraint violations of the incumbent solutions, generated by `evaluate_population()`

`normYs`	List generated by `scale_objectives()`, containing two matrices of scaled objective values (`normYs$Y` and `normYs$Yt`) and two vectors, containing the current estimates of the ideal (`normYs$minP`) and nadir (`normYs$maxP`) points. See `scale_objectives()` for details.
`W`	matrix of weights, generated by `generate_weights()`.
`BP`	Neighborhood list, generated by `define_neighborhood()`.
`constraint`	list containing the parameters defining the constraint handling method. See Section `Constraint Handling` of the `moead()` documentation for details.
`aggfun`	List containing the aggregation function parameters. See Section `Scalar Aggregation Functions` of the `moead()` documentation for details.
`...`	other parameters (included for compatibility with generic call)

Details

The Best Neighborhood Replacement method consists of three steps:

- For each subproblem `i`, the best candidate solution `x_j` from the entire population is determined.
- The neighborhood of subproblem `i` is replaced by the neighborhood of subproblem j. The size of this neighborhood is given by a parameter `Tr`.
- The Restricted replacement (see `updt_restricted()`) is then applied using this new neighborhood.

This update routine is intended to be used internally by the main `moead()` function, and should not be called directly by the user.

Value

List object containing the update population matrix (`X`), and its corresponding matrix of objective function values (`Y`) and constraint value list (`V`).

`updt_restricted` *Restricted Neighborhood Replacement Update for MOEA/D*

Description

Population update using the restricted neighborhood replacement method for the MOEADr package.

Usage

```
updt_restricted(update, X, Xt, Y, Yt, V, Vt, sel.indx, B, ...)
```

Arguments

`update`	List containing the population update parameters. See Section `Update Strategies` of the `moead()` documentation for details. `update` must contain a field `update$nr`, a positive integer that determines the maximum number of copies of each candidate solution.
`X`	Matrix of candidate solutions
`Xt`	Matrix of incumbent solutions
`Y`	Matrix of objective function values of `X`
`Yt`	Matrix of objective function values of `Xt`
`V`	List object containing information about the constraint violations of the candidate solutions, generated by `evaluate_population()`
`Vt`	List object containing information about the constraint violations of the incumbent solutions, generated by `evaluate_population()`
`sel.indx`	matrix of selection indices, generated by `order_neighborhood()`
`B`	Neighborhood matrix, generated by `define_neighborhood()`.
`...`	other parameters (included for compatibility with generic call)

Details

The restricted neighborhood replacement method behaves like the "standard" replacement method, except that each individual can only be selected up to `nr` times. After this limit has been reached, the next best individual in the same neighborhood is selected.

This update routine is intended to be used internally by the main `moead()` function, and should not be called directly by the user.

Value

List object containing the update population matrix (`X`), and its corresponding matrix of objective function values (`Y`) and constraint value list (`V`).

`updt_standard` *Standard Neighborhood Replacement Update for MOEA/D*

Description

Population update using the standard neighborhood replacement method for the MOEADr package.

Usage

```
updt_standard(X, Xt, Y, Yt, V, Vt, sel.indx, B, ...)
```

Arguments

`X`	Matrix of candidate solutions
`Xt`	Matrix of incumbent solutions
`Y`	Matrix of objective function values of `X`
`Yt`	Matrix of objective function values of `Xt`
`V`	List object containing information about the constraint violations of the candidate solutions, generated by `evaluate_population()`
`Vt`	List object containing information about the constraint violations of the incumbent solutions, generated by `evaluate_population()`
`sel.indx`	matrix of selection indices, generated by `order_neighborhood()`
`B`	Neighborhood matrix, generated by `define_neighborhood()`.
`...`	other parameters (included for compatibility with generic call)

Details

This routine executes the standard neighborhood replacement operation to update the population matrix of the MOEA/D. This update routine is intended to be used internally by the main `moead()` function, and should not be called directly by the user.

Value

List object containing the update population matrix (`X`), and its corresponding matrix of objective function values (`Y`) and constraint value list (`V`).

`variation_binrec` *Binomial Recombination*

Description

Binomial recombination implementation for the MOEA/D.

Usage

```
variation_binrec(X, Xt, rho, ...)
```

Arguments

`X`	Population matrix
`Xt`	Original population matrix
`rho`	mutation probability
`...`	other parameters (included for compatibility with generic call)

Details

This variation operator only works if at least one other variation operator is performed prior to its execution, otherwise it becomes an identity operator (returns an unchanged matrix X).

Value

Matrix `X`' containing the recombined population

References

K. Price, R.M. Storn, J.A. Lampinen, "Differential Evolution: A Practical Approach to Global Optimization", Springer 2005

`variation_diffmut` *Differential Mutation*

Description

Differential Mutation implementation for the MOEA/D

Usage

```
variation_diffmut(X, P, B, Phi = NULL, basis = "rand", ...)
```

Arguments

`X`	Population matrix
`P`	Matrix of selection probabilities (generated by `define_neighborhood()`)
`B`	Matrix of neighborhoods (generated by `define_neighborhood()`)
`Phi`	Mutation parameter. Either a scalar numeric constant, or NULL for randomly chosen between `0` and `1` (independently sampled for each operation).
`basis`	how to select the basis vector. Currently supported methods are: • `basis = "rand"`, for using a randomly sampled vector from the population; • `basis = "mean"`, for using the mean point of the neighborhood; • `basis = "wgi"`, for using the the weighted mean point of the neighborhood.
`...`	other parameters to be passed down to specific options of basis vector generation (e.g., `Y`, `Yt`, `W`, `scaling` and `aggfun`, required when `basis = "wgi"`).

Details

This function generalizes many variations of the Differential Mutation operator with general form:

`u = x_basis + Phi(x_a - x_b)`

Where u is the new candidate vector, `Phi != 0` is a real number, and `x_basis`, `x_a` and `x_b` are distinct vectors.

This routine is intended to be used internally by `perform_variation()`, and should not be called directly by the user.

Value

Matrix `X`' containing the mutated population

References

K. Price, R.M. Storn, J.A. Lampinen, "Differential Evolution: A Practical Approach to Global Optimization", Springer 2005

D. V. Arnold, “Weighted multirecombination evolution strategies,” Theoretical Computer Science 361(1):18–37, 2006.

`variation_localsearch` *Local search Operators*

Description

Local search operators for the MOEA/D

Usage

```
variation_localsearch(...)
```

Arguments

`...`	arguments to be passed down to the specific `ls_`**xyz**`()` functions. A list of available local search methods can be generated by `get_localsearch_methods()`. Consult the documentation of the specific functions for details.

Details

This routine calls the local search operator for the MOEADr package, as part of the call to `perform_variation()`. This operator requires its entry in the variation stack (see Section `Variation Operators` of `moead()`) to contain the following fields:

- `name = "localsearch"`
- `type` (see `get_localsearch_methods()` for details)
- `gamma.ls` (optional): probability of application of local search to a given subproblem at any given iteration (numeric between 0 and 1)
- `tau.ls` (optional): period of application of local search to each subproblem (positive integer)
- `trunc.x` (optional): logical flag for truncating the results of the local search operator to the limits defined by `problem$xmin`, `problem$xmax` (logical). Defaults to `TRUE`.

Whenever local search is triggered for a given subproblem, it cancels all other variation operators *for that subproblem* and is executed directly on the incumbent solution.

This routine is intended to be used internally by `perform_variation()`, and should not be called directly by the user.

Value

Either a matrix `Xls` containing the modified points (points that did not undergo local search are indicated as NA in this output matrix), or a list object containing the `Xls` matrix and an integer `nfe`, informing how many additional function evaluations were performed by the local search operator. The specific output is defined by the `ls_`**xyz**`()` method used.

`variation_none` *Identity operator*

Description

Identity operator (no variation performed)

Usage

```
variation_none(X, ...)
```

Arguments

`X`	Population matrix
`...`	other parameters (included for compatibility with generic call)

Details

Performs the identity operator (no variation). This routine is included to simplify the use of automated tuning / design tools such as Iterated Racing.

Value

Input matrix `X`

`variation_polymut` *Polynomial mutation*

Description

Polynomial mutation implementation for the MOEA/D

Usage

```
variation_polymut(X, etam, pm, eps = 1e-06, ...)
```

Arguments

`X`	Population matrix
`etam`	mutation constant
`pm`	variable-wise probability of mutation
`eps`	small constant used to prevent divisions by zero
`...`	other parameters (included for compatibility with generic call)

Details

This R implementation of the Polynomial Mutation reproduces the C code implementation available in the R package **emoa** 0.5-0, by Olaf Mersmann. The differences between the present version and the original one are:

- The operator is performed on the variables scaled to the `[0, 1]` interval, which simplifies the calculations.
- Calculations are vectorized over variables, which also simplifies the implementation.

Value

Matrix `X`' containing the mutated population

References

K. Deb and S. Agrawal (1999). A Niched-Penalty Approach for Constraint Handling in Genetic Algorithms. In: Artificial Neural Nets and Genetic Algorithms, pp. 235-243, Springer.

Olaf Mersmann (2012). emoa: Evolutionary Multiobjective Optimization Algorithms. R package version 0.5-0.
http://CRAN.R-project.org/package=emoa

`variation_sbx` *Simulated binary crossover*

Description

SBX implementation for the MOEA/D

Usage

```
variation_sbx(X, P, etax, pc = 1, eps = 1e-06, ...)
```

Arguments

`X`	Population matrix
`P`	Matrix of probabilities of selection for variation (created by `define_neighborhood()`).
`etax`	spread constant
`pc`	variable-wise probability of recombination
`eps`	smallest difference considered for recombination
`...`	other parameters (included for compatibility with generic call)

Details

This R implementation of the Simulated Binary Crossover reproduces the C code implementation available in the R package **emoa** 0.5-0, by Olaf Mersmann. The differences between the present version and the original one are:

- The operator is performed on the variables scaled to the `[0, 1]` interval, which simplifies the calculations.
- Calculations are vectorized over variables, which also simplifies the implementation.

Value

Matrix `X`' containing the recombined population

References

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

Olaf Mersmann (2012). emoa: Evolutionary Multiobjective Optimization Algorithms. R package version 0.5-0.
http://CRAN.R-project.org/package=emoa

`variation_truncate` *Truncate*

Description

Truncation variation operator

Usage

```
variation_truncate(X, ...)
```

Arguments

`X`	Population matrix
`...`	other parameters (included for compatibility with generic call)

Details

Truncate the solution matrix `X` to the `[0, 1]` interval.

Value

Truncated matrix `X`’.

Index