Package ‘miesmuschel’

November 16, 2022

Title Mixed Integer Evolution Strategies

Description Evolutionary black box optimization algorithms building on the 'bbotk' package. 'miesmuschel' offers both ready-to-use optimization algorithms, as well as their fundamental building blocks that can be used to manually construct specialized optimization loops. The Mixed Integer Evolution Strategies as described by Li et al. (2013) <doi:10.1162/EVCO_a_00059> can be implemented, as well as the multi-objective optimization algorithms NSGA-II by Deb, Pratap, Agarwal, and Meyarivan (2002) <doi:10.1109/4235.996017>.

URL https://github.com/mlr-org/miesmuschel

BugReports https://github.com/mlr-org/miesmuschel/issues

License MIT + file LICENSE

Encoding UTF-8

ByteCompile yes

Version 0.0.2

Depends paradox (>= 0.7.1)

Imports mlr3misc (>= 0.5.0), checkmate (>= 1.9.0), R6, bbotk (>= 0.3.0.900), data.table, matrixStats, lgr

Suggests tinytest, mlr3tuning, mlr3, mlr3learners, ranger, xgboost, rpart

RoxygenNote 7.2.2

Collate 'dictionaries.R' 'MiesOperator.R' 'Filtor.R' 'FiltorMaybe.R' 'FiltorNull.R' 'FiltorProxy.R' 'FiltorSurrogate.R' 'FiltorSurrogateProgressive.R' 'FiltorSurrogateTournament.R' 'Mutator.R' 'MutatorCmpMaybe.R' 'MutatorDiscreteUniform.R' 'MutatorErase.R' 'MutatorGauss.R' 'MutatorMaybe.R' 'MutatorNull.R' 'MutatorProxy.R' 'MutatorSequential.R' 'ParamSetShadow.R' 'OperatorCombination.R' 'Recombinator.R' 'Selector.R' 'mies_methods.R' 'OptimizerMies.R' 'RecombinatorCmpMaybe.R' 'RecombinatorConvex.R' 'RecombinatorConvexPair.R' 'RecombinatorCrossoverNary.R' 'RecombinatorCrossoverUniform.R' 'RecombinatorMaybe.R' 'RecombinatorNull.R' 'RecombinatorProxy.R'

'RecombinatorSequential.R'
'RecombinatorSimulatedBinaryCrossover.R' 'RecombinatorSwap.R'
'Scalor.R' 'ScalorAggregate.R' 'ScalorDomcount.R'
'ScalorFixedProjections.R' 'ScalorHypervolume.R'
'ScalorNondom.R' 'ScalorOne.R' 'ScalorProxy.R'
'ScalorSingleObjective.R' 'SelectorBest.R' 'SelectorMaybe.R'
'SelectorNull.R' 'SelectorProxy.R' 'SelectorRandom.R'
'SelectorSequential.R' 'SelectorTournament.R'
'TerminatorBudget.R' 'TerminatorGenerations.R' 'TunerMies.R'
'bibentries.R' 'repr.R' 'utils.R' 'utils_mo.R' 'zzz.R'

NeedsCompilation no

Author Martin Binder [aut, cre],
Lennart Schneider [ctb] (<https://orcid.org/0000-0003-4152-5308>),
Susanne Dandl [ctb] (<https://orcid.org/0000-0003-4324-4163>),
Andreas Hofheinz [ctb]

Maintainer Martin Binder <mlr.developer@mb706.com>

Repository CRAN

Date/Publication 2022-11-16 17:20:02 UTC

R topics documented:

miesmuschel-package 4
crate_env 5
dict_filtors 6
dict_filtors_maybe 7
dict_filtors_null 10
dict_filtors_proxy 11
dict_filtors_surprog 13
dict_filtors_surtour 16
dict_mutators 18
dict_mutators_cmpmaybe 19
dict_mutators_erase 22
dict_mutators_gauss 23
dict_mutators_maybe 25
dict_mutators_null 27
dict_mutators_proxy 29
dict_mutators_sequential 30
dict_mutators_unif 33
dict_recombinators 35
dict_recombinators_cmpmaybe 36
dict_recombinators_convex 38
dict_recombinators_cvxpair 40
dict_recombinators_maybe 42
dict_recombinators_null 45
dict_recombinators_proxy 47
dict_recombinators_sbx 49

dict_recombinators_sequential . . . 51
dict_recombinators_swap . . . 54
dict_recombinators_xonary . . . 55
dict_recombinators_xounif . . . 57
dict_scalors . . . 59
dict_scalors_aggregate . . . 59
dict_scalors_domcount . . . 62
dict_scalors_fixedprojection . . . 64
dict_scalors_hypervolume . . . 66
dict_scalors_nondom . . . 67
dict_scalors_one . . . 69
dict_scalors_proxy . . . 70
dict_scalors_single . . . 72
dict_selectors . . . 74
dict_selectors_best . . . 74
dict_selectors_maybe . . . 76
dict_selectors_null . . . 80
dict_selectors_proxy . . . 81
dict_selectors_random . . . 83
dict_selectors_sequential . . . 85
dict_selectors_tournament . . . 87
dist_crowding . . . 89
domhv_contribution . . . 90
domhv_improvement . . . 91
Filtor . . . 92
FiltorSurrogate . . . 94
MiesOperator . . . 97
mies_aggregate_generations . . . 101
mies_evaluate_offspring . . . 103
mies_filter_offspring . . . 106
mies_generate_offspring . . . 107
mies_generation . . . 109
mies_get_fitnesses . . . 111
mies_get_generation_results . . . 112
mies_init_population . . . 114
mies_prime_operators . . . 117
mies_select_from_archive . . . 119
mies_step_fidelity . . . 122
mies_survival_comma . . . 124
mies_survival_plus . . . 126
mlr_terminators_budget . . . 128
mlr_terminators_gens . . . 129
mut . . . 131
Mutator . . . 132
MutatorDiscrete . . . 134
MutatorNumeric . . . 136
OperatorCombination . . . 138
OptimizerMies . . . 145

ParamSetShadow . . . 153
rank_nondominated . . . 156
Recombinator . . . 156
RecombinatorPair . . . 159
repr . . . 161
SamplerRandomWeights . . . 162
Scalarizer . . . 163
scalarizer_chebyshev . . . 163
scalarizer_linear . . . 164
Scalor . . . 165
Selector . . . 167
SelectorScalar . . . 170
terminator_get_generations . . . 172

Index **174**

`miesmuschel-package` *miesmuschel: Mixed Integer Evolution Strategies*

Description

`miesmuschel` offers both an `Optimizer` and a `Tuner` for general MIES-optimization, as well as all the building blocks for building a custom optimization algorithm that is more flexible and can be used for research into novel evolution strategies.

The call-graph of the default algorithm in `OptimizerMies` / `TunerMies` is as follows, and is shown here as an overview over the `mies_*` functions, and how they are usually connected. (Note that only the exported `mies_*` functions are shown.) See the help information of these functions for more info.

```
OptimizerMies$.optimize(inst)
|- mies_prime_operators()  # prime operators on instance's search_space
|- mies_init_population()  # sample and evaluate first generation
|  `- mies_evaluate_offspring()  # evaluate sampled individuals
|     `- inst$eval_batch()  # The OptimInst's evaluation method
`- repeat # Repeat the following until terminated
   |- mies_step_fidelity()  # Evaluate individuals with changing fidelity
   |  `- inst$eval_batch()  # The OptimInst's evaluation method
   |- mies_generate_offspring()  # Sample parents, recombine, mutate
   |  `- mies_select_from_archive()  # Use 'Selector' on 'Archive'
   |     `- mies_get_fitnesses()  # Get objective values as fitness matrix
   |- mies_evaluate_offspring()  # evaluate sampled individuals
   |  `- inst$eval_batch()  # The OptimInst's evaluation method
   `- mies_survival_plus() / mies_survival_comma()  # survival
      `- mies_select_from_archive()  # Use 'Selector' on 'Archive'
```

Author(s)

Maintainer: Martin Binder <mlr.developer@mb706.com>

Other contributors:

- Lennart Schneider <lennart.sch@web.de> (ORCID) [contributor]
- Susanne Dandl <dandl.susanne@googlemail.com> (ORCID) [contributor]
- Andreas Hofheinz <andreas.hofheinz@outlook.com> [contributor]

See Also

Useful links:

- https://github.com/mlr-org/miesmuschel
- Report bugs at https://github.com/mlr-org/miesmuschel/issues

---

crate_env   *Set a Function's Environment*

Description

Useful to represent functions efficiently within `repr()`.

Usage

```
crate_env(fun, namespace = "R_GlobalEnv", ..., selfref = NULL)
```

Arguments

`fun`	(`function`) Function of which the environment should be set.
`namespace`	(`character(1)`) Name of the namespace, as given by `environmentName()`, to be used as the (parent of the) environment of `fun`. Special values `"R_GlobalEnv"` (global environment), `"R_BaseEnv"` (base environment; note this one is non-standard within R), `"R_EmptyEnv"` (empty environment). The content of the namespace is not modified. Default `"R_GlobalEnv"`.
`...`	(`list`) Content of environments within which to place `fun`.
`selfref`	(`character(1)` | named `integer` | `NULL`) If `character(1)`: The name of the entry of the first element in `...` that refers to the function itself. If a named `integer`, then the values indicate the lists in `...` where the reference to the function should be placed see examples. Default `NULL`: No reference to the function itself is present.

Value

`function`: The given `fun` with changed environment.

Examples

```
identity2 = crate_env(function(x) x, "base")
identical(identity, identity2)  # TRUE

y = 1
f1 = mlr3misc::crate(function(x) x + y, y, .parent = .GlobalEnv)
f2 = crate_env(function(x) x + y, "R_GlobalEnv", list(y = 1))

# Note identical() does not apply because both contain (equal, but not
# identical) 'y = 1'-environments
all.equal(f1, f2)  # TRUE
f1(10)  # 10 + 1 == 11

factorial1 = mlr3misc::crate(
  function(x) if (x > 0) x * factorial1(x - 1) else 1,
  y, .parent = .GlobalEnv
)
environment(factorial1)$factorial1 = factorial1

factorial2 = crate_env(
  function(x) if (x > 0) x * factorial1(x - 1) else 1,
  "R_GlobalEnv", list(y = 1), selfref = "factorial1")
# putting 'factorial1' into the list (or repeating function(x) ....)
# would *not* work, since we want:
identical(environment(factorial2)$factorial1, factorial2)  # TRUE

all.equal(factorial1, factorial2)  # TRUE

g = crate_env(function(x) x + y + z, "miesmuschel",
  list(y = 1), list(z = 2), selfref = c(X = 1, Y = 2, Z = 2))
g(0)  # 0 + 1 + 2 == 3
identical(environment(g)$X, g)
identical(parent.env(environment(g))$Y, g)
identical(parent.env(environment(g))$Z, g)
identical(
  parent.env(parent.env(environment(g))),
  loadNamespace("miesmuschel")
)
```

`dict_filtors` *Dictionary of Filtors*

Description

Dictionary of Filtors

Usage

```
dict_filtors
```

Format

An object of class DictionaryFiltor (inherits from DictionaryEx, Dictionary, R6) of length 15.

Methods

Methods inherited from Dictionary, as well as:

- help(key, help_type)
  (character(1), character(1))
  Displays help for the dictionary entry key. help_type is one of "text", "html", "pdf" and given as the help_type argument of R's help().

See Also

Other dictionaries: dict_mutators, dict_recombinators, dict_scalors, dict_selectors, mut()

---

dict_filtors_maybe *Filtor-Combination that Filters According to Two Filtors*

Description

Filtor that wraps two other Filtors given during construction and chooses which operation to perform. Each of the resulting n_filter individuals is chosen either from $filtor, or from $filtor_not.

This makes it possible to implement filter methods such as random interleaving, where only a fraction of p individuals were filtered and the others were not.

Letting the number of individuals chosen by $filtor be n_filter_f, then n_filter_f is either fixed set to round(n_filter * p), (when random_choise is FALSE) or to rbinom(1, n_filter, p) (when random_choice is TRUE).

When random_choice is FALSE, then $needed_input() is calculated directly from $needed_input() of $filtor and $filtor_not, as well as n_filter_f and n_filter - n_filter_f.

When random_choice is TRUE, then $needed_input() is considers the "worst case" from $filtor and $filtor_not, and assumes that $needed_input() is monotonically increasing in its input argument.

To make the worst case less extreme, the number of individuals chosen with random_choice set to TRUE is limited to qbinom(-20, n_filter, p, log.p = TRUE) (with lower.tail FALSE and TRUE for $filtor and $filtor_not, respectively), which distorts the binomial distribution with probability 1 - exp(-20) or about 1 - 0.5e-9.

Configuration Parameters

This operator has the configuration parameters of the Filtors that it wraps: The configuration parameters of the operator given to the `filtor` construction argument are prefixed with `"maybe."`, the configuration parameters of the operator given to the `filtor_not` construction argument are prefixed with `"maybe_not."`.

Additional configuration parameters:

- `p` :: `numeric(1)`
  Probability per individual (when `random_choise` is `TRUE`), or fraction of individuals (when `random_choice` is `FALSE`), that are chosen from `$filtor` instead of `$filtor_not`. Must be set by the user.
- `random_choice` :: `logical(1)`
  Whether to sample the number of individuals chosen by `$filtor` according to `rbinom(1, n_filter, p)`, or to use a fixed fraction. Initialized to `FALSE`.

Supported Operand Types

Supported `Param` classes are the set intersection of supported classes of `filtor` and `filtor_not`.

Dictionary

This `Filtor` can be created with the short access form `ftr()` (`ftrs()` to get a list), or through the the dictionary `dict_filtors` in the following way:

```
# preferred:
ftr("maybe", <filtor> [, <filtor_not>])
ftrs("maybe", <filtor> [, <filtor_not>])  # takes vector IDs, returns list of Filtors

# long form:
dict_filtors$get("maybe", <filtor> [, <filtor_not>])
```

Super classes

`miesmuschel::MiesOperator` -> `miesmuschel::Filtor` -> `FiltorMaybe`

Active bindings

`filtor` (`Filtor`)
: `Filtor` being wrapped. This operator gets run with probability / proportion `p` (configuration parameter).

`filtor_not` (`Filtor`)
: Alternative `Filtor` being wrapped. This operator gets run with probability / proportion `1 - p` (configuration parameter).

Methods

Public methods:

- `FiltorMaybe$new()`

- FiltorMaybe$prime()
- FiltorMaybe$clone()

Method `new()`**:** Initialize the `FiltorMaybe` object.

Usage:

```
FiltorMaybe$new(filtor, filtor_not = FiltorNull$new())
```

Arguments:

`filtor` (Filtor)
: Filtor to wrap. This operator gets run with probability p (Configuration parameter).
The constructed object gets a *clone* of this argument. The `$filtor` field will reflect this value.

`filtor_not` (Filtor)
: Another Filtor to wrap. This operator runs when `filtor` is not chosen. By default, this is FiltorNull, i.e. no filtering. With this default, the `FiltorMaybe` object applies the `filtor` operation with probability / proportion p, and no operation at all otherwise.
The constructed object gets a *clone* of this argument. The `$filtor_not` field will reflect this value.

Method `prime()`**:** See MiesOperator method. Primes both this operator, as well as the wrapped operators given to `filtor` and `filtor_not` during construction.

Usage:

```
FiltorMaybe$prime(param_set)
```

Arguments:

`param_set` (ParamSet)
: Passed to MiesOperator`$prime()`.

Returns: invisible `self`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FiltorMaybe$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other filtors: FiltorSurrogate, Filtor, dict_filtors_null, dict_filtors_proxy, dict_filtors_surprog, dict_filtors_surtour

Other filtor wrappers: dict_filtors_proxy

Examples

```
library("mlr3")
library("mlr3learners")

fm = ftr("maybe", ftr("surprog", lrn("regr.lm"), filter.pool_factor = 2), p = 0.5)
p = ps(x = p_dbl(-5, 5))
```

```
known_data = data.frame(x = 1:5)
fitnesses = 1:5
new_data = data.frame(x = c(0.5, 1.5, 2.5, 3.5, 4.5))

fm$prime(p)

fm$needed_input(2)

fm$operate(new_data, known_data, fitnesses, 2)

fm$param_set$values$p = 0.33

fm$needed_input(3)

fm$operate(new_data, known_data, fitnesses, 3)
```

dict_filtors_null *Null-Filtor*

Description

Null-filtor that does not perform filtering. Its `needed_input()` is always the `output_size`, and `operate()` selects the first `n_filter` values from its input.

Useful in particular with operator-wrappers such as `FiltorProxy`, and to make filtering optional.

Configuration Parameters

This operator has no configuration parameters.

Supported Operand Types

Supported `Param` classes are: `ParamLgl`, `ParamInt`, `ParamDbl`, `ParamFct`

Dictionary

This `Filtor` can be created with the short access form `ftr()` (`ftrs()` to get a list), or through the the dictionary `dict_filtors` in the following way:

```
# preferred:
ftr("null")
ftrs("null")  # takes vector IDs, returns list of Filtors

# long form:
dict_filtors$get("null")
```

Super classes

`miesmuschel::MiesOperator` -> `miesmuschel::Filtor` -> `FiltorNull`

Methods

Public methods:

- FiltorNull$new()
- FiltorNull$clone()

Method `new()`**:** Initialize the `FiltorNull` object.

Usage:

```
FiltorNull$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FiltorNull$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other filtors: `FiltorSurrogate`, `Filtor`, `dict_filtors_maybe`, `dict_filtors_proxy`, `dict_filtors_surprog`, `dict_filtors_surtour`

Examples

```
fn = ftr("null")

p = ps(x = p_dbl(-5, 5))
known_data = data.frame(x = 1:5)
fitnesses = 1:5

new_data = data.frame(x = c(2.5, 4.5))

fn$prime(p)

fn$needed_input(1)

fn$operate(new_data, known_data, fitnesses, 1)
```

`dict_filtors_proxy` *Proxy-Filtor that Filters According to its Configuration Parameter*

Description

Filtor that performs the operation in its `operation` configuration parameter. This can be used to make filtor operations fully parametrizable.

Configuration Parameters

- operation :: Filtor
  Operation to perform. Must be set by the user. This is primed when $prime() of SelectorProxy is called, and also when $operate() is called, to make changing the operation as part of self-adaption possible. However, if the same operation gets used inside multiple SelectorProxy objects, then it is recommended to $clone(deep = TRUE) the object before assigning them to operation to avoid frequent re-priming.

Supported Operand Types

Supported Param classes are: ParamLgl, ParamInt, ParamDbl, ParamFct

Dictionary

This Selector can be created with the short access form sel() (sels() to get a list), or through the the dictionary dict_selectors in the following way:

```
# preferred:
sel("proxy")
sels("proxy")  # takes vector IDs, returns list of Selectors

# long form:
dict_selectors$get("proxy")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Filtor -> FiltorProxy

Methods

Public methods:

- FiltorProxy$new()
- FiltorProxy$prime()
- FiltorProxy$clone()

Method new()**:** Initialize the FiltorProxy object.

Usage:

```
FiltorProxy$new()
```

Method prime()**:** See MiesOperator method. Primes both this operator, as well as the operator given to the operation configuration parameter. Note that this modifies the $param_set$values$operation object.

Usage:

```
FiltorProxy$prime(param_set)
```

Arguments:

param_set (ParamSet)
  Passed to MiesOperator$prime().

Returns: invisible `self`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FiltorProxy$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other filtors: `FiltorSurrogate`, `Filtor`, `dict_filtors_maybe`, `dict_filtors_null`, `dict_filtors_surprog`, `dict_filtors_surtour`

Other filtor wrappers: `dict_filtors_maybe`

Examples

```
library("mlr3")
library("mlr3learners")
fp = ftr("proxy")
p = ps(x = p_dbl(-5, 5))
known_data = data.frame(x = 1:5)
fitnesses = 1:5
new_data = data.frame(x = c(2.5, 4.5))

fp$param_set$values$operation = ftr("null")
fp$prime(p)
fp$operate(new_data, known_data, fitnesses, 1)

fp$param_set$values$operation = ftr("surprog", lrn("regr.lm"), filter.pool_factor = 2)
fp$operate(new_data, known_data, fitnesses, 1)
```

`dict_filtors_surprog` *Progressive Surrogate Model Filtering*

Description

Performs progressive surrogate model filtering. A surrogate model is used, as described in the parent class `FiltorSurrogate`. The filtering is "progressive" in that successive values are filtered more agressively.

Algorithm

Given the number `n_filter` of of individuals to sample, and the desired pool size at round `i` `pool_size(i)`, progressive surrogate model filtering proceeds as follows:

1. Train the `surrogate_learner` `LearnerRegr` on the `known_values` and their `fitnesses`.
2. Take `pool_size(1)` configurations, predict their expected performance using the surrogate model, and put them into a pool `P` of configurations to consider.

3. Initialize i to 1.
4. Take the individual that is optimal according to predicted performance, remove it from P and add it to solution set S.
5. If the number of solutions in S equals n_filter, quit.
6. If pool_size(i + 1) is larger than pool_size(i), take the next pool_size(i + 1) - pool_size(i) configurations, predict their expected performance using the surrogate model, and add them to P. Otherwise, remove pool_size(i) - pool_size(i + 1) random individuals from the pool. The size of P ends up being pool_size(i + 1) - i, as i individuals have also been removed and added to S.
7. Increment i, jump to 4.

(The algorithm presented here is optimized for clarity; the actual implementation does all the surrogate model prediction in one go, but is functionally equivalent).

pool_size(i) is calculated as round(n_filter * pool_factor * (pool_factor_last / pool_factor) ^ (i / n_filter)), i.e. a log-linear interpolation from pool_factor * n_filter to pool_factor_last * n_filter.

The pool_factor and pool_factor_last configuration parameters of this algorithm determine how agressively the surrogate model is used to filter out sampled configurations. If the filtering is agressive (large values), then more "exploitation" at the cost of "exploration" is performed. When pool_factor is small but pool_factor_last is large (or vice-versa), then different individuals are filtered with different agressiveness, potentially leading to a tradeoff between "exploration" and "exploitation".

When pool_factor_last is set, it defaults to pool_factor, with no new individuals added and no individuals removed from the filter pool during filtering. It is equivalent to taking the top n_filter individuals out of a sample of n_filter * pool_factor.

Configuration Parameters

FiltorSurrogateProgressive's configuration parameters are the hyperparameters of the FiltorSurrogate base class, as well as:

- filter.pool_factor :: numeric(1)
  pool_factor parameter of the progressive surrogate model filtering algorithm, see the corresponding section. Initialized to 1. Together with the default of filter.pool_factor_last, this is equivalent to random sampling new individuals.
- filter.pool_factor_last :: numeric(1)
  pool_factor_last parameter of the progressive surrogate model filtering algorithm, see the corresponding section. When not given, it defaults to filter.pool_factor, equivalent to taking the top n_filter from n_filter * pool_factor individuals.

Supported Operand Types

See FiltorSurrogate about supported operand types.

Dictionary

This `Filtor` can be created with the short access form `ftr()` (`ftrs()` to get a list), or through the the dictionary `dict_filtors` in the following way:

```
# preferred:
ftr("surprog", <surrogate_learner> [, <surrogate_selector>])
ftrs("surprog", <surrogate_learner> [, <surrogate_selector>])  # takes vector IDs, returns list of Filt

# long form:
dict_filtors$get("surprog", <surrogate_learner> [, <surrogate_selector>])
```

Super classes

`miesmuschel::MiesOperator` -> `miesmuschel::Filtor` -> `miesmuschel::FiltorSurrogate` -> `FiltorSurrogateProgressive`

Methods

Public methods:

- `FiltorSurrogateProgressive$new()`
- `FiltorSurrogateProgressive$clone()`

Method `new()`**:** Initialize the `FiltorSurrogateProgressive`.

Usage:

```
FiltorSurrogateProgressive$new(
  surrogate_learner,
  surrogate_selector = SelectorBest$new()
)
```

Arguments:

`surrogate_learner` (`mlr3::LearnerRegr`)
Regression learner for the surrogate model filtering algorithm.
The `$surrogate_learner` field will reflect this value.

`surrogate_learner` (`mlr3::LearnerRegr`)
Regression learner for the surrogate model filtering algorithm.
The `$surrogate_learner` field will reflect this value.

`surrogate_selector` (`Selector`) `Selector` for the surrogate model filtering algorithm.
The `$surrogate_selector` field will reflect this value.

`surrogate_selector` (`Selector`) `Selector` for the surrogate model filtering algorithm.
The `$surrogate_selector` field will reflect this value.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FiltorSurrogateProgressive$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other filtors: FiltorSurrogate, Filtor, dict_filtors_maybe, dict_filtors_null, dict_filtors_proxy, dict_filtors_surtour

Examples

```
library("mlr3")
library("mlr3learners")
fp = ftr("surprog", lrn("regr.lm"), filter.pool_factor = 2)

p = ps(x = p_dbl(-5, 5))
known_data = data.frame(x = 1:5)
fitnesses = 1:5
new_data = data.frame(x = c(2.5, 4.5))

fp$prime(p)

fp$needed_input(1)

fp$operate(new_data, known_data, fitnesses, 1)
```

---

dict_filtors_surtour *Tournament Surrogate Model Filtering*

Description

Performs tournament surrogate model filtering. A surrogate model is used, as described in the parent class FiltorSurrogate.

Algorithm

Selects individuals from a tournament by taking the top `per_tournament` individuals, according to `surrogate_selector` and as predicted by `surrogate_learner`, from a sample of `tournament_size(i)`, where `tournament_size(1)` is given by `tournament_size`, `tournament_size(ceiling(n_filter / per_tournament))` is given by `tournament_size_last`, and `tournament_size(i)` for `i` between these values is linearly interpolated on a log scale.

Configuration Parameters

`FiltorSurrogateProgressive`'s configuration parameters are the hyperparameters of the FiltorSurrogate base class, as well as:

- `filter.per_tournament` :: `integer(1)`
  Number of individuals to select from each tournament. If `per_tournament` is not a divider of `n_filter`, then the last tournament selects a random subset of `n_filter %% per_tournament` individuals out of the top `per_tournament` individuals. Initialized to 1.

- `filter.tournament_size` :: `numeric(1)`
  Tournament size used for filtering. If `tournament_size_last` is not given, all `n_filter` individuals are selected in batches of `per_tournament` from tournaments of this size. If it is given, then the actual tournament size is interpolated between `tournament_size` and `tournament_size_last` on a logarithmic scale. Tournaments with tournament size below `per_tournament` select `per_tournament` individuals without tournament, i.e. no filtering. Initialized to 1.
- `filter.tournament_size_last` :: `numeric(1)`
  Tournament size used for the last tournament, see description of `tournament_size`. Defaults to `tournament_size` when not given, i.e. all tournaments have the same size.

Supported Operand Types

See `FiltorSurrogate` about supported operand types.

Dictionary

This `Filtor` can be created with the short access form `ftr()` (`ftrs()` to get a list), or through the the dictionary `dict_filtors` in the following way:

```
# preferred:
ftr("surtour", <surrogate_learner> [, <surrogate_selector>])
ftrs("surtour", <surrogate_learner> [, <surrogate_selector>])  # takes vector IDs, returns list of Filt

# long form:
dict_filtors$get("surtour", <surrogate_learner> [, <surrogate_selector>])
```

Super classes

`miesmuschel::MiesOperator` -> `miesmuschel::Filtor` -> `miesmuschel::FiltorSurrogate` -> `FiltorSurrogateTournament`

Methods

Public methods:

- `FiltorSurrogateTournament$new()`
- `FiltorSurrogateTournament$clone()`

Method `new()`**:** Initialize the `FiltorSurrogateTournament`.

Usage:

```
FiltorSurrogateTournament$new(
  surrogate_learner,
  surrogate_selector = SelectorBest$new()
)
```

Arguments:

`surrogate_learner` (`mlr3::LearnerRegr`)
Regression learner for the surrogate model filtering algorithm.
The `$surrogate_learner` field will reflect this value.

`surrogate_learner` (`mlr3::LearnerRegr`)
Regression learner for the surrogate model filtering algorithm.
The `$surrogate_learner` field will reflect this value.

`surrogate_selector` (`Selector`) `Selector` for the surrogate model filtering algorithm.
The `$surrogate_selector` field will reflect this value.

`surrogate_selector` (`Selector`) `Selector` for the surrogate model filtering algorithm.
The `$surrogate_selector` field will reflect this value.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FiltorSurrogateTournament$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other filtors: `FiltorSurrogate`, `Filtor`, `dict_filtors_maybe`, `dict_filtors_null`, `dict_filtors_proxy`, `dict_filtors_surprog`

Examples

```
library("mlr3")
library("mlr3learners")
fp = ftr("surtour", lrn("regr.lm"), filter.tournament_size = 2)

p = ps(x = p_dbl(-5, 5))
known_data = data.frame(x = 1:5)
fitnesses = 1:5
new_data = data.frame(x = c(2.5, 4.5))

fp$prime(p)

fp$needed_input(1)

fp$operate(new_data, known_data, fitnesses, 1)
```

`dict_mutators` *Dictionary of Mutators*

Description

Dictionary of Mutators

Usage

```
dict_mutators
```

Format

An object of class DictionaryMutator (inherits from DictionaryEx, Dictionary, R6) of length 15.

Methods

Methods inherited from Dictionary, as well as:

- help(key, help_type)
  (character(1), character(1))
  Displays help for the dictionary entry key. help_type is one of "text", "html", "pdf" and given as the help_type argument of R's help().

See Also

Other dictionaries: dict_filtors, dict_recombinators, dict_scalors, dict_selectors, mut()

---

dict_mutators_cmpmaybe

Mutator Choosing Action Component-Wise Independently

Description

Mutator that chooses which operation to perform probabilistically. The Mutator wraps two other Mutators given during construction, and both of these operators are run. The ultimate result is sampled from the results of these operations independently for each individuum and component: with probability p (configuration parameter), the result from the Mutator given to the mutator construction argument is used, and with probability p - 1 the one given to mutator_not is used.

Configuration Parameters

This operator has the configuration parameters of the Mutators that it wraps: The configuration parameters of the operator given to the mutator construction argument are prefixed with "cmpmaybe.", the configuration parameters of the operator given to the mutator_not construction argument are prefixed with "cmpmaybe_not.".

Additional configuration parameters:

- p :: numeric(1)
  Probability per component with which to apply the operator given to the mutator construction argument. Must be set by the user.

Supported Operand Types

Supported Param classes are the set intersection of supported classes of mutator and mutator_not.

Dictionary

This Mutator can be created with the short access form mut() (muts() to get a list), or through the the dictionary dict_mutators in the following way:

```
# preferred:
mut("cmpmaybe", <mutator> [, <mutator_not>])
muts("cmpmaybe", <mutator> [, <mutator_not>]) # takes vector IDs, returns list of Mutators

# long form:
dict_mutators$get("cmpmaybe", <mutator> [, <mutator_not>])
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Mutator -> MutatorCmpMaybe

Active bindings

mutator (Mutator)
: Mutator being wrapped. This operator gets run with probability p (configuration parameter).

mutator_not (Mutator)
: Alternative Mutator being wrapped. This operator gets run with probability 1 - p (configuration parameter).

Methods

Public methods:

- MutatorCmpMaybe$new()
- MutatorCmpMaybe$prime()
- MutatorCmpMaybe$clone()

Method new()**:** Initialize the MutatorCmpMaybe object.

Usage:

```
MutatorCmpMaybe$new(mutator, mutator_not = MutatorNull$new())
```

Arguments:

mutator (Mutator)
: Mutator to wrap. Component-wise results of this operator are used with probability p (Configuration parameter).
The constructed object gets a *clone* of this argument. The $mutator field will reflect this value.

mutator_not (Mutator)
: Another Mutator to wrap. Results from this operator are used when mutator is not chosen. By default, this is MutatorNull, i.e. no operation.
With this default, the MutatorCmpMaybe object applies the mutator operation with probability p, and no operation at all otherwise.
The constructed object gets a *clone* of this argument. The $mutator_not field will reflect this value.

Method `prime()`**:** See `MiesOperator` method. Primes both this operator, as well as the wrapped operators given to `mutator` and `mutator_not` during construction.

Usage:

```
MutatorCmpMaybe$prime(param_set)
```

Arguments:

`param_set` (`ParamSet`)
Passed to `MiesOperator$prime()`.

Returns: invisible `self`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MutatorCmpMaybe$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other mutators: `MutatorDiscrete`, `MutatorNumeric`, `Mutator`, `OperatorCombination`, `dict_mutators_erase`, `dict_mutators_gauss`, `dict_mutators_maybe`, `dict_mutators_null`, `dict_mutators_proxy`, `dict_mutators_sequential`, `dict_mutators_unif`

Other mutator wrappers: `OperatorCombination`, `dict_mutators_maybe`, `dict_mutators_proxy`, `dict_mutators_sequential`

Examples

```
set.seed(1)
mcm = mut("cmpmaybe", mut("gauss", sdev = 5), p = 0.5)
p = ps(x = p_int(-5, 5), y = p_dbl(-5, 5))
data = data.frame(x = rep(0, 5), y = rep(0, 5))

mcm$prime(p)
mcm$operate(data)

mcm$param_set$values$p = 0.2
mcm$operate(data)

mcm2 = mut("cmpmaybe",
  mutator = mut("gauss", sdev = 0.01),
  mutator_not = mut("gauss", sdev = 10),
  p = 0.5
)

mcm2$prime(p)
mcm2$operate(data)
```

dict_mutators_erase *Uniform Sample Mutator*

Description

"Mutates" individuals by forgetting the current value and sampling new individuals from scratch.

Since the information loss is very high, this should in most cases be combined with MutatorCmpMaybe.

Configuration Parameters

- initializer :: function
  Function that generates the initial population as a Design object, with arguments param_set and n, functioning like paradox::generate_design_random or paradox::generate_design_lhs. This is equivalent to the initializer parameter of mies_init_population(), see there for more information. Initialized to generate_design_random().

Supported Operand Types

Supported Param classes are: ParamLgl, ParamInt, ParamDbl, ParamFct

Dictionary

This Mutator can be created with the short access form mut() (muts() to get a list), or through the the dictionary dict_mutators in the following way:

```
# preferred:
mut("erase")
muts("erase")  # takes vector IDs, returns list of Mutators

# long form:
dict_mutators$get("erase")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Mutator -> MutatorErase

Methods

Public methods:

- MutatorErase$new()
- MutatorErase$clone()

Method new(): Initialize the MutatorErase object.

Usage:

```
MutatorErase$new()
```

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
MutatorErase$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other mutators: MutatorDiscrete, MutatorNumeric, Mutator, OperatorCombination, dict_mutators_cmpmaybe, dict_mutators_gauss, dict_mutators_maybe, dict_mutators_null, dict_mutators_proxy, dict_mutators_sequential, dict_mutators_unif

Examples

```
set.seed(1)
mer = mut("erase")
p = ps(x = p_lgl(), y = p_fct(c("a", "b", "c")), z = p_dbl(0, 1))
data = data.frame(x = rep(TRUE, 5), y = rep("a", 5),
  z = seq(0, 1, length.out = 5),
  stringsAsFactors = FALSE)  # necessary for R <= 3.6

mer$prime(p)
mer$operate(data)
```

---

dict_mutators_gauss *Gaussian Distribution Mutator*

Description

Individuals are mutated with an independent normal random variable on each component.

Configuration Parameters

- `sdev` :: `numeric`
  Standard deviation of normal distribuion. This is absolute if `sdev_is_relative` is `FALSE`, and multiplied with each individual component's range (upper - lower) if `sdev_is_relative` is `TRUE`. This may either be a scalar, in which case it is applied to all input components, or a vector, in which case it must have the length of the input and applies to components in order in which they appear in the priming ParamSet. Must be set by the user.
- `sdev_is_relative` :: `logical(1)`
  Whether `sdev` is absolute (`FALSE`) or relative to component range (`TRUE`). Initialized to `FALSE`.
- `truncated_normal` :: `logical(1)`
  Whether to draw individuals from a normal distribution that is truncated at the bounds of each component (`TRUE`), or to draw from a normal distribution and then restrict to bounds afterwards (`FALSE`). The former (`TRUE`) will lead to very few to no samples landing on the exact bounds (analytically it would be none almost surely, but this is subject to machine precision), the latter (`FALSE`) can lead to a substantial number of samples landing on the exact bounds. Initialized to `FALSE`.

Supported Operand Types

Supported Param classes are: ParamInt, ParamDbl

Dictionary

This Mutator can be created with the short access form mut() (muts() to get a list), or through the the dictionary dict_mutators in the following way:

```
# preferred:
mut("gauss")
muts("gauss")  # takes vector IDs, returns list of Mutators

# long form:
dict_mutators$get("gauss")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Mutator -> miesmuschel::MutatorNumeric -> MutatorGauss

Methods

Public methods:

- MutatorGauss$new()
- MutatorGauss$clone()

Method `new()`**:** Initialize the MutatorGauss object.

Usage:

`MutatorGauss$new()`

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`MutatorGauss$clone(deep = FALSE)`

Arguments:

deep Whether to make a deep clone.

See Also

Other mutators: MutatorDiscrete, MutatorNumeric, Mutator, OperatorCombination, dict_mutators_cmpmaybe, dict_mutators_erase, dict_mutators_maybe, dict_mutators_null, dict_mutators_proxy, dict_mutators_sequential, dict_mutators_unif

Examples

```
set.seed(1)
mg = mut("gauss", sdev = 0.1)
p = ps(x = p_int(-5, 5), y = p_dbl(-5, 5))
data = data.frame(x = rep(0, 5), y = rep(0, 5))

mg$prime(p)
mg$operate(data)

mg$param_set$values$sdev = 100
mg$operate(data)
```

---

dict_mutators_maybe *Mutator Choosing Action Probabilistically*

Description

`Mutator` that chooses which operation to perform probabilistically. The `Mutator` wraps two other `Mutator`s given during construction, and for each individuum, the operation to perform is sampled: with probability `p` (configuration parameter), the `Mutator` given to the `mutator` construction argument is applied, and with probability `p - 1` the one given to `mutator_not` is applied.

Configuration Parameters

This operator has the configuration parameters of the `Mutator`s that it wraps: The configuration parameters of the operator given to the `mutator` construction argument are prefixed with `"maybe."`, the configuration parameters of the operator given to the `mutator_not` construction argument are prefixed with `"maybe_not."`.

Additional configuration parameters:

- `p` :: `numeric(1)`
  Probability per individual with which to apply the operator given to the `mutator` construction argument. Must be set by the user.

Supported Operand Types

Supported `Param` classes are the set intersection of supported classes of `mutator` and `mutator_not`.

Dictionary

This `Mutator` can be created with the short access form `mut()` (`muts()` to get a list), or through the the dictionary `dict_mutators` in the following way:

```
# preferred:
mut("maybe", <mutator> [, <mutator_not>])
muts("maybe", <mutator> [, <mutator_not>])  # takes vector IDs, returns list of Mutators

# long form:
dict_mutators$get("maybe", <mutator> [, <mutator_not>])
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Mutator -> MutatorMaybe

Active bindings

mutator (Mutator)
: Mutator being wrapped. This operator gets run with probability p (configuration parameter).

mutator_not (Mutator)
: Alternative Mutator being wrapped. This operator gets run with probability 1 - p (configuration parameter).

Methods

Public methods:

- MutatorMaybe$new()
- MutatorMaybe$prime()
- MutatorMaybe$clone()

Method `new()`**:** Initialize the MutatorMaybe object.

Usage:

```
MutatorMaybe$new(mutator, mutator_not = MutatorNull$new())
```

Arguments:

mutator (Mutator)
: Mutator to wrap. This operator gets run with probability p (configuration parameter).
The constructed object gets a *clone* of this argument.
The $mutator field will reflect this value.

mutator_not (Mutator)
: Another Mutator to wrap. This operator runs when mutator is not chosen. By default, this is MutatorNull, i.e. no operation. With this default, the MutatorMaybe object applies the mutator operation with probability p, and no operation at all otherwise.
The constructed object gets a *clone* of this argument. The $mutator_not field will reflect this value.

Method `prime()`**:** See MiesOperator method. Primes both this operator, as well as the wrapped operators given to mutator and mutator_not during construction.

Usage:

```
MutatorMaybe$prime(param_set)
```

Arguments:

param_set (ParamSet)
: Passed to MiesOperator$prime().

Returns: invisible self.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MutatorMaybe$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other mutators: MutatorDiscrete, MutatorNumeric, Mutator, OperatorCombination, dict_mutators_cmpmaybe, dict_mutators_erase, dict_mutators_gauss, dict_mutators_null, dict_mutators_proxy, dict_mutators_sequential, dict_mutators_unif

Other mutator wrappers: OperatorCombination, dict_mutators_cmpmaybe, dict_mutators_proxy, dict_mutators_sequential

Examples

```
set.seed(1)
mm = mut("maybe", mut("gauss", sdev = 5), p = 0.5)
p = ps(x = p_int(-5, 5), y = p_dbl(-5, 5))
data = data.frame(x = rep(0, 5), y = rep(0, 5))

mm$prime(p)
mm$operate(data)

mm$param_set$values$p = 0.3
mm$operate(data)

mm2 = mut("maybe",
  mutator = mut("gauss", sdev = 0.01),
  mutator_not = mut("gauss", sdev = 10),
  p = 0.5
)

mm2$prime(p)
mm2$operate(data)
```

dict_mutators_null *Null Mutator*

Description

Null-mutator that does not perform any operation on its input. Useful in particular with operator-wrappers such as MutatorMaybe or MutatorCombination.

Configuration Parameters

This operator has no configuration parameters.

Supported Operand Types

Supported Param classes are: ParamLgl, ParamInt, ParamDbl, ParamFct

Dictionary

This Mutator can be created with the short access form mut() (muts() to get a list), or through the the dictionary dict_mutators in the following way:

```
# preferred:
mut("null")
muts("null")  # takes vector IDs, returns list of Mutators

# long form:
dict_mutators$get("null")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Mutator -> MutatorNull

Methods

Public methods:

- MutatorNull$new()
- MutatorNull$clone()

Method new()**:** Initialize the MutatorNull object.

Usage:

```
MutatorNull$new()
```

Method clone()**:** The objects of this class are cloneable with this method.

Usage:

```
MutatorNull$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other mutators: MutatorDiscrete, MutatorNumeric, Mutator, OperatorCombination, dict_mutators_cmpmaybe, dict_mutators_erase, dict_mutators_gauss, dict_mutators_maybe, dict_mutators_proxy, dict_mutators_sequential, dict_mutators_unif

Examples

```
mn = mut("null")
p = ps(x = p_int(-5, 5), y = p_dbl(-5, 5), z = p_lgl())
data = data.frame(x = rep(0, 5), y = rep(0, 5), z = rep(TRUE, 5))

mn$prime(p)
mn$operate(data)
```

`dict_mutators_proxy` *Proxy-Mutator that Mutates According to its Configuration parameter*

Description

Mutator that performs the operation in its `operation` configuration parameter. This is useful, e.g., to make `OptimizerMies`'s mutation operation fully parametrizable.

Configuration Parameters

- `operation` :: `Mutator`
  Operation to perform. Must be set by the user. This is primed when `$prime()` of `MutatorProxy` is called, and also when `$operate()` is called, to make changing the operation as part of self-adaption possible. However, if the same operation gets used inside multiple `MutatorProxy` objects, then it is recommended to `$clone(deep = TRUE)` the object before assigning them to `operation` to avoid frequent re-priming.

Supported Operand Types

Supported `Param` classes are: `ParamLgl`, `ParamInt`, `ParamDbl`, `ParamFct`

Dictionary

This `Mutator` can be created with the short access form `mut()` (`muts()` to get a list), or through the the dictionary `dict_mutators` in the following way:

```
# preferred:
mut("proxy")
muts("proxy")  # takes vector IDs, returns list of Mutators

# long form:
dict_mutators$get("proxy")
```

Super classes

`miesmuschel::MiesOperator` -> `miesmuschel::Mutator` -> `MutatorProxy`

Methods

Public methods:

- `MutatorProxy$new()`
- `MutatorProxy$prime()`
- `MutatorProxy$clone()`

Method `new()`**:** Initialize the `MutatorProxy` object.

Usage:

```
MutatorProxy$new()
```

Method `prime()`**:** See `MiesOperator` method. Primes both this operator, as well as the operator given to the `operation` configuration parameter. Note that this modifies the `$param_set$values$operation` object.

Usage:

```
MutatorProxy$prime(param_set)
```

Arguments:

`param_set` (`ParamSet`)
Passed to `MiesOperator$prime()`.

Returns: invisible `self`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MutatorProxy$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other mutators: `MutatorDiscrete`, `MutatorNumeric`, `Mutator`, `OperatorCombination`, `dict_mutators_cmpmaybe`, `dict_mutators_erase`, `dict_mutators_gauss`, `dict_mutators_maybe`, `dict_mutators_null`, `dict_mutators_sequential`, `dict_mutators_unif`

Other mutator wrappers: `OperatorCombination`, `dict_mutators_cmpmaybe`, `dict_mutators_maybe`, `dict_mutators_sequential`

Examples

```
set.seed(1)
mp = mut("proxy", operation = mut("gauss", sdev = 0.1))
p = ps(x = p_int(-5, 5), y = p_dbl(-5, 5))
data = data.frame(x = rep(0, 5), y = rep(0, 5))

mp$prime(p)
mp$operate(data)

mp$param_set$values$operation = mut("null")
mp$operate(data)
```

`dict_mutators_sequential`

Run Multiple Mutator Operations in Sequence

Description

`Mutator` that wraps multiple other `Mutator`s given during construction and uses them for mutation in sequence.

Configuration Parameters

This operator has the configuration parameters of the Mutators that it wraps: The configuration parameters of the operator given to the mutators construction argument are prefixed with "mutator_1", "mutator_2", ... up to "mutator_#", where # is length(mutators).

Supported Operand Types

Supported Param classes are the set intersection of supported classes of the Mutators given in mutators.

Dictionary

This Mutator can be created with the short access form mut() (muts() to get a list), or through the the dictionary dict_mutators in the following way:

```
# preferred:
mut("sequential", <mutators>)
muts("sequential", <mutators>)  # takes vector IDs, returns list of Mutators

# long form:
dict_mutators$get("sequential", <mutators>)
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Mutator -> MutatorSequential

Active bindings

mutators (list of Mutator)
: Mutators being wrapped. These operators get run sequentially in order.

Methods

Public methods:

- MutatorSequential$new()
- MutatorSequential$prime()
- MutatorSequential$clone()

Method new()**:** Initialize the MutatorSequential object.

Usage:

```
MutatorSequential$new(mutators)
```

Arguments:

mutators (list of Mutator)
: Mutators to wrap. The operations are run in order given to mutators. The constructed object gets a *clone* of this argument. The $mutators field will reflect this value.

Method prime()**:** See MiesOperator method. Primes both this operator, as well as the wrapped operators given to mutator and mutator_not during construction.

Usage:

```
MutatorSequential$prime(param_set)
```

Arguments:

param_set (ParamSet)
Passed to MiesOperator$prime().

Returns: invisible self.

Method clone()**:** The objects of this class are cloneable with this method.

Usage:

```
MutatorSequential$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other mutators: MutatorDiscrete, MutatorNumeric, Mutator, OperatorCombination, dict_mutators_cmpmaybe, dict_mutators_erase, dict_mutators_gauss, dict_mutators_maybe, dict_mutators_null, dict_mutators_proxy, dict_mutators_unif

Other mutator wrappers: OperatorCombination, dict_mutators_cmpmaybe, dict_mutators_maybe, dict_mutators_proxy

Examples

```
set.seed(1)

# dataset:
#  - x1 is mutated around +- 10
#  - x2 influences sdev of mutation of x1
ds = data.frame(x1 = 0, x2 = c(.01, 0.1, 1))
p = ps(x1 = p_dbl(-10, 10), x2 = p_dbl(0, 10))

# operator that only mutates x1, with sdev given by x2
gauss_x1 = mut("combine",
  operators = list(
    x1 = mut("gauss", sdev_is_relative = FALSE),
    x2 = mut("null")
  ),
  adaptions = list(x1.sdev = function(x) x$x2)
)

gauss_x1$prime(p)
gauss_x1$operate(ds)  # see how x1[1] changes little, x1[3] changes a lot

# operator that mutates x1  with sdev given by x2, as well as x2. However,
# the value that x2 takes after mutation does not influence the value that
# the mutator of x1 "sees" -- although x2 is mutated to extreme values,
# mutation of x1 happens as in `gauss_x1`.
gauss_x1_x2 = mut("combine",
  operators = list(
```

```
    x1 = mut("gauss", sdev_is_relative = FALSE),
    x2 = mut("gauss", sdev = 100)
  ),
  adaptions = list(x1.sdev = function(x) x$x2)
)

gauss_x1_x2$prime(p)
gauss_x1_x2$operate(ds)  # see how x1 changes in similar ways to above

# operator that mutates sequentially: first x2, and then x1 with sdev given
# by x2. The value that x2 takes after mutation *does* influence the value
# that the mutator of x1 "sees": x1 is mutated either to a large degree,
# or not at all.

gauss_x2_then_x1 = mut("sequential", list(
    mut("combine",
      operators = list(
        x1 = mut("null"),
        x2 = mut("gauss", sdev = 100)
      )
    ),
    mut("combine",
      operators = list(
        x1 = mut("gauss", sdev_is_relative = FALSE),
        x2 = mut("null")
      ),
      adaptions = list(x1.sdev = function(x) x$x2)
    )
))

gauss_x2_then_x1$prime(p)
gauss_x2_then_x1$operate(ds)
```

---

dict_mutators_unif *Uniform Discrete Mutator*

Description

Discrete components are mutated by sampling from a uniform distribution, either from all possible values of each component, or from all values except the original value.

Since the information loss is very high, this should in most cases be combined with MutatorCmpMaybe.

Configuration Parameters

- `can_mutate_to_same` :: `logical(1)`
  Whether to sample from entire range of each parameter (`TRUE`) or from all values except the current value (`FALSE`). Initialized to `TRUE`.

Supported Operand Types

Supported Param classes are: ParamLgl, ParamFct

Dictionary

This Mutator can be created with the short access form mut() (muts() to get a list), or through the the dictionary dict_mutators in the following way:

```
# preferred:
mut("unif")
muts("unif")  # takes vector IDs, returns list of Mutators

# long form:
dict_mutators$get("unif")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Mutator -> miesmuschel::MutatorDiscrete -> MutatorDiscreteUniform

Methods

Public methods:

- MutatorDiscreteUniform$new()
- MutatorDiscreteUniform$clone()

Method `new()`**:** Initialize the `MutatorDiscreteUniform` object.

Usage:

```
MutatorDiscreteUniform$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MutatorDiscreteUniform$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other mutators: MutatorDiscrete, MutatorNumeric, Mutator, OperatorCombination, dict_mutators_cmpmaybe, dict_mutators_erase, dict_mutators_gauss, dict_mutators_maybe, dict_mutators_null, dict_mutators_proxy, dict_mutators_sequential

Examples

```
set.seed(1)
mdu = mut("unif")
p = ps(x = p_lgl(), y = p_fct(c("a", "b", "c")))
data = data.frame(x = rep(TRUE, 5), y = rep("a", 5),
  stringsAsFactors = FALSE)  # necessary for R <= 3.6

mdu$prime(p)
mdu$operate(data)

mdu$param_set$values$can_mutate_to_same = FALSE
mdu$operate(data)
```

---

dict_recombinators   *Dictionary of Recombinators*

Description

Dictionary of Recombinators

Usage

```
dict_recombinators
```

Format

An object of class `DictionaryRecombinator` (inherits from `DictionaryEx`, `Dictionary`, `R6`) of length 15.

Methods

Methods inherited from `Dictionary`, as well as:

- `help(key, help_type)`
  `(character(1), character(1))`
  Displays help for the dictionary entry key. `help_type` is one of `"text"`, `"html"`, `"pdf"` and given as the `help_type` argument of R's `help()`.

See Also

Other dictionaries: `dict_filtors`, `dict_mutators`, `dict_scalors`, `dict_selectors`, `mut()`

dict_recombinators_cmpmaybe

Recombinator Choosing Action Component-Wise Independently

Description

Recombinator that chooses which operation to perform probabilistically and independently for each component. The Recombinator wraps two other Recombinators given during construction, and both of these operators are run. The ultimate result is sampled from the results of these operations independently for each individuum and component: with probability p (configuration parameter), the result from the Recombinator given to the recombinator construction argument is used, and with probability p - 1 the one given to recombinator_not is used.

The values of $n_indivs_in and $n_indivs_out is set to the corresponding values of the wrapped Recombinators. Both recombinator and recombinator_not must currently have the same respective $n_indivs_in and $n_indivs_out values.

Configuration Parameters

This operator has the configuration parameters of the Recombinators that it wraps: The configuration parameters of the operator given to the recombinator construction argument are prefixed with "cmpmaybe.", the configuration parameters of the operator given to the recombinator_not construction argument are prefixed with "cmpmaybe_not.".

Additional configuration parameters:

- p :: numeric(1)
  Probability per component with which to use the result of applying the operator given to the recombinator construction argument. Must be set by the user.

Supported Operand Types

Supported Param classes are the set intersection of supported classes of recombinator and recombinator_not.

Dictionary

This Recombinator can be created with the short access form rec() (recs() to get a list), or through the the dictionary dict_recombinators in the following way:

```
# preferred:
rec("cmpmaybe", <recombinator> [, <recombinator_not>])
recs("cmpmaybe", <recombinator> [, <recombinator_not>])  # takes vector IDs, returns list of Recombinat

# long form:
dict_recombinators$get("cmpmaybe", <recombinator> [, <recombinator_not>])
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Recombinator -> RecombinatorCmpMaybe

Active bindings

`recombinator` (Recombinator)
: Recombinator being wrapped. This operator gets run with probability `p` (configuration parameter).

`recombinator_not` (Recombinator)
: Alternative Recombinator being wrapped. This operator gets run with probability `1 - p` (configuration parameter).

Methods

Public methods:

- `RecombinatorCmpMaybe$new()`
- `RecombinatorCmpMaybe$prime()`
- `RecombinatorCmpMaybe$clone()`

Method `new()`**:** Initialize the `RecombinatorCmpMaybe` object.

Usage:

```
RecombinatorCmpMaybe$new(recombinator, recombinator_not = NULL)
```

Arguments:

`recombinator` (Recombinator)
: Recombinator to wrap. Component-wise results of this operator are used with probability `p` (Configuration parameter).
The constructed object gets a *clone* of this argument. The `$recombinator` field will reflect this value.

`recombinator_not` (Recombinator)
: Another Recombinator to wrap. Results from this operator are used when `recombinator` is not chosen. By default, this is RecombinatorNull, i.e. no operation, with both `n_indivs_in` and `n_indivs_out` set to match `recombinator`. This does not work when `recombinator` has `n_indivs_in` < `n_indivs_out`, in which case this argument must be set explicitly.
With this default, the `RecombinatorCmpMaybe` object applies the `recombinator` operation with probability `p`, and no operation at all otherwise.
The constructed object gets a *clone* of this argument. The `$recombinator_not` field will reflect this value.

Method `prime()`**:** See MiesOperator method. Primes both this operator, as well as the wrapped operators given to `recombinator` and `recombinator_not` during construction.

Usage:

```
RecombinatorCmpMaybe$prime(param_set)
```

Arguments:

`param_set` (ParamSet)
: Passed to `MiesOperator$prime()`.

Returns: invisible `self`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
RecombinatorCmpMaybe$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other recombinators: `OperatorCombination`, `RecombinatorPair`, `Recombinator`, `dict_recombinators_convex`, `dict_recombinators_cvxpair`, `dict_recombinators_maybe`, `dict_recombinators_null`, `dict_recombinators_prox`
`dict_recombinators_sbx`, `dict_recombinators_sequential`, `dict_recombinators_swap`, `dict_recombinators_xona`
`dict_recombinators_xounif`

Other recombinator wrappers: `OperatorCombination`, `dict_recombinators_maybe`, `dict_recombinators_proxy`, `dict_recombinators_sequential`

Examples

```
set.seed(1)
rm = rec("cmpmaybe", rec("swap"), p = 0.5)
p = ps(x = p_int(1, 8), y = p_dbl(1, 8), z = p_lgl())
data = data.frame(x = 1:8, y = 1:8, z = rep(TRUE, 4))

rm$prime(p)
rm$operate(data)

rm$param_set$values$p = 0.3
rm$operate(data)

# equivalent to rec("cmpmaybe", rec("swap", keep_complement = FALSE), p = 0.7)
rm2 = rec("cmpmaybe",
  recombinator = rec("null", 2, 1),
  recombinator_not = rec("swap", keep_complement = FALSE),
  p = 0.3
)

rm2$prime(p)
rm2$operate(data)
```

---

dict_recombinators_convex

Convex Combination Recombinator

Description

Numeric Values between various individuals are recombined via component-wise convex combination (or weighted mean). The number of individuals over which the convex combination is taken must be determined during construction as `n_indivs_in`.

The number of output individuals is always 1, i.e. `n_indivs_in` are used to create one output value. When using this recombinator in a typical EA setting, e.g. with `mies_generate_offspring`, it is therefore recommended to use a parent-selector where the expected quality of selected parents does not depend on the number of parents selected when n_indivs_in is large: sel("tournament") is preferred to sel("best").

Configuration Parameters

- lambda :: numeric | matrix
  Combination weights; these are normalized to sum to 1 internally. Must either be a vector of length n_indivs_in, or a matrix with n_indivs_in rows and as many columns as there are components in the values being operated on. Must be non-negative, at least one value per column must be greater than zero, but it is not necessary that they sum to 1.
  Initialized to rep(1, n_indivs_in), i.e. equal weights to all individuals being operated on.

Supported Operand Types

Supported Param classes are: ParamDbl

Dictionary

This Recombinator can be created with the short access form rec() (recs() to get a list), or through the the dictionary dict_recombinators in the following way:

```
# preferred:
rec("convex")
recs("convex")  # takes vector IDs, returns list of Recombinators

# long form:
dict_recombinators$get("convex")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Recombinator -> RecombinatorConvex

Methods

Public methods:

- RecombinatorConvex$new()
- RecombinatorConvex$clone()

Method new(): Initialize the RecombinatorConvex object.

Usage:

```
RecombinatorConvex$new(n_indivs_in = 2)
```

Arguments:

n_indivs_in (integer(1))
  Number of individuals to consider at the same time. When operating, the number of input individuals must be divisible by this number. Default 2.
  The $n_indivs_in field will reflect this value.

Method clone(): The objects of this class are cloneable with this method.

Usage:

`RecombinatorConvex$clone(deep = FALSE)`

Arguments:

deep Whether to make a deep clone.

See Also

Other recombinators: OperatorCombination, RecombinatorPair, Recombinator, dict_recombinators_cmpmaybe, dict_recombinators_cvxpair, dict_recombinators_maybe, dict_recombinators_null, dict_recombinators_proxy, dict_recombinators_sbx, dict_recombinators_sequential, dict_recombinators_swap, dict_recombinators_xonary, dict_recombinators_xounif

Examples

```
set.seed(1)
rcvx = rec("convex", n_indivs_in = 3)
p = ps(x = p_dbl(-5, 5), y = p_dbl(-5, 5), z = p_dbl(-5, 5))
data = data.frame(x = 0:5, y = 0:5, z = 0:5)

rcvx$prime(p)
rcvx$operate(data)  # mean of groups of 3

rcvx = rec("convex", 3, lambda = c(0, 1, 2))$prime(p)
rcvx$operate(data)  # for groups of 3, take 1/3 of 2nd and 2/3 of 3rd row

lambda = matrix(c(0, 1, 2, 1, 1, 1, 1, 0, 0), ncol = 3)
lambda

rcvx = rec("convex", 3, lambda = lambda)$prime(p)
rcvx$operate(data)  # componentwise different operation
```

`dict_recombinators_cvxpair`

Convex Combination Recombinator for Pairs

Description

Numeric Values between various individuals are recombined via component-wise convex combination (or weighted mean). Exactly two individuals are being recombined, and the `lambda` configuration parameter determines the relative weight of the first individual in each pair for the first result, and the relative weight of the second indivudual for the complement, if initialized with `keep_complement` set to `TRUE`.

Configuration Parameters

- `lambda` :: `numeric`
  Combination weight. If `keep_complement` is `TRUE`, then two individuals are returned for each pair of input individuals: one corresponding to `lambda * <1st individual> + (1-lambda) * <2nd individual>`, and one corresponding to `(1-lambda) * <1st individual> + lambda * <2nd individual>` (i.e. the complement). Otherwise, only the first of these two is generated. Must either be a scalar, or a vector with length equal to the number of components in the values being operated on. Must be between 0 and 1.
  Initialized to 0.5.

Supported Operand Types

Supported `Param` classes are: `ParamDbl`

Dictionary

This `Recombinator` can be created with the short access form `rec()` (`recs()` to get a list), or through the the dictionary `dict_recombinators` in the following way:

```
# preferred:
rec("convex")
recs("convex")  # takes vector IDs, returns list of Recombinators

# long form:
dict_recombinators$get("convex")
```

Super classes

`miesmuschel::MiesOperator` -> `miesmuschel::Recombinator` -> `miesmuschel::RecombinatorPair` -> `RecombinatorConvexPair`

Methods

Public methods:

- `RecombinatorConvexPair$new()`
- `RecombinatorConvexPair$clone()`

Method `new()`**:** Initialize the `RecombinatorConvexPair` object.

Usage:

```
RecombinatorConvexPair$new(keep_complement = TRUE)
```

Arguments:

`keep_complement (logical(1))`
Whether the operation should keep both resulting individuals (`TRUE`), or only the first and discard the complement (`FALSE`). Default `TRUE`. The `$keep_complement` field will reflect this value.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
RecombinatorConvexPair$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other recombinators: OperatorCombination, RecombinatorPair, Recombinator, dict_recombinators_cmpmaybe, dict_recombinators_convex, dict_recombinators_maybe, dict_recombinators_null, dict_recombinators_proxy, dict_recombinators_sbx, dict_recombinators_sequential, dict_recombinators_swap, dict_recombinators_xona dict_recombinators_xounif

Examples

```
set.seed(1)
rcvx = rec("cvxpair")
p = ps(x = p_dbl(-5, 5), y = p_dbl(-5, 5), z = p_dbl(-5, 5))
data = data.frame(x = 0:5, y = 0:5, z = 0:5)

rcvx$prime(p)
rcvx$operate(data)  # mean of groups of 2
# with the default value of lambda = 0.5, the default of
# keep_complement = TRUE means that pairs of equal values are generated;
# consider setting keep_complement = FALSE int that case.

rcvx$param_set$values$lambda = 0.1
rcvx$operate(data)
```

---

dict_recombinators_maybe

Recombinator Choosing Action Probabilistically

Description

Recombinator that chooses which operation to perform probabilistically. The Recombinator wraps two other Recombinators given during construction, and for each group of $n_indivs_in individuals, the operation to perform is sampled: with probability p (configuration parameter), the Recombinator given to the recombinator construction argument is applied, and with probability p - 1 the one given to recombinator_not is applied.

The values of $n_indivs_in and $n_indivs_out is set to the corresponding values of the wrapped Recombinators. Both recombinator and recombinator_not must currently have the same respective $n_indivs_in and $n_indivs_out values.

Configuration Parameters

This operator has the configuration parameters of the Recombinators that it wraps: The configuration parameters of the operator given to the recombinator construction argument are prefixed with "maybe.", the configuration parameters of the operator given to the recombinator_not construction argument are prefixed with "maybe_not.".

Additional configuration parameters:

- p :: numeric(1)
  Probability per group of n_indivs_in individuals with which to apply the operator given to the recombinator construction argument. Must be set by the user.

Supported Operand Types

Supported Param classes are the set intersection of supported classes of recombinator and recombinator_not.

Dictionary

This Recombinator can be created with the short access form rec() (recs() to get a list), or through the the dictionary dict_recombinators in the following way:

```
# preferred:
rec("maybe", <recombinator> [, <recombinator_not>])
recs("maybe", <recombinator> [, <recombinator_not>])  # takes vector IDs, returns list of Recombinators

# long form:
dict_recombinators$get("maybe", <recombinator> [, <recombinator_not>])
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Recombinator -> RecombinatorMaybe

Active bindings

recombinator (Recombinator)
: Recombinator being wrapped. This operator gets run with probability p (configuration parameter).

recombinator_not (Recombinator)
: Alternative Recombinator being wrapped. This operator gets run with probability 1 - p (configuration parameter).

Methods

Public methods:

- RecombinatorMaybe$new()
- RecombinatorMaybe$prime()
- RecombinatorMaybe$clone()

Method new()**:** Initialize the RecombinatorMaybe object.

Usage:

`RecombinatorMaybe$new(recombinator, recombinator_not = NULL)`

Arguments:

`recombinator` (`Recombinator`)
: `Recombinator` to wrap. This operator gets run with probability `p` (Configuration parameter).
The constructed object gets a *clone* of this argument. The `$recombinator` field will reflect this value.

`recombinator_not` (`Recombinator`)
: Another `Recombinator` to wrap. This operator runs when `recombinator` is not chosen. By default, this is `RecombinatorNull`, i.e. no operation, with both `n_indivs_in` and `n_indivs_out` set to match `recombinator`. This does not work when `recombinator` has `n_indivs_in` < `n_indivs_out`, in which case this argument must be set explicitly.
With this default, the `RecombinatorMaybe` object applies the `recombinator` operation with probability `p`, and no operation at all otherwise.
The constructed object gets a *clone* of this argument. The `$recombinator_not` field will reflect this value.

Method `prime()`: See `MiesOperator` method. Primes both this operator, as well as the wrapped operators given to `recombinator` and `recombinator_not` during construction.

Usage:

`RecombinatorMaybe$prime(param_set)`

Arguments:

`param_set` (`ParamSet`)
: Passed to `MiesOperator$prime()`.

Returns: invisible `self`.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

`RecombinatorMaybe$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

See Also

Other recombinators: `OperatorCombination`, `RecombinatorPair`, `Recombinator`, `dict_recombinators_cmpmaybe`, `dict_recombinators_convex`, `dict_recombinators_cvxpair`, `dict_recombinators_null`, `dict_recombinators_prox`, `dict_recombinators_sbx`, `dict_recombinators_sequential`, `dict_recombinators_swap`, `dict_recombinators_xona`, `dict_recombinators_xounif`

Other recombinator wrappers: `OperatorCombination`, `dict_recombinators_cmpmaybe`, `dict_recombinators_proxy`, `dict_recombinators_sequential`

Examples

```
set.seed(1)
rm = rec("maybe", rec("xounif", p = 1), p = 0.5)
p = ps(x = p_int(1, 8), y = p_dbl(1, 8), z = p_lgl())
data = data.frame(x = 1:8, y = 1:8, z = rep(TRUE, 4))

rm$prime(p)
rm$operate(data)

rm$param_set$values$p = 0.3
rm$operate(data)

rm2 = rec("maybe",
  recombinator = rec("xounif", p = 1),
  recombinator_not = rec("xounif", p = 0.5),
  p = 0.5
)

rm2$prime(p)
rm2$operate(data)
```

`dict_recombinators_null` *Null-Recombinator*

Description

Null-recombinator that does not perform any operation on its input. Useful in particular with operator-wrappers such as `RecombinatorMaybe` or `RecombinatorCombination`.

`n_indivs_in` and `n_indivs_out` can be set during construction, where `n_indivs_out` must be less or equal `n_indivs_in`. If it is strictly less, then the operation returns only the first `n_indivs_out` individuals out of each `n_indivs_in` sized group.

Configuration Parameters

This operator has no configuration parameters.

Supported Operand Types

Supported `Param` classes are: `ParamLgl`, `ParamInt`, `ParamDbl`, `ParamFct`

Dictionary

This `Recombinator` can be created with the short access form `rec()` (`recs()` to get a list), or through the the dictionary `dict_recombinators` in the following way:

```
# preferred:
rec("null")
recs("null")  # takes vector IDs, returns list of Recombinators

# long form:
dict_recombinators$get("null")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Recombinator -> RecombinatorNull

Methods

Public methods:

- RecombinatorNull$new()
- RecombinatorNull$clone()

Method new()**:** Initialize base class components of the Recombinator.

Usage:

```
RecombinatorNull$new(n_indivs_in = 1, n_indivs_out = n_indivs_in)
```

Arguments:

n_indivs_in (integer(1))
: Number of individuals to consider at the same time. When operating, the number of input individuals must be divisible by this number. Setting this number to a number unequal 1 is mostly useful when incorporating this operator in wrappers such as RecombinatorMaybe or RecombinatorCombination. Default 1.
The $n_indivs_in field will reflect this value.

n_indivs_out (integer(1))
: Number of individuals that result for each n_indivs_in lines of input. Must be at most n_indivs_in. If this is less than n_indivs_in, then only the first n_indivs_out individuals out of each n_indivs_in sized group are returned by an operation. Setting this number to a number unequal 1 is mostly useful when incorporating this operator in wrappers such as RecombinatorMaybe or RecombinatorCombination. Default equal to n_indivs_in.
The $n_indivs_out field will reflect this value.

Method clone()**:** The objects of this class are cloneable with this method.

Usage:

```
RecombinatorNull$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other recombinators: OperatorCombination, RecombinatorPair, Recombinator, dict_recombinators_cmpmaybe, dict_recombinators_convex, dict_recombinators_cvxpair, dict_recombinators_maybe, dict_recombinators_pro dict_recombinators_sbx, dict_recombinators_sequential, dict_recombinators_swap, dict_recombinators_xona dict_recombinators_xounif

Examples

```
rn = rec("null")
p = ps(x = p_int(-5, 5), y = p_dbl(-5, 5), z = p_lgl())
data = data.frame(x = 1:4, y = 0:3, z = rep(TRUE, 4))

rn$prime(p)
rn$operate(data)

rn_half = rec("null", n_indivs_in = 2, n_indivs_out = 1)
rn_half$prime(p)
rn_half$operate(data)
```

---

dict_recombinators_proxy

Proxy-Recombinator that Recombines According to its Configuration parameter

---

Description

Recombinator that performs the operation in its `operation` configuration parameter. This is useful, e.g., to make `OptimizerMies`'s recombination operation fully parametrizable.

Configuration Parameters

- `operation` :: `Recombinator`
  Operation to perform. Must be set by the user. This is primed when `$prime()` of `RecombinatorProxy` is called, and also when `$operate()` is called, to make changing the operation as part of self-adaption possible. However, if the same operation gets used inside multiple `RecombinatorProxy` objects, then it is recommended to `$clone(deep = TRUE)` the object before assigning them to `operation` to avoid frequent re-priming.

Supported Operand Types

Supported `Param` classes are: `ParamLgl`, `ParamInt`, `ParamDbl`, `ParamFct`

Dictionary

This `Recombinator` can be created with the short access form `rec()` (`recs()` to get a list), or through the the dictionary `dict_recombinators` in the following way:

```
# preferred:
rec("proxy")
recs("proxy")  # takes vector IDs, returns list of Recombinators

# long form:
dict_recombinators$get("proxy")
```

Super classes

`miesmuschel::MiesOperator` -> `miesmuschel::Recombinator` -> `RecombinatorProxy`

Methods

Public methods:

- `RecombinatorProxy$new()`
- `RecombinatorProxy$prime()`
- `RecombinatorProxy$clone()`

Method `new()`: Initialize the `RecombinatorProxy` object.

Usage:

```
RecombinatorProxy$new(n_indivs_in = 2, n_indivs_out = n_indivs_in)
```

Arguments:

`n_indivs_in` (`integer(1)`)
: Number of individuals to consider at the same time. When operating, the number of input individuals must be divisible by this number. Furthermore, the `Recombinator` assigned to the `operation` configuration parameter must have an `n_indivs_in` that is a divisor of this number. Default 2.
The `$n_indivs_in` field will reflect this value.

`n_indivs_out` (`integer(1)`)
: Number of individuals that result for each `n_indivs_in` lines of input. Must be at most `n_indivs_in`. The ratio of `$n_indivs_in` to `$n_indivs_out` of the `Recombinator` assigned to the `operation` configuration parameter must be the same as `n_indivs_in` to `n_indivs_out` of this object. Default equal to `n_indivs_in`.
The `$n_indivs_out` field will reflect this value.

Method `prime()`: See `MiesOperator` method. Primes both this operator, as well as the operator given to the `operation` configuration parameter. Note that this modifies the `$param_set$values$operation` object.

Usage:

```
RecombinatorProxy$prime(param_set)
```

Arguments:

`param_set` (`ParamSet`)
: Passed to `MiesOperator$prime()`.

Returns: invisible `self`.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
RecombinatorProxy$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other recombinators: OperatorCombination, RecombinatorPair, Recombinator, dict_recombinators_cmpmaybe, dict_recombinators_convex, dict_recombinators_cvxpair, dict_recombinators_maybe, dict_recombinators_nul dict_recombinators_sbx, dict_recombinators_sequential, dict_recombinators_swap, dict_recombinators_xona dict_recombinators_xounif

Other recombinator wrappers: OperatorCombination, dict_recombinators_cmpmaybe, dict_recombinators_maybe, dict_recombinators_sequential

Examples

```
set.seed(1)
rp = rec("proxy", operation = rec("xounif"))
p = ps(x = p_int(-5, 5), y = p_dbl(-5, 5), z = p_lgl())
data = data.frame(x = 1:4, y = 0:3, z = rep(TRUE, 4))

rp$prime(p)
rp$operate(data)  # default operation: null

rp$param_set$values$operation = rec("xounif", p = 0.5)
rp$operate(data)
```

---

dict_recombinators_sbx — *Simulated Binary Crossover Recombinator*

Description

Numeric Values between two individuals are recombined via component-wise independent simulated binary crossover. See Deb (1995) for more details.

This operator is applied to all components; It is common to apply the operator to only some randomly chosen components, in which case the rec("cmpmaybe") operator should be used; see examples.

Configuration Parameters

- n :: numeric
  Non-negative distribution index of the polynomial distribution for each component. Generally spoken, the higher n, the higher the probability of creating near parent values. This may either be a scalar in which case it is applied to all input components, or a vector, in which case it must have the length of the input components and applies to components in order in which they appear in the priming ParamSet. Initialized to 1.

Supported Operand Types

Supported Param classes are: ParamInt, ParamDbl

Dictionary

This Recombinator can be created with the short access form rec() (recs() to get a list), or through the the dictionary dict_recombinators in the following way:

```
# preferred:
rec("sbx")
recs("sbx")  # takes vector IDs, returns list of Recombinators

# long form:
dict_recombinators$get("sbx")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Recombinator -> miesmuschel::RecombinatorPair -> RecombinatorSimulatedBinaryCrossover

Methods

Public methods:

- RecombinatorSimulatedBinaryCrossover$new()
- RecombinatorSimulatedBinaryCrossover$clone()

Method `new()`**:** Initialize the `RecombinatorSimulatedBinaryCrossover` object.

Usage:

```
RecombinatorSimulatedBinaryCrossover$new(keep_complement = TRUE)
```

Arguments:

`keep_complement` (`logical(1)`)
Whether the operation should keep both resulting individuals (`TRUE`), or only the first and discard the complement (`FALSE`). Default `TRUE`. The `$keep_complement` field will reflect this value.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
RecombinatorSimulatedBinaryCrossover$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Deb, Kalyanmoy, Agrawal, Bhushan R, others (1995). "Simulated binary crossover for continuous search space." *Complex systems*, **9**(2), 115–148.

See Also

Other recombinators: OperatorCombination, RecombinatorPair, Recombinator, dict_recombinators_cmpmaybe, dict_recombinators_convex, dict_recombinators_cvxpair, dict_recombinators_maybe, dict_recombinators_nul dict_recombinators_proxy, dict_recombinators_sequential, dict_recombinators_swap, dict_recombinators_xonary, dict_recombinators_xounif

Examples

```
set.seed(1)
rsbx = rec("cmpmaybe", rec("sbx"), p = 0.5)
p = ps(x = p_dbl(-5, 5), y = p_dbl(-5, 5), z = p_dbl(-5, 5))
data = data.frame(x = 0:5, y = 0:5, z = 0:5)

rsbx$prime(p)
rsbx$operate(data)

rsbx = rec("sbx", n = c(0.5, 1, 10))
rsbx$prime(p)
rsbx$operate(data)
```

---

dict_recombinators_sequential

Run Multiple Recombinator Operations in Sequence

Description

Recombinator that wraps multiple other Recombinators given during construction and uses them for mutation in sequence.

When subsequent Recombinators have mismatching n_indivs_out / n_indivs_in, then RecombinatorSequential tries to match them by running them multiple times. If e.g. recombinators[[1]]$n_indivs_out is 2 and recombinators[[2]]$n_indivs_in is 1, then recombinators[[2]] is run twice, once for each output of recombinators[[1]].

When the allow_lcm_packing argument is FALSE, then an error is given if neither n_indivs_out of a Recombinator divides n_indivs_in of the following Recombinator, nor n_indivs_in of the latter divides n_indivs_out of the former even when considering that the former is run multiple times. If allow_lcm_packing is TRUE, then both recombinators are run multiple times, according to the lowest common multiple ("lcm") of the two.

However, allow_lcm_packing can lead to very large values of n_indivs_in / n_indivs_out, so it may instead be preferred to add RecombinatorNull objects with fitting n_indivs_in / n_indivs_out values to match subsequent recombinators.

Configuration Parameters

This operator has the configuration parameters of the Recombinators that it wraps: The configuration parameters of the operator given to the recombinators construction argument are prefixed with "recombinator_1", "recombinator_2", ... up to "recombinator_#", where # is length(recombinators).

Additional configuration parameters:

- shuffle_between :: logical(1)
  Whether to reorder values between invocations of recombinators. Initialized to TRUE.

Supported Operand Types

Supported Param classes are the set intersection of supported classes of the Recombinators given in recombinators.

Dictionary

This Recombinator can be created with the short access form rec() (recs() to get a list), or through the the dictionary dict_recombinators in the following way:

```
# preferred:
rec("sequential", <recombinators>)
recs("sequential", <recombinators>)  # takes vector IDs, returns list of Recombinators

# long form:
dict_recombinators$get("sequential", <recombinators>)
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Recombinator -> RecombinatorSequential

Active bindings

recombinators (list of Recombinator)
: Recombinators being wrapped. These operators get run sequentially in order.

allow_lcm_packing (logical(1))
: Whether to allow lowest common multiple packing.

Methods

Public methods:

- RecombinatorSequential$new()
- RecombinatorSequential$prime()
- RecombinatorSequential$clone()

Method new()**:** Initialize the RecombinatorSequential object.

Usage:

```
RecombinatorSequential$new(recombinators, allow_lcm_packing = FALSE)
```

Arguments:

recombinators (list of Recombinator)
: Recombinators to wrap. The operations are run in order given to recombinators. The constructed object gets a *clone* of this argument. The $recombinators field will reflect this value.

allow_lcm_packing (logical(1))
: Whether to allow lowest common multiple packing. Default FALSE. The $allow_lcm_packing field will reflect this value.

Method prime()**:** See MiesOperator method. Primes both this operator, as well as the wrapped operators given to recombinator and recombinator_not during construction.

Usage:

```
RecombinatorSequential$prime(param_set)
```

Arguments:

`param_set` (ParamSet)
Passed to `MiesOperator$prime()`.

Returns: invisible `self`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
RecombinatorSequential$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other recombinators: `OperatorCombination`, `RecombinatorPair`, `Recombinator`, `dict_recombinators_cmpmaybe`, `dict_recombinators_convex`, `dict_recombinators_cvxpair`, `dict_recombinators_maybe`, `dict_recombinators_nu`, `dict_recombinators_proxy`, `dict_recombinators_sbx`, `dict_recombinators_swap`, `dict_recombinators_xonary`, `dict_recombinators_xounif`

Other recombinator wrappers: `OperatorCombination`, `dict_recombinators_cmpmaybe`, `dict_recombinators_maybe`, `dict_recombinators_proxy`

Examples

```
set.seed(1)

ds = data.frame(a = c(0, 1), b = c(0, 1))
p = ps(a = p_dbl(0, 1), b = p_dbl(0, 1))


convex = rec("cvxpair", lambda = 0.7)
swap = rec("swap")

convex_then_swap = rec("sequential", list(convex, swap))

ds

convex$prime(p)$operate(ds)

swap$prime(p)$operate(ds)

convex_then_swap$prime(p)$operate(ds)
```

dict_recombinators_swap
Swap Recombinator

Description

Values between two individuals are exchanged. This is relatively useless as an operator by itself, but is used in combination with RecombinatorCmpMaybe to get a recombinator that is crossing over individuals uniformly at random. Because this is such a frequently-used operation, the RecombinatorCrossoverUniform pseudo-class exists as a shortcut.

Configuration Parameters

None.

Supported Operand Types

Supported Param classes are: ParamLgl, ParamInt, ParamDbl, ParamFct

Dictionary

This Recombinator can be created with the short access form rec() (recs() to get a list), or through the the dictionary dict_recombinators in the following way:

```
# preferred:
rec("swap")
recs("swap")  # takes vector IDs, returns list of Recombinators

# long form:
dict_recombinators$get("swap")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Recombinator -> miesmuschel::RecombinatorPair -> RecombinatorSwap

Methods

Public methods:

- RecombinatorSwap$new()
- RecombinatorSwap$clone()

Method `new()`**:** Initialize the `RecombinatorCrossoverSwap` object.

Usage:

```
RecombinatorSwap$new(keep_complement = TRUE)
```

Arguments:

`keep_complement` (`logical(1)`)
Whether the operation should keep both resulting individuals (`TRUE`), or only the first and discard the complement (`FALSE`). Default `TRUE`. The `$keep_complement` field will reflect this value.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
RecombinatorSwap$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other recombinators: `OperatorCombination`, `RecombinatorPair`, `Recombinator`, `dict_recombinators_cmpmaybe`, `dict_recombinators_convex`, `dict_recombinators_cvxpair`, `dict_recombinators_maybe`, `dict_recombinators_nul`
`dict_recombinators_proxy`, `dict_recombinators_sbx`, `dict_recombinators_sequential`, `dict_recombinators_xor`
`dict_recombinators_xounif`

Examples

```
set.seed(1)
rs = rec("swap")
p = ps(x = p_int(-5, 5), y = p_dbl(-5, 5), z = p_dbl(-5, 5))
data = data.frame(x = 0:5, y = 0:5, z = 0:5)

rs$prime(p)
rs$operate(data)

rx = rec("cmpmaybe", rec("swap"), p = 0.5)  # the same as 'rec("xounif")'
rx$prime(p)
rx$operate(data)
```

`dict_recombinators_xonary` *N-ary Crossover Recombinator*

Description

Values are chosen componentwise independently at random from multiple individuals. The number of individuals must be determined during construction as `n_indivs_in`.

The number of output individuals is always 1, i.e. `n_indivs_in` are used to create one output value. When using this recombinator in a typical EA setting, e.g. with `mies_generate_offspring`, it is therefore recommended to use a parent-selector where the expected quality of selected parents does not depend on the number of parents selected when `n_indivs_in` is large: `sel("tournament")` is preferred to `sel("best")`.

Configuration Parameters

- p :: numeric | matrix
  Sampling weights these are normalized to sum to 1 internally. Must either be a vector of length n_indivs_in, or a matrix with n_indivs_in rows and as many columns as there are components in the values being operated on. Must be non-negative, at least one value per column must be greater than zero, but it is not necessary that they sum to 1.
  Initialized to rep(1, n_indivs_in), i.e. uniform sampling from all individuals being operated on.

Supported Operand Types

Supported Param classes are: ParamDbl

Dictionary

This Recombinator can be created with the short access form rec() (recs() to get a list), or through the the dictionary dict_recombinators in the following way:

```
# preferred:
rec("convex")
recs("convex")  # takes vector IDs, returns list of Recombinators

# long form:
dict_recombinators$get("convex")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Recombinator -> RecombinatorCrossoverNary

Methods

Public methods:

- RecombinatorCrossoverNary$new()
- RecombinatorCrossoverNary$clone()

Method new()**:** Initialize the RecombinatorConvex object.

Usage:

```
RecombinatorCrossoverNary$new(n_indivs_in = 2)
```

Arguments:

n_indivs_in (integer(1))
: Number of individuals to consider at the same time. When operating, the number of input individuals must be divisible by this number. Default 2.
  The $n_indivs_in field will reflect this value.

Method clone()**:** The objects of this class are cloneable with this method.

Usage:

```
RecombinatorCrossoverNary$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other recombinators: `OperatorCombination`, `RecombinatorPair`, `Recombinator`, `dict_recombinators_cmpmaybe`, `dict_recombinators_convex`, `dict_recombinators_cvxpair`, `dict_recombinators_maybe`, `dict_recombinators_nul` `dict_recombinators_proxy`, `dict_recombinators_sbx`, `dict_recombinators_sequential`, `dict_recombinators_swa` `dict_recombinators_xounif`

Examples

```
set.seed(1)
rxon = rec("xonary", n_indivs_in = 3)
p = ps(x = p_dbl(-5, 5), y = p_dbl(-5, 5), z = p_dbl(-5, 5))
data = data.frame(x = 0:5, y = 0:5, z = 0:5)

rxon$prime(p)
rxon$operate(data)  # uniform sampling from groups of 3

rxon = rec("xonary", 3, p = c(0, 1, 2))$prime(p)
# for groups of 3, take with probability 1/3 from 2nd and with probability 2/3 from 3rd row
rxon$operate(data)

pmat = matrix(c(0, 1, 2, 1, 1, 1, 1, 0, 0), ncol = 3)
pmat

rxon = rec("xonary", 3, p = pmat)$prime(p)
rxon$operate(data)  # componentwise different operation
```

---

dict_recombinators_xounif

Crossover Recombinator

Description

Values between two individuals are exchanged with component-wise independent probability.

This is a pseudo-class: It does not create a single R6-object of a class; instead, it creates the object `rec("cmpmaybe", rec("swap"), p = 0.5)`, making use of the `RecombinatorCmpMaybe` and `RecombinatorSwap` operators.

Usage

```
RecombinatorCrossoverUniform(keep_complement = TRUE)
```

Arguments

`keep_complement`
: (`logical(1)`)
Whether the operation should keep both resulting individuals (`TRUE`), or only the first and discard the complement (`FALSE`). Default `TRUE`. The `$keep_complement` field will reflect this value.

Value

an object of class Recombinator: `rec("cmpmaybe", rec("swap"))`.

Supported Operand Types

Supported Param classes are: ParamLgl, ParamInt, ParamDbl, ParamFct

Dictionary

This Recombinator can be created with the short access form `rec()` (`recs()` to get a list), or through the the dictionary dict_recombinators in the following way:

```
# preferred:
rec("xounif")
recs("xounif")  # takes vector IDs, returns list of Recombinators

# long form:
dict_recombinators$get("xounif")
```

See Also

Other recombinators: OperatorCombination, RecombinatorPair, Recombinator, dict_recombinators_cmpmaybe, dict_recombinators_convex, dict_recombinators_cvxpair, dict_recombinators_maybe, dict_recombinators_nul dict_recombinators_proxy, dict_recombinators_sbx, dict_recombinators_sequential, dict_recombinators_swa dict_recombinators_xonary

Examples

```
set.seed(1)
rx = rec("xounif")

print(rx)

p = ps(x = p_int(-5, 5), y = p_dbl(-5, 5), z = p_dbl(-5, 5))
data = data.frame(x = 0:5, y = 0:5, z = 0:5)

rx$prime(p)
rx$operate(data)

rx$param_set$values$p = 0.3
rx$operate(data)
```

dict_scalors *Dictionary of Scalors*

Description

Dictionary of Scalors

Usage

```
dict_scalors
```

Format

An object of class `DictionaryScalor` (inherits from `DictionaryEx`, `Dictionary`, `R6`) of length 15.

Methods

Methods inherited from `Dictionary`, as well as:

- `help(key, help_type)`
  `(character(1), character(1))`
  Displays help for the dictionary entry `key`. `help_type` is one of `"text"`, `"html"`, `"pdf"` and given as the `help_type` argument of R's `help()`.

See Also

Other dictionaries: `dict_filtors`, `dict_mutators`, `dict_recombinators`, `dict_selectors`, `mut()`

dict_scalors_aggregate *Scalor giving Weighted Sum of Multiple Scalors*

Description

`Scalor` that applies multiple other `Scalor`s and calculates their weighted sum.

Configuration Parameters

This operation has the configuration parameters of the `Scalor`s that it wraps: The configuration parameters of the operator given to the `scalors` construction argument are prefixed with `"scalor_1"`, `"scalor_2"`, ... up to `"scalor_#"`, where # is `length(scalors)`.

Additional configuration parameters:

- weight_1, weight_2, ... :: numeric(1)
  Weight factors of scalors[[1]], scalors[[2]], etc. Depending on scaling, the outputs of scalors is multiplied with this (when scaling is "linear" or "rank"), or ties between ranks are broken with it (when scaling is "tiebreak"). Initialized to 1.
- scaling :: character(1)
  How to calculate output values, one of "linear", "rank" or "tiebreak". When scaling is "linear", then the output is calculated as the weighted sum of the outputs of scalors, weighted by weight_1, weight_2 etc. When scaling is "rank", then the output is calculated as the weighted sum of the rank() of scalors, weighted by weight_1, weight_2 etc., with ties broken by average. When scaling is "tiebreak", then the output is calculated as the averaged rank() of the scalors with the highest weight_#, with ties broken by the average rank() of the second highest weight_#, with remaining ties broken by scalors with third highest weight_# etc. Initialized to "linear".
- scale_output :: logical(1)
  Whether to scale the output to lie between 0 and 1. Initialized to FALSE.

Dictionary

This Scalor can be created with the short access form scl() (scls() to get a list), or through the the dictionary dict_scalors in the following way:

```
# preferred:
scl("aggregate", <scalors>)
scls("aggregate", <scalors>)  # takes vector IDs, returns list of Scalors

# long form:
dict_scalors$get("aggregate", <scalors>)
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Scalor -> ScalorAggregate

Active bindings

scalors (list of Scalor)
: Scalors being wrapped. These operators are run and their outputs weighted.

Methods

Public methods:

- ScalorAggregate$new()
- ScalorAggregate$prime()
- ScalorAggregate$clone()

Method new(): Initialize the ScalorAggregate object.

Usage:

```
ScalorAggregate$new(scalors)
```

Arguments:

scalors (list of Scalor)
: Scalors to wrap. The operations are run and weighted by weight_# configuration parameters, depending on the scaling configuration parameter. The constructed object gets a *clone* of this argument. The $scalors field will reflect this value.

Method prime()**:** See MiesOperator method. Primes both this operator, as well as the wrapped operators given to scalors during construction.

Usage:

```
ScalorAggregate$prime(param_set)
```

Arguments:

param_set (ParamSet)
: Passed to MiesOperator$prime().

Returns: invisible self.

Method clone()**:** The objects of this class are cloneable with this method.

Usage:

```
ScalorAggregate$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other scalors: Scalor, dict_scalors_domcount, dict_scalors_fixedprojection, dict_scalors_hypervolume, dict_scalors_nondom, dict_scalors_one, dict_scalors_proxy, dict_scalors_single

Other scalor wrappers: dict_scalors_fixedprojection, dict_scalors_proxy

Examples

```
p = ps(x = p_dbl(-5, 5))
data = data.frame(x = rep(0, 5))

sa = scl("aggregate", list(
    scl("one", objective = 1),
    scl("one", objective = 2)
))
sa$prime(p)

(fitnesses = matrix(c(1, 5, 2, 3, 0, 3, 1, 0, 10, 8), ncol = 2))

# to see the fitness matrix, use:
## plot(fitnesses, pch = as.character(1:5))

# default weight 1 -- sum of both objectives
sa$operate(data, fitnesses)

# only first objective
sa$param_set$values[c("weight_1", "weight_2")] = c(1, 0)
sa$operate(data, fitnesses)
```

```
# only 2 * second objective
sa$param_set$values[c("weight_1", "weight_2")] = c(0, 2)
sa$operate(data, fitnesses)
```

`dict_scalors_domcount` *Scalor Counting Dominating Individuals*

Description

`Scalor` that returns a the number of (weakly, epsilon-) dominated or dominating individuals for each individuum.

Configuration Parameters

- `output` :: `character(1)`
  What to count: individuals that are being dominated by the point under consideration(`"count_dominated"`), or individuals that do not dominate the point under consideration (`"count_not_dominating"`). In both cases, a larger output means the individual is "better", in some way, according to the fitness values. Initialized with `"count_not_dominating"`.
- `epsilon` :: `numeric`
  Epsilon-value for non-dominance, as used by `rank_nondominated`. Initialized to `0`.
- `jitter` :: `logical(1)`
  Whether to add random jitter to points, with magnitude `sqrt(.Machine$double.eps)` relative to fitness values. This is used to effectively break ties.
- `scale_output` :: `logical(1)`
  Whether to scale output by the total numberof individuals, giving output between `0` and `1` (inclusive) when `TRUE` or integer outputs ranging from 0 and `nrow(fitnesses)` (inclusive) when `FALSE`. Initialized to `TRUE`.

Supported Operand Types

Supported `Param` classes are: `ParamLgl`, `ParamInt`, `ParamDbl`, `ParamFct`

Dictionary

This `Scalor` can be created with the short access form `scl()` (`scls()` to get a list), or through the the dictionary `dict_scalors` in the following way:

```
# preferred:
scl("domcount")
scls("domcount")  # takes vector IDs, returns list of Scalors

# long form:
dict_scalors$get("domcount")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Scalor -> ScalorDomcount

Methods

Public methods:

- ScalorDomcount$new()
- ScalorDomcount$clone()

Method `new()`**:** Initialize the `ScalorNondom` object.

Usage:

```
ScalorDomcount$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ScalorDomcount$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other scalors: Scalor, dict_scalors_aggregate, dict_scalors_fixedprojection, dict_scalors_hypervolume, dict_scalors_nondom, dict_scalors_one, dict_scalors_proxy, dict_scalors_single

Examples

```
p = ps(x = p_dbl(-5, 5))
data = data.frame(x = rep(0, 5))

sd = scl("domcount")
sd$prime(p)

(fitnesses = matrix(c(1, 5, 2, 3, 0, 3, 1, 0, 10, 8), ncol = 2))

# to see the fitness matrix, use:
## plot(fitnesses, pch = as.character(1:5))

# note that for both 2 and 4, all points do not dominate them
# their value is therefore 1
sd$operate(data, fitnesses)

sd$param_set$values$scale_output = FALSE
sd$operate(data, fitnesses)

sd$param_set$values$output = "count_dominated"
# point 4 dominates three other points, point 2 only one other point.
sd$operate(data, fitnesses)
```

dict_scalors_fixedprojection
Multi-Objective Fixed Projection Scalor

Description

Scalor that returns the maximum of a set of projections.

Priming PS must contain a "scalarization_weights" tagged ParamUty that contains weight matrices (Nobjectives x Nweights) or vectors (if Nweights is 1).

Configuration Parameters

- scalarization :: function
  Function taking a fitness-matrix fitnesses (Nindivs x Nobjectives, with higher values indicating higher desirability) and a list of weight matrices weights (Nindivs elements of Nobjectives x Nweights matrices; positive weights should indicate a positive contribution to scale) and returns a matrix of scalarizations (Nindivs x Nweights, with higher values indicating greater desirability).
  While custom functions can be used, it is recommended to use a Scalarizer, such as scalarizer_linear(), or scalarizer_chebyshev().

Supported Operand Types

Supported Param classes are: ParamLgl, ParamInt, ParamDbl, ParamFct

Dictionary

This Scalor can be created with the short access form scl() (scls() to get a list), or through the the dictionary dict_scalors in the following way:

```
# preferred:
scl("fixedprojection")
scls("fixedprojection")  # takes vector IDs, returns list of Scalors

# long form:
dict_scalors$get("fixedprojection")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Scalor -> ScalorFixedProjection

Active bindings

weights_component_id (numeric(1))
search space component identifying the weights by which to scalarize.

Methods

Public methods:

- `ScalorFixedProjection$new()`
- `ScalorFixedProjection$prime()`
- `ScalorFixedProjection$clone()`

Method `new()`**:** Initialize the `ScalorFixedProjection` object.

Usage:

```
ScalorFixedProjection$new(weights_component_id = "scalarization_weights")
```

Arguments:

`weights_component_id` (`character(1)`)
Id of the search space component identifying the weights by which to scalarize. Default `"scalarization_weights"`.

Method `prime()`**:** See `MiesOperator` method. Primes both this operator, as well as the operator given to the `operation` configuration parameter. Note that this modifies the `$param_set$values$operation` object.

Usage:

```
ScalorFixedProjection$prime(param_set)
```

Arguments:

`param_set` (`ParamSet`)
Passed to `MiesOperator$prime()`.

Returns: invisible `self`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ScalorFixedProjection$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other scalors: `Scalor`, `dict_scalors_aggregate`, `dict_scalors_domcount`, `dict_scalors_hypervolume`, `dict_scalors_nondom`, `dict_scalors_one`, `dict_scalors_proxy`, `dict_scalors_single`

Other scalor wrappers: `dict_scalors_aggregate`, `dict_scalors_proxy`

Examples

```
set.seed(1)
```

dict_scalors_hypervolume *Hypervolume Scalor*

Description

Scalor that returns the hypervolume of each individual, relative to `nadir` and as a contribution over `baseline`. The returned scalar value is the measure of all points that have fitnesses that are

- greater than the respective value in `nadir` in all dimensions, and
- smaller than the respective value in the given point in all dimensions, and
- greater than all points in `baseline` in at least one dimension.

`baseline` should probably be a `paradox::ContextPV` and generate fitness values from the `Archive` in the context using `mies_get_fitnesses`.

Configuration Parameters

- `scale_output` :: `logical(1)`
  Whether to scale output to lie between 0 and 1.
- `nadir` :: `numeric`
  Nadir of fitness values relative to which hypervolume ution is calculated.
- `baseline` :: `matrix`
  Fitness-matrix with one column per objective, giving a population over which the hypervolume improvement should be calculated.

Supported Operand Types

Supported `Param` classes are: `ParamLgl`, `ParamInt`, `ParamDbl`, `ParamFct`

Dictionary

This `Scalor` can be created with the short access form `scl()` (`scls()` to get a list), or through the the dictionary `dict_scalors` in the following way:

```
# preferred:
scl("hypervolume")
scls("hypervolume")  # takes vector IDs, returns list of Scalors

# long form:
dict_scalors$get("hypervolume")
```

Super classes

`miesmuschel::MiesOperator` -> `miesmuschel::Scalor` -> `ScalorHypervolume`

Methods

Public methods:

- `ScalorHypervolume$new()`
- `ScalorHypervolume$clone()`

Method `new()`**:** Initialize the `ScalorHypervolume` object.

Usage:

`ScalorHypervolume$new()`

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`ScalorHypervolume$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

See Also

Other scalors: `Scalor`, `dict_scalors_aggregate`, `dict_scalors_domcount`, `dict_scalors_fixedprojection`, `dict_scalors_nondom`, `dict_scalors_one`, `dict_scalors_proxy`, `dict_scalors_single`

Examples

```
sv = scl("hypervolume")
p = ps(x = p_dbl(-5, 5))
# dummy data; note that ScalorHV does not depend on data content
data = data.frame(x = rep(0, 5))
fitnesses = matrix(c(1, 5, 2, 3, 0, 3, 1, 0, 10, 8), ncol = 2)
sv$param_set$values$baseline = matrix(c(1, 1), ncol = 2)
sv$param_set$values$nadir = c(0, -1)

sv$prime(p)

sv$operate(data, fitnesses)
```

`dict_scalors_nondom` *Nondominated Sorting Scalor*

Description

`Scalor` that returns a the rank of the pareto-front in nondominated sorting as scale. Higher ranks indocate higher fitnesses and therefore "better" individuals.

Configuration Parameters

- epsilon
- nadir
- jitter
- scale_output
- tiebreak

Supported Operand Types

Supported Param classes are: ParamLgl, ParamInt, ParamDbl, ParamFct

Dictionary

This Scalor can be created with the short access form scl() (scls() to get a list), or through the the dictionary dict_scalors in the following way:

```
# preferred:
scl("nondom")
scls("nondom")  # takes vector IDs, returns list of Scalors

# long form:
dict_scalors$get("nondom")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Scalor -> ScalorNondom

Methods

Public methods:

- ScalorNondom$new()
- ScalorNondom$clone()

Method new(): Initialize the ScalorNondom object.

Usage:

```
ScalorNondom$new()
```

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
ScalorNondom$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other scalors: Scalor, dict_scalors_aggregate, dict_scalors_domcount, dict_scalors_fixedprojection, dict_scalors_hypervolume, dict_scalors_one, dict_scalors_proxy, dict_scalors_single

Examples

```
so = scl("nondom")
p = ps(x = p_dbl(-5, 5))
# dummy data; note that ScalorNondom does not depend on data content
data = data.frame(x = rep(0, 5))
fitnesses = matrix(c(1, 5, 2, 3, 0, 3, 1, 0, 10, 8), ncol = 2)

so$prime(p)

so$operate(data, fitnesses)
```

`dict_scalors_one` *Single Dimension Scalor*

Description

`Scalor` that returns a the fitness value of a single objective dimension as scale.

Configuration Parameters

- `objective` :: `integer(1)`
  objective to return as scale, ranges from 1 (the default, first objective) to the number of objectives of the function being optimized.

Supported Operand Types

Supported `Param` classes are: `ParamLgl`, `ParamInt`, `ParamDbl`, `ParamFct`

Dictionary

This `Scalor` can be created with the short access form `scl()` (`scls()` to get a list), or through the the dictionary `dict_scalors` in the following way:

```
# preferred:
scl("one")
scls("one")  # takes vector IDs, returns list of Scalors

# long form:
dict_scalors$get("one")
```

Super classes

`miesmuschel::MiesOperator` -> `miesmuschel::Scalor` -> `ScalorOne`

Methods

Public methods:

- `ScalorOne$new()`
- `ScalorOne$clone()`

Method `new()`**:** Initialize the `ScalorOne` object.

Usage:

`ScalorOne$new()`

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`ScalorOne$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

See Also

Other scalors: `Scalor`, `dict_scalors_aggregate`, `dict_scalors_domcount`, `dict_scalors_fixedprojection`, `dict_scalors_hypervolume`, `dict_scalors_nondom`, `dict_scalors_proxy`, `dict_scalors_single`

Examples

```
so = scl("one")
p = ps(x = p_dbl(-5, 5))
# dummy data; note that ScalorOne does not depend on data content
data = data.frame(x = rep(0, 5))
fitnesses = matrix(c(1, 5, 2, 3, 0, 3, 1, 0, 10, 8), ncol = 2)

so$prime(p)

so$operate(data, fitnesses)

so$param_set$values$objective = 2

so$operate(data, fitnesses)
```

`dict_scalors_proxy` *Proxy-Scalor that Scales According to its Configuration parameter*

Description

`Scalor` that performs the operation in its `operation` configuration parameter. This is useful, e.g., to make `SelectorBest`'s operation fully parametrizable.

Configuration Parameters

- operation :: Scalor
  Operation to perform. Initialized to ScalorSingleObjective. This is primed when $prime() of ScalorProxy is called, and also when $operate() is called, to make changing the operation as part of self-adaption possible. However, if the same operation gets used inside multiple ScalorProxy objects, then it is recommended to $clone(deep = TRUE) the object before assigning them to operation to avoid frequent re-priming.

Supported Operand Types

Supported Param classes are: ParamLgl, ParamInt, ParamDbl, ParamFct

Dictionary

This Scalor can be created with the short access form scl() (scls() to get a list), or through the the dictionary dict_scalors in the following way:

```
# preferred:
scl("proxy")
scls("proxy")  # takes vector IDs, returns list of Scalors

# long form:
dict_scalors$get("proxy")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Scalor -> ScalorProxy

Methods

Public methods:

- ScalorProxy$new()
- ScalorProxy$prime()
- ScalorProxy$clone()

Method `new()`**:** Initialize the ScalorProxy object.

Usage:

```
ScalorProxy$new()
```

Method `prime()`**:** See MiesOperator method. Primes both this operator, as well as the operator given to the operation configuration parameter. Note that this modifies the $param_set$values$operation object.

Usage:

```
ScalorProxy$prime(param_set)
```

Arguments:

param_set (ParamSet)
    Passed to MiesOperator$prime().

Returns: invisible self.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
ScalorProxy$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other scalors: Scalor, dict_scalors_aggregate, dict_scalors_domcount, dict_scalors_fixedprojection, dict_scalors_hypervolume, dict_scalors_nondom, dict_scalors_one, dict_scalors_single

Other scalor wrappers: dict_scalors_aggregate, dict_scalors_fixedprojection

Examples

```
set.seed(1)
sp = scl("proxy")
p = ps(x = p_dbl(-5, 5))
# dummy data; note that ScalorOne does not depend on data content
data = data.frame(x = rep(0, 5))
fitnesses = c(1, 5, 2, 3, 0)

sp$param_set$values$operation = scl("one")
sp$prime(p)
sp$operate(data, fitnesses)
```

---

dict_scalors_single *Single Objective Scalor*

Description

Scalor that uses a single given objective, throwing an error in case it is used in a multi-objective problem.

In contrast to ScalorOne, this Scalor throws an error when more than one objective is present. When this Scalor gets used as the default value, e.g. for a Selector, then it forces the user to make an explicit decision about what Scalor to use in a multi-objective setting.

Configuration Parameters

No configuration parameters.

Supported Operand Types

Supported Param classes are: ParamLgl, ParamInt, ParamDbl, ParamFct

Dictionary

This Scalor can be created with the short access form scl() (scls() to get a list), or through the the dictionary dict_scalors in the following way:

```
# preferred:
scl("single")
scls("single")  # takes vector IDs, returns list of Scalors

# long form:
dict_scalors$get("single")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Scalor -> ScalorSingleObjective

Methods

Public methods:

- ScalorSingleObjective$new()
- ScalorSingleObjective$clone()

Method new()**:** Initialize the ScalorSingleObjective object.

Usage:

```
ScalorSingleObjective$new()
```

Method clone()**:** The objects of this class are cloneable with this method.

Usage:

```
ScalorSingleObjective$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other scalors: Scalor, dict_scalors_aggregate, dict_scalors_domcount, dict_scalors_fixedprojection, dict_scalors_hypervolume, dict_scalors_nondom, dict_scalors_one, dict_scalors_proxy

Examples

```
ss = scl("single")
p = ps(x = p_dbl(-5, 5))
# dummy data; note that ScalorOne does not depend on data content
data = data.frame(x = rep(0, 5))
fitnesses_so = c(1, 5, 2, 3, 0)
fitnesses_mo = matrix(c(1, 5, 2, 3, 0, 3, 1, 0, 10, 8), ncol = 2)

ss$prime(p)

ss$operate(data, fitnesses_so)
```

```
try(ss$operate(data, fitnesses_mo))
```

dict_selectors *Dictionary of Selectors*

Description

Dictionary of Selectors

Usage

```
dict_selectors
```

Format

An object of class `DictionarySelector` (inherits from `DictionaryEx`, `Dictionary`, `R6`) of length 15.

Methods

Methods inherited from `Dictionary`, as well as:

- `help(key, help_type)`
  `(character(1), character(1))`
  Displays help for the dictionary entry `key`. `help_type` is one of `"text"`, `"html"`, `"pdf"` and given as the `help_type` argument of R's `help()`.

See Also

Other dictionaries: `dict_filtors`, `dict_mutators`, `dict_recombinators`, `dict_scalors`, `mut()`

dict_selectors_best *Best Value Selector*

Description

`Selector` that selects the top `n_select` individuals based on the fitness value, breaking ties randomly. When `n_select` is larger than the number of individuals, the selection wraps around: All `nrow(values)` individuals are selected at least `floor(nrow(values) / n_select)` times, with the top `nrow(values) %% n_select` individuals being selected one more time.

Configuration Parameters

- shuffle_selection :: logical(1)
  Whether to shuffle the selected output. When this is TRUE, selected individuals are returned in random order, so when this operator is e.g. used in mies_generate_offspring(), then subsequent recombination operators effectively operate on pairs (or larger groups) of random individuals. Otherwise they are returned in order, and recombination operates on the first batch of n_indivs_in returned individuals first, then the second batch etc. in order. Initialized to TRUE (recommended).

Supported Operand Types

Supported Param classes are: ParamLgl, ParamInt, ParamDbl, ParamFct

Dictionary

This Selector can be created with the short access form sel() (sels() to get a list), or through the the dictionary dict_selectors in the following way:

```
# preferred:
sel("best")
sels("best")  # takes vector IDs, returns list of Selectors

# long form:
dict_selectors$get("best")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Selector -> miesmuschel::SelectorScalar -> SelectorBest

Methods

Public methods:

- SelectorBest$new()
- SelectorBest$clone()

Method new()**:** Initialize the SelectorBest object.

Usage:

```
SelectorBest$new(scalor = ScalorSingleObjective$new())
```

Arguments:

scalor (Scalor)
: Scalor to use to generate scalar values from multiple objectives, if multi-objective optimization is performed. Initialized to ScalorSingleObjective: Doing single-objective optimization normally, throwing an error if used in multi-objective setting: In that case, a Scalor needs to be explicitly chosen.

Method clone()**:** The objects of this class are cloneable with this method.

Usage:

```
SelectorBest$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other selectors: SelectorScalar, Selector, dict_selectors_maybe, dict_selectors_null, dict_selectors_proxy, dict_selectors_random, dict_selectors_sequential, dict_selectors_tournament

Examples

```
sb = sel("best")
p = ps(x = p_dbl(-5, 5))
# dummy data; note that SelectorBest does not depend on data content
data = data.frame(x = rep(0, 5))
fitnesses = c(1, 5, 2, 3, 0)

sb$prime(p)

sb$operate(data, fitnesses, 2)

sb$param_set$values$shuffle_selection = FALSE

sb$operate(data, fitnesses, 4)
```

---

dict_selectors_maybe *Selector-Combination that Selects According to Two Selectors*

Description

Selector that wraps two other Selectors given during construction and uses both for selection proportionally. Each of the resulting n_select individuals is chosen either from $selector, or from $selector_not.

This makes it possible to implement selection methods such as random interleaving, where only a fraction of p individuals were selected by a criterion, while the others are taken randomly.

Algorithm

To perform selection, n_selector_in rows of values are given to $selector, and the remaining nrow(values) - n_selector_in rows are given to $selector_not. Both selectors are used to generate a subset of selected individuals: $selector generates n_selector_out individuals, and $selector_not generates n_select - n_selector_out individuals.

n_selector_in is either set to round(nrow(values) * p_in) when proportion_in is "exact", or to rbinom(1, nrow(values), p_in) when proportion_in is "random".

n_selector_out is set to round(n_select * p_out) when proportion_out is "exact", or to rbinom(1, n_select, p_out) when proportion_out is "random".

When odds_correction is TRUE, then p_out is adjusted depending on the used n_selector_in value before being applied. Let odds(p) = p/(1-p). Then the effective p_out is set such that odds(effective p_out) = odds(p_out) * n_selector_in / (nrow(values) - n_selector_in) / odds(p_in). This corrects for the discrepancy between the chosen p_in and the effective proportion of n_selector_in / nrow(values) caused either by rounding errors or when proportion_in is "random".

When p_in is exactly 1 or exactly 0, and p_out is not equal to p_in, then an error is given.

If nrow(values) is 1, then this individuum is returned and $selector / $selector_not are not called.

If try_unique is TRUE, then n_selector_out is set to at most n_selector_in and at least n_select - nrow(values) + n_selector_in, and an error is generated when nrow(values) is less than n_select.

If try_unique is FALSE and odds_correction is TRUE and n_selector_in is either 0 or nrow(values), then $p_out is set to either 0 or 1, respectively.

If try_unique is FALSE and odds_correction is FALSE and n_selector_in is either 0 or nrow(values), and n_selector_out is not equal to 0 or n_select, respectively, then n_selector_in is increased / decreased by 1 to give $selector_not / $selector at least one individuum to choose from. While this behaviour may seem pathological, it is to ensure continuity with sampled values of n_selector_in that are close to 0 or n_select.

If n_selector_out is n_select or 0, or if n_selector_in is nrows(values) - 1 or 1, then only $selector / $selector_not is executed, respectively; possibly with a subset of values if n_selector_in differs from nrow(values) / 0.

Configuration Parameters

This operator has the configuration parameters of the Selectors that it wraps: The configuration parameters of the operator given to the selector construction argument are prefixed with "maybe.", the configuration parameters of the operator given to the selector_not construction argument are prefixed with "maybe_not.".

Additional configuration parameters:

- p_in :: numeric(1)
  Probability per individual (when random_choise is TRUE), or fraction of individuals (when random_choice is FALSE), that are given to $selector instead of $selector_not. This may be overriden when try_unique is TRUE, in which case at least as many rows are given to $selector and $selector_not as they are generating output values respectively. When this is exactly 1 or exactly 0, then p_out must be equal to p_in. Must be set by the user.
- p_out :: numeric(1)
  Probability per output value (when random_choise is TRUE), or fraction of output values (when random_choice is FALSE), that are generated by $selector instead of $selector_not. When this values is not given, it defaults to p_in.
- shuffle_input :: logical(1)
  Whether to distribute input values randomly to $selector / $selector_not. If FALSE, then the first part of values is given to $selector. This only randomizes *which* lines of values are given to $selector / $selector_not, but it does not necessarily reorder the lines of values given to each. In particular, if p_out is 0 or 1, then no shuffling takes place. Initialized to TRUE.

- proportion_in :: character(1)
  When set to "random", sample the number of individuals given to $selector according to rbinom(1, nrow(values), p_in). When set to "exact", give $selector round(nrow(values) * p_in) individuals. Initialized to "exact".
- proportion_out :: character(1)
  When set to "random", sample the number of individuals generated by $selector according to rbinom(1, n_select, p_out). When set to "exact", have $selector generate round(n_select * p_out) individuals.
- odds_correction :: logical(1)
  When set, the effectively used value of p_out is set to 1 / (1 + ((nrow(values) - n_selector_in) * p_in * (1 - p_out)) / (n_selector_in * p_out * (1 - p_in))), see the **Algorithm** section. Initialized to FALSE.
- try_unique :: logical(1)
  Whether to give at least as many rows of values to each of $selector and $selector_not as they are generating output values. This should be set to TRUE whenever SelectorMaybe is used to select unique values, and can be set to FALSE when selecting values multiple times is acceptable. When this is TRUE, then having n_select > nrow(values) generates an error. Initialized to TRUE.

Supported Operand Types

Supported Param classes are the set intersection of supported classes of selector and selector_not.

Dictionary

This Filtor can be created with the short access form ftr() (ftrs() to get a list), or through the the dictionary dict_filtors in the following way:

```
# preferred:
ftr("maybe", <selector> [, <selector_not>])
ftrs("maybe", <selector> [, <selector_not>])  # takes vector IDs, returns list of Filtors

# long form:
dict_filtors$get("maybe", <selector> [, <selector_not>])
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Selector -> SelectorMaybe

Active bindings

selector (Selector)
: Selector being wrapped. This operator gets run with probability / proportion p_in and generates output with probability / proportion p_out (configuration parameters).

selector_not (Selector)
: Alternative Selector being wrapped. This operator gets run with probability / proportion 1 - p_in and generates output with probability / proportion 1 - p_out (configuration parameters).

Methods

Public methods:

- SelectorMaybe$new()
- SelectorMaybe$prime()
- SelectorMaybe$clone()

Method `new()`**:** Initialize the `SelectorMaybe` object.

Usage:

```
SelectorMaybe$new(selector, selector_not = SelectorRandom$new())
```

Arguments:

`selector` (Selector)
Selector to wrap. This operator gets run with probability / fraction `p_in` (Configuration parameter).
The constructed object gets a *clone* of this argument. The `$selector` field will reflect this value.

`selector_not` (Selector)
Another Selector to wrap. This operator runs when `selector` is not chosen. By default, this is SelectorRandom, i.e. selecting randomly.
The constructed object gets a *clone* of this argument. The `$selector_not` field will reflect this value.

Method `prime()`**:** See MiesOperator method. Primes both this operator, as well as the wrapped operators given to `selector` and `selector_not` during construction.

Usage:

```
SelectorMaybe$prime(param_set)
```

Arguments:

`param_set` (ParamSet)
Passed to MiesOperator`$prime()`.

Returns: invisible `self`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SelectorMaybe$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other selectors: SelectorScalar, Selector, dict_selectors_best, dict_selectors_null, dict_selectors_proxy, dict_selectors_random, dict_selectors_sequential, dict_selectors_tournament

Other selector wrappers: dict_selectors_proxy, dict_selectors_sequential dict_selectors_null *Null Selector*

Description

Selector that disregards fitness and individual values and selects individuals by order in which they are given.

Configuration Parameters

- shuffle_selection :: logical(1)
  Whether to shuffle the selected output. When this is TRUE, selected individuals are returned in random order, so when this operator is e.g. used in mies_generate_offspring(), then subsequent recombination operators effectively operate on pairs (or larger groups) of random individuals. Otherwise they are returned in order, and recombination operates on the first batch of n_indivs_in returned individuals first, then the second batch etc. in order. Initialized to TRUE (recommended).

Supported Operand Types

Supported Param classes are: ParamLgl, ParamInt, ParamDbl, ParamFct

Dictionary

This Selector can be created with the short access form sel() (sels() to get a list), or through the the dictionary dict_selectors in the following way:

```
# preferred:
sel("null")
sels("null")  # takes vector IDs, returns list of Selectors

# long form:
dict_selectors$get("null")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Selector -> SelectorNull

Methods

Public methods:

- SelectorNull$new()
- SelectorNull$clone()

Method new()**:** Initialize the SelectorNull object.

Usage:

```
SelectorNull$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SelectorNull$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other selectors: `SelectorScalar`, `Selector`, `dict_selectors_best`, `dict_selectors_maybe`, `dict_selectors_proxy`, `dict_selectors_random`, `dict_selectors_sequential`, `dict_selectors_tournament`

Examples

```
sn = sel("null")
p = ps(x = p_dbl(-5, 5))
# dummy data; note that SelectorNull does not depend on data content
data = data.frame(x = rep(0, 5))
fitnesses = c(1, 5, 2, 3, 0)

sn$prime(p)

sn$operate(data, fitnesses, 2)
sn$operate(data, fitnesses, 4)
sn$operate(data, fitnesses, 6)
```

`dict_selectors_proxy` *Proxy-Selector that Selects According to its Configuration Parameter*

Description

`Selector` that performs the operation in its `operation` configuration parameter. This is useful, e.g., to make `OptimizerMies`'s selection operations fully parametrizable.

Configuration Parameters

- `operation` :: `Selector`
  Operation to perform. Initialized to `SelectorBest`. This is primed when `$prime()` of `SelectorProxy` is called, and also when `$operate()` is called, to make changing the operation as part of self-adaption possible. However, if the same operation gets used inside multiple `SelectorProxy` objects, then it is recommended to `$clone(deep = TRUE)` the object before assigning them to `operation` to avoid frequent re-priming.

Supported Operand Types

Supported `Param` classes are: `ParamLgl`, `ParamInt`, `ParamDbl`, `ParamFct`

Dictionary

This Selector can be created with the short access form sel() (sels() to get a list), or through the the dictionary dict_selectors in the following way:

```
# preferred:
sel("proxy")
sels("proxy")  # takes vector IDs, returns list of Selectors

# long form:
dict_selectors$get("proxy")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Selector -> SelectorProxy

Methods

Public methods:

- SelectorProxy$new()
- SelectorProxy$prime()
- SelectorProxy$clone()

Method new()**:** Initialize the SelectorProxy object.

Usage:

```
SelectorProxy$new()
```

Method prime()**:** See MiesOperator method. Primes both this operator, as well as the operator given to the operation configuration parameter. Note that this modifies the $param_set$values$operation object.

Usage:

```
SelectorProxy$prime(param_set)
```

Arguments:

param_set (ParamSet)
    Passed to MiesOperator$prime().

Returns: invisible self.

Method clone()**:** The objects of this class are cloneable with this method.

Usage:

```
SelectorProxy$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other selectors: SelectorScalar, Selector, dict_selectors_best, dict_selectors_maybe, dict_selectors_null, dict_selectors_random, dict_selectors_sequential, dict_selectors_tournament

Other selector wrappers: dict_selectors_maybe, dict_selectors_sequential

Examples

```
set.seed(1)
sp = sel("proxy")
p = ps(x = p_dbl(-5, 5))
# dummy data; note that SelectorBest does not depend on data content
data = data.frame(x = rep(0, 5))
fitnesses = c(1, 5, 2, 3, 0)

sp$param_set$values$operation = sel("random")
sp$prime(p)
sp$operate(data, fitnesses, 2)

sp$param_set$values$operation = sel("best")
sp$operate(data, fitnesses, 2)
```

---

dict_selectors_random *Random Selector*

Description

Random selector that disregards fitness and individual values and selects individuals randomly. Depending on the configuration parameter `replace`, it samples with or without replacement.

Configuration Parameters

- `sample_unique` :: `character(1)`
  Whether to sample individuals globally unique (`"global"`), unique within groups (`"groups"`), or not unique at all (`"no"`, sample with replacement). This is done with best effort; if `group_size` (when `sample_unique` is `"groups"`) or `n_select` (when `sample_unique` is `"global"`) is greater than `nrow(values)`, then individuals are selected with as few repeats as possible. Initialized to `"groups"`.

Supported Operand Types

Supported `Param` classes are: `ParamLgl`, `ParamInt`, `ParamDbl`, `ParamFct`

Dictionary

This `Selector` can be created with the short access form `sel()` (`sels()` to get a list), or through the the dictionary `dict_selectors` in the following way:

```
# preferred:
sel("random")
sels("random")  # takes vector IDs, returns list of Selectors

# long form:
dict_selectors$get("random")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Selector -> SelectorRandom

Methods

Public methods:

- SelectorRandom$new()
- SelectorRandom$clone()

Method `new()`: Initialize the `SelectorRandom` object.

Usage:

```
SelectorRandom$new()
```

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
SelectorRandom$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other selectors: SelectorScalar, Selector, dict_selectors_best, dict_selectors_maybe, dict_selectors_null, dict_selectors_proxy, dict_selectors_sequential, dict_selectors_tournament

Examples

```
set.seed(1)
sr = sel("random")
p = ps(x = p_dbl(-5, 5))
# dummy data; note that SelectorRandom does not depend on data content
data = data.frame(x = rep(0, 5))
fitnesses = c(1, 5, 2, 3, 0)

sr$prime(p)

sr$operate(data, fitnesses, 2)
sr$operate(data, fitnesses, 2)
sr$operate(data, fitnesses, 2)

sr$operate(data, fitnesses, 4)
sr$operate(data, fitnesses, 4)
sr$operate(data, fitnesses, 4)
```

`dict_selectors_sequential` *Run Multiple Selection Operations in Sequence*

Description

`Selector` that wraps multiple other `Selector`s given during construction and uses them for selection in sequence. This makes it possible for one `Selector` to discard a few individuals, followed by a second `Selector` to discard more, etc., until `n_select` individuals are remaining.

Algorithm

Given that there are `nrow(values)` input individuals in an operation, and `n_select` individuals requested to be selected, the operation calls `selector_i` for `i` in 1 ... `length(selectors)` to reduce the number of individuals in this pipeline. The relative quantity by which the number of individuals is reduced in each step is determined by the configuration parameters `reduction_1`, `reduction_2`, etc., and also dependent on the sum of these values, in the following denoted, with a slight abuse of notation, by `sum[reduction_#]`.

Let the number of individuals passed to step `i` be denoted by `n_values[i]`, and the number of individuals requested to be selected by that step be denoted as `n_select_[i]`. In particular, `n_values[1] == nrow(values)`, and `n_select_[length(selectors)] == n_select`.

When `reduction_by_factor` is `TRUE`, then the reduction at step `i` is done by a factor, meaning that `n_values[i] / n_select_[i]` is set (up to rounding). This factor is `(nrow(values) / n_select) ^ (reduction_i / sum[`

When `reduction_by_factor` is `FALSE`, then the reduction at step `i` is done by absolute differences, meaning that `n_values[i] - n_select_[i]` is set (up to rounding). This difference is `(nrow(values) - n_select) * (reduction_i / sum[reduction_#])`, with `sum[reduction_#]` as above.

In particular, this means that when all `reduction_#` values are the same and `reduction_by_factor` is `TRUE`, then each operation reduces the number of individuals in the pipeline by the same factor. When `reduction_by_factor` is `FALSE`, then each operation removes the same absolute number of individuals.

While the illustrations are done with the assumption that `nrow(values) >= n_select`, they hold equivalently with `nrow(values) < n_select`.

All except the last `Selector`s are called with `group_size` set to their `n_select` value; the last `Selector` is called with the `group_size` value given as input.

Configuration Parameters

This operator has the configuration parameters of the `Selector`s that it wraps: The configuration parameters of the operator given to the `selectors` construction argument are prefixed with `"selector_1"`, `"selector_2"`, ... up to `"selector_#"`, where # is `length(selectors)`.

Additional configuration parameters:

- reduction_1, reduction_2, ... :: numeric(1)
  Relative reduction done by selector_1, selector_2, ..., as described in the section **Algorithm**. The values are all initialized to 1, meaning the same factor (when reduction_by_factor is TRUE) or absolute number (otherwise) of reduction by each operation.
- reduction_by_factor :: logical(1)
  Whether to do reduction by factor (TRUE) or absolute number (FALSE), as described in **Algorithm**. Initialized to TRUE.

Supported Operand Types

Supported Param classes are the set intersection of supported classes of the Selectors given in selectors.

Dictionary

This Selector can be created with the short access form sel() (sels() to get a list), or through the the dictionary dict_selectors in the following way:

```
# preferred:
sel("sequential", <selectors>)
sels("sequential", <selectors>)  # takes vector IDs, returns list of Selectors

# long form:
dict_selectors$get("sequential", <selectors>)
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Selector -> SelectorSequential

Active bindings

selectors (list of Selector)
  Selectors being wrapped. These operators get run sequentially in order.

Methods

Public methods:

- SelectorSequential$new()
- SelectorSequential$prime()
- SelectorSequential$clone()

Method new()**:** Initialize the SelectorSequential object.

Usage:

SelectorSequential$new(selectors)

Arguments:

selectors (list of Selector)
  Selectors to wrap. The operations are run in order given to selectors. The constructed object gets a *clone* of this argument. The $selectors field will reflect this value.

Method `prime()`**:** See `MiesOperator` method. Primes both this operator, as well as the wrapped operators given to `selectors` during construction.

Usage:

```
SelectorSequential$prime(param_set)
```

Arguments:

`param_set` (`ParamSet`)
Passed to `MiesOperator$prime()`.

Returns: invisible `self`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SelectorSequential$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other selectors: `SelectorScalar`, `Selector`, `dict_selectors_best`, `dict_selectors_maybe`, `dict_selectors_null`, `dict_selectors_proxy`, `dict_selectors_random`, `dict_selectors_tournament`

Other selector wrappers: `dict_selectors_maybe`, `dict_selectors_proxy`

`dict_selectors_tournament`
Tournament Selector

Description

`Selector` that repeatedly samples `k` individuals and selects the best ouf of these.

Configuration Parameters

- `k` :: `integer(1)`
  Tournament size. Must be set by the user.
- `choose_per_tournament` :: Number of individuals to choose in each tournament. Must be smaller than `k`. The special value `0` sets this to the `group_size` hint given to the `$operate()`-call (but at most `k`). This is equal to `n_select` when used as survival-selector in `mies_survival_plus()`/`mies_surviva` and equal to `n_indivs_in` of a `Recombinator` used in `mies_generate_offspring()`.
  Initialized to 1.
- `sample_unique` :: `character(1)`
  Whether to sample individuals globally unique (`"global"`, selected individuals are removed from the population after each tournament), unique within groups (`"groups"`, individuals are replaced when `group_size` individuals were sampled), unique per tournament (`"tournament"`, individuals are replaced after each tournament), or not unique at all (`"no"`, individuals are sampled with replacement within tournaments). This is done with best effort; if `group_size`

(when sample_unique is "groups") or n_select (when sample_unique is "global") is greater than nrow(values), then the first nrow(values) * floor(group_size / nrow(values)) or nrow(values) * floor(n_select / nrow(values)) individuals are chosen deterministically by selecting every individual with the same frequency, followed by tournament selection for the remaining required individuals. Initialized to "groups".

Supported Operand Types

Supported Param classes are: ParamLgl, ParamInt, ParamDbl, ParamFct

Dictionary

This Selector can be created with the short access form sel() (sels() to get a list), or through the the dictionary dict_selectors in the following way:

```
# preferred:
sel("tournament")
sels("tournament")  # takes vector IDs, returns list of Selectors

# long form:
dict_selectors$get("tournament")
```

Super classes

miesmuschel::MiesOperator -> miesmuschel::Selector -> miesmuschel::SelectorScalar -> SelectorTournament

Methods

Public methods:

- SelectorTournament$new()
- SelectorTournament$clone()

Method new()**:** Initialize the SelectorTournament object.

Usage:

```
SelectorTournament$new(scalor = ScalorSingleObjective$new())
```

Arguments:

scalor (Scalor)
: Scalor to use to generate scalar values from multiple objectives, if multi-objective optimization is performed. Initialized to ScalorSingleObjective: Doing single-objective optimization normally, throwing an error if used in multi-objective setting: In that case, a Scalor needs to be explicitly chosen.

Method clone()**:** The objects of this class are cloneable with this method.

Usage:

```
SelectorTournament$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other selectors: SelectorScalar, Selector, dict_selectors_best, dict_selectors_maybe, dict_selectors_null, dict_selectors_proxy, dict_selectors_random, dict_selectors_sequential

Examples

```
sb = sel("tournament", k = 4)
p = ps(x = p_dbl(-5, 5))
# dummy data; note that SelectorBest does not depend on data content
data = data.frame(x = rep(0, 7))
fitnesses = c(1, 5, 2, 3, 0, 4, 6)

sb$prime(p)

sb$operate(data, fitnesses, 2)

sb$operate(data, fitnesses, 4, group_size = 2)
```

---

dist_crowding *Calculate Crowding Distance*

Description

Takes a `matrix` of fitness values and calculates the crowding distance for individuals in that `matrix`.

Individuals that are minimal or maximal with respect to at least one dimension are assigned infinite crowding distance.

Individuals are assumed to be in a (epsilon-) nondominated front.

Usage

```
dist_crowding(fitnesses)
```

Arguments

fitnesses	(numeric matrix) fitness matrix, with one row per individual and one column per objective

Value

`numeric`: Vector of crowding distances.

domhv_contribution — *Calculate Hypervolume Contribution*

Description

Takes a `matrix` of fitness values and calculates the hypervolume contributions of individuals in that `matrix`.

Hypervolume contribution of an individual I is the difference between the dominated hypervolume of a set of individuals including I, where the fitness of I is increased by `epsilon`, and the dominated hypervolume of the same set but excluding I.

Individuals that are less than another individual more than `epsilon` in any dimension have hypervolume contribution of 0.

Usage

```
domhv_contribution(fitnesses, nadir = 0, epsilon = 0)
```

Arguments

`fitnesses`	(`numeric matrix`) fitness matrix, with one row per individual and one column per objective
`nadir`	(`numeric`) Lowest fitness point up to which to calculate dominated hypervolume. May be a scalar, in which case it is used for all dimensions, or a vector, in which case its length must match the number of dimensions. Default 0.
`epsilon`	(`numeric`) Added to each individual before calculating its particular hypervolume contribution. `epsilon` may be a scalar, in which case it is used for all dimensions, or a vector, in which case its length must match the number of dimensions. Default 0.

Value

`numeric`: The vector of dominated hypervolume contributions for each individual in `fitnesses`.

Examples

```
(fitnesses = matrix(c(1, 5, 2, 3, 0, 3, 1, 0, 10, 8), ncol = 2))

# to see the fitness matrix, use:
## plot(fitnesses, pch = as.character(1:5))

domhv_contribution(fitnesses)
```

domhv_improvement *Calculate Hypervolume Improvement*

Description

Takes a matrix of fitness values and calculates the hypervolume improvement of individuals in that matrix, one by one, over the baseline individuals.

The hypervolume improvement for each point is the measure of all points that have fitnesses that are

- greater than the respective value in nadir in all dimensions, and
- smaller than the respective value in the given point in all dimensions, and
- greater than all points in baseline in at least one dimension.

Individuals in fitnesses are considered independently of each other. A possible speedup is achieved because baseline individuals only need to be pre-filtered once.

Usage

```
domhv_improvement(fitnesses, baseline = NULL, nadir = 0)
```

Arguments

fitnesses	(numeric matrix) fitness matrix, with one row per individual and one column per objective
baseline	(matrix | NULL) Fitness-matrix with one column per objective, giving a population over which the hypervolume improvement should be calculated. If NULL, the hypervolume of each individual in fitnesses is calculated.
nadir	(numeric) Lowest fitness point up to which to calculate dominated hypervolume. May be a scalar, in which case it is used for all dimensions, or a vector, in which case its length must match the number of dimensions. Default 0.

Value

numeric: The vector of dominated hypervolume contributions for each individual in fitnesses.

Examples

```
(fitnesses = matrix(c(1, 5, 2, 3, 0, 3, 1, 0, 10, 8), ncol = 2))

# to see the fitness matrix, use:
## plot(fitnesses, pch = as.character(1:5))

domhv_improvement(fitnesses)

domhv_improvement(fitnesses, fitnesses[1, , drop = FALSE])
```

Filtor *Filtor Base Class*

Description

Base class representing filter operations, inheriting from MiesOperator.

A Filtor gets a table of individuals that are to be filtered, as well as a table of individuals that were already evaluated, along with information on the latter individuals' performance values. Furthermore, the number of individuals to return is given. The Filtor returns a vector of unique integers indicating which individuals were selected.

Filter operations are performed in ES algorithms to facilitate concentration towards individuals that likely perform well with regard to the fitness measure, without evaluating the fitness measure, for example through a surrogate model.

Fitness values are always *maximized*, both in single- and multi-criterion optimization.

Unlike most other operator types inheriting from MiesOperator, the `$operate()` function has four arguments, which are passed on to `$.filter()`

- `values` :: `data.frame`
  Individuals to filter. Must pass the check of the Param given in the last `$prime()` call and may not have any missing components.
- `known_values` :: `data.frame`
  Individuals to use for filtering. Must pass the check of the Param given in the last `$prime()` call and may not have any missing components. Note that `known_values` may be empty.
- `fitnesses` :: `numeric` | `matrix`
  Fitnesses for each individual given in `known_values`. If this is a `numeric`, then its length must be equal to the number of rows in `values`. If this is a `matrix`, if number of rows must be equal to the number of rows in `values`, and it must have one column when doing single-crit optimization and one column each for each "criterion" when doing multi-crit optimization.
- `n_filter` :: `integer(1)`
  Number of individuals to select. Some `Filtor`s select individuals with replacement, for which this value may be greater than the number of rows in `values`.

The return value for an operation will be a numeric vector of integer values of ength `n_filter` indexing the individuals that were selected. `Filtor` must always return unique integers, i.e. select every individual at most once.

Inheriting

`Filtor` is an abstract base class and should be inherited from. Inheriting classes should implement the private `$.filter()` function. The user of the object calls `$operate()`, and the arguments are passed on to private `$.filter()` after checking that the operator is primed, that the `values` and `known_values` arguments conforms to the primed domain and that other values match.

The `private$.needed_input()` function should also be overloaded, it is called by the public `$needed_input()` function after initial checks; see the documentation there.

Typically, the `$initialize()` function should also be overloaded, and optionally the `$prime()` function; they should call their `super` equivalents.

Super class

miesmuschel::MiesOperator -> Filtor

Active bindings

supported (character)
Optimization supported by this Filtor, can be "single-crit", "multi-crit", or both.

Methods

Public methods:

- Filtor$new()
- Filtor$needed_input()
- Filtor$clone()

Method new()**:** Initialize base class components of the Filtor.

Usage:

```
Filtor$new(
  param_classes = c("ParamLgl", "ParamInt", "ParamDbl", "ParamFct"),
  param_set = ps(),
  supported = c("single-crit", "multi-crit"),
  packages = character(0),
  dict_entry = NULL,
  own_param_set = quote(self$param_set)
)
```

Arguments:

param_classes (character)
Classes of parameters that the operator can handle. May contain any of "ParamLgl", "ParamInt", "ParamDbl", "ParamFct". Default is all of them.
The $param_classes field will reflect this value.

param_set (ParamSet | list of expression)
Strategy parameters of the operator. This should be created by the subclass and given to super$initialize(). If this is a ParamSet, it is used as the MiesOperator's ParamSet directly. Otherwise it must be a list of expressions e.g. created by alist() that evaluate to ParamSets, possibly referencing self and private. These ParamSet are then combined using a ParamSetCollection. Default is the empty ParamSet.
The $param_set field will reflect this value.

supported (character)
Subset of "single-crit" and "multi-crit", indicating wether single and / or multi-criterion optimization is supported. Default both of them.
The $supported field will reflect this value.

packages (character) Packages that need to be loaded for the operator to function. This should be declared so these packages can be loaded when operators run on parallel instances. Default is character(0).
The $packages field will reflect this values.

`dict_entry` (`character(1)` | `NULL`)
: Key of the class inside the Dictionary (usually one of dict_mutators, dict_recombinators, dict_selectors), where it can be retrieved using a short access function. May be NULL if the operator is not entered in a dictionary.
The $dict_entry field will reflect this value.

`own_param_set` (`language`)
: An expression that evaluates to a ParamSet indicating the configuration parameters that are entirely owned by this operator class (and not proxied from a construction argument object). This should be quote(self$param_set) (the default) when the param_set argument is not a list of expressions.

Method `needed_input()`: Calculate the number of values that are required to filter down to output_size, given the current configuraiton parameter settings.

Usage:

```
Filtor$needed_input(output_size)
```

Arguments:

`output_size` (`integer(1)`)
: A positive integer indicating the number of individuals for which the needed input size should be calculated.

Returns: integer(1): The minimum number of rows required to filter down to output_size. At least output_size.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
Filtor$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other base classes: FiltorSurrogate, MiesOperator, MutatorDiscrete, MutatorNumeric, Mutator, OperatorCombination, RecombinatorPair, Recombinator, Scalor, SelectorScalar, Selector

Other filtors: FiltorSurrogate, dict_filtors_maybe, dict_filtors_null, dict_filtors_proxy, dict_filtors_surprog, dict_filtors_surtour

---

FiltorSurrogate *Abstract Surrogate Model Filtering Base Class*

Description

Abstract base class for surrogate model filtering.

A *surrogate model* is a regression model, based on an mlr3::Learner, which predicts the approximate performance of newly sampled configurations given the empirical performance of already evaluated configurations. The surrogate model can be used to propose points that have, according to the surrogate model, a relatively high chance of performing well.

The FiltorSurrogate base class can be inherited from to create different Filtors that filter based on a surrogate model, for example tournament filtering or progresive filtering.

Configuration Parameters

FiltorSurrogateProgressive's configuration parameters are the hyperparameters of the surrogate_learner Learner, as well as the configuration parameters of the surrogate_selector Selector.

Supported Operand Types

Supported Param classes depend on the supported feature types of the surrogate_learner, as reported by surrogate_learner$feature_types: "ParamInt" requires "integer", "ParamDbl" requires "numeric", "ParamLgl" requires "logical", and "ParamFct" requires "factor".

Super classes

miesmuschel::MiesOperator -> miesmuschel::Filtor -> FiltorSurrogate

Active bindings

surrogate_learner (mlr3::LearnerRegr)
Regression learner for the surrogate model filtering algorithm.

surrogate_selector (Selector)
Selector with which to select using surrogate-predicted performance

Methods

Public methods:

- FiltorSurrogate$new()
- FiltorSurrogate$prime()
- FiltorSurrogate$clone()

Method new()**:** Initialize the base class components of the FiltorSurrogate.

Usage:

```
FiltorSurrogate$new(
  surrogate_learner,
  surrogate_selector = SelectorBest$new(),
  param_set = ps(),
  packages = character(0),
  dict_entry = NULL
)
```

Arguments:

surrogate_learner (mlr3::LearnerRegr)
Regression learner for the surrogate model filtering algorithm.
The $surrogate_learner field will reflect this value.

surrogate_learner (mlr3::LearnerRegr)
Regression learner for the surrogate model filtering algorithm.
The $surrogate_learner field will reflect this value.

surrogate_selector (Selector) Selector for the surrogate model filtering algorithm.
The $surrogate_selector field will reflect this value.

surrogate_selector (Selector) Selector for the surrogate model filtering algorithm.
The $surrogate_selector field will reflect this value.

param_set (ParamSet)
ParamSet of the method implemented in the inheriting class with configuration parameters that go beyond the parameters of the surrogate_learner and surrogate_selector.

param_set (ParamSet | list of expression)
Strategy parameters of the operator. This should be created by the subclass and given to super$initialize(). If this is a ParamSet, it is used as the MiesOperator's ParamSet directly. Otherwise it must be a list of expressions e.g. created by alist() that evaluate to ParamSets, possibly referencing self and private. These ParamSet are then combined using a ParamSetCollection. Default is the empty ParamSet.
The $param_set field will reflect this value.

packages (character) Packages that need to be loaded for the operator to function. This should be declared so these packages can be loaded when operators run on parallel instances. Default is character(0).
The $packages field will reflect this values.

packages (character) Packages that need to be loaded for the operator to function. This should be declared so these packages can be loaded when operators run on parallel instances. Default is character(0).
The $packages field will reflect this values.

dict_entry (character(1) | NULL)
Key of the class inside the Dictionary (usually one of dict_mutators, dict_recombinators, dict_selectors), where it can be retrieved using a short access function. May be NULL if the operator is not entered in a dictionary.
The $dict_entry field will reflect this value.

dict_entry (character(1) | NULL)
Key of the class inside the Dictionary (usually one of dict_mutators, dict_recombinators, dict_selectors), where it can be retrieved using a short access function. May be NULL if the operator is not entered in a dictionary.
The $dict_entry field will reflect this value.

Method prime()**:** See MiesOperator method. Primes both this operator, as well as the wrapped operator given to surrogate_selector during construction.

Usage:

```
FiltorSurrogate$prime(param_set)
```

Arguments:

param_set (ParamSet)
Passed to MiesOperator$prime().

Returns: invisible self.

Method clone()**:** The objects of this class are cloneable with this method.

Usage:

```
FiltorSurrogate$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other base classes: Filtor, MiesOperator, MutatorDiscrete, MutatorNumeric, Mutator, OperatorCombination, RecombinatorPair, Recombinator, Scalor, SelectorScalar, Selector

Other filtors: Filtor, dict_filtors_maybe, dict_filtors_null, dict_filtors_proxy, dict_filtors_surprog, dict_filtors_surtour

---

MiesOperator *Operator Base Class*

Description

Base class representing MIES-operators: Recombinator, Mutator, and Selector.

Operators perform a specific function within ES algorithms, and by exchanging them, the character of ES algorithms can be modified. Operators operate on collections of individuals and return modified individuals (mutated or recombined) or indices of selected individuals. Operators can be combined using MutatorCombination / RecombinatorCombination and other operators wrappers.

Before applying operators, they have to be *primed* for the domain of the individuals which they are operating on; this is done using the `$prime()` function. Afterwards, the `$operate()` function may be called with a `data.frame` of individuals that fall into this domain. `$operate()` may be called multiple times after priming, and a once primed operator can be primed again for a different domain by calling `$prime()` agian (which forgets the old priming).

Inheriting

`MiesOperator` is an abstract base class and should be inherited from. Inheriting classes should implement the private `$.operate()` function. The user of the object calls `$operate()`, and the arguments are passed on to private `$.operate()` after checking that the operator is primed, and that the `values` argument conforms to the primed domain. Typically, the `$initialize()` and `$prime()` functions are also overloaded, but should call their `super` equivalents.

In most cases, the `MiesOperator` class should not be inherited from, directly; instead, the operator classes (Recombinator, Mutator, Selector) or their subclasses should be inherited.

Active bindings

`param_set` (ParamSet)
: Configuration parameters of the `MiesOperator` object. Read-only.

`param_classes` (`character`)
: Classes of parameters that the operator can handle, contains any of `"ParamLgl"`, `"ParamInt"`, `"ParamDbl"`, `"ParamFct"`. Read-only.

`packages` (`character`)
: Packages needed for the operator. Read-only.

`dict_entry` (`character(1)` | `NULL`)
: Key of this class in its respective Dictionary. Is `NULL` if this class it not (known to be) in a Dictionary. Read-only.

`dict_shortaccess` (character(1) | NULL)
: Name of Dictionary short-access function where an object of this class can be retrieved. Is NULL if this class is not (known to be) in a Dictionary with a short-access function. Read-only.

`endomorphism` (logical(1))
: Whether the output of $operate() is a data.frame / data.table in the same domain as its input. Read-only.

`primed_ps` (ParamSet | NULL)
: ParamSet on which the MiesOperator is primed. Is NULL if it has not been primed. Writing to this acrive binding calls $prime().

`is_primed` (logical(1))
: Whether the MiesOperator was primed before. Is FALSE exactly when $primed_ps is NULL. Read-only.

`man` (character(1))
: Name of this class, in the form <package>::<classname>. Used by the $help() method.

Methods

Public methods:

- MiesOperator$new()
- MiesOperator$repr()
- MiesOperator$print()
- MiesOperator$prime()
- MiesOperator$operate()
- MiesOperator$help()
- MiesOperator$clone()

Method `new()`**:** Initialize base class components of the MiesOperator.

Usage:

```
MiesOperator$new(
  param_classes = c("ParamLgl", "ParamInt", "ParamDbl", "ParamFct"),
  param_set = ps(),
  packages = character(0),
  dict_entry = NULL,
  dict_shortaccess = NULL,
  own_param_set = quote(self$param_set),
  endomorphism = TRUE
)
```

Arguments:

`param_classes` (character)
: Classes of parameters that the operator can handle. May contain any of "ParamLgl", "ParamInt", "ParamDbl", "ParamFct". Default is all of them.
The $param_classes field will reflect this value.

`param_set` (ParamSet | list of expression)
: Strategy parameters of the operator. This should be created by the subclass and given to

super$initialize(). If this is a ParamSet, it is used as the MiesOperator's ParamSet directly. Otherwise it must be a list of expressions e.g. created by alist() that evaluate to ParamSets, possibly referencing self and private. These ParamSet are then combined using a ParamSetCollection. Default is the empty ParamSet.
The $param_set field will reflect this value.

packages (character) Packages that need to be loaded for the operator to function. This should be declared so these packages can be loaded when operators run on parallel instances. Default is character(0).
The $packages field will reflect this values.

dict_entry (character(1) | NULL)
Key of the class inside the Dictionary (usually one of dict_mutators, dict_recombinators, dict_selectors), where it can be retrieved using a short access function. May be NULL if the operator is not entered in a dictionary.
The $dict_entry field will reflect this value.

dict_shortaccess (character(1) | NULL)
Name of the Dictionary short access function in which the operator is registered. This is used to inform the user about how to construct a given object. Should ordinarily be one of "mut", "rec", "sel".
The $dict_shortaccess field will reflect this value.

own_param_set (language)
An expression that evaluates to a ParamSet indicating the configuration parameters that are entirely owned by this operator class (and not proxied from a construction argument object). This should be quote(self$param_set) (the default) when the param_set argument is not a list of expressions.

endomorphism (logical(1))
Whether the private $.operate() operation creates a data.table with the same columns as the input (i.e. conforming to the primed ParamSet). If this is TRUE (default), then the return value of $.operate() is checked for this and columns are put in the correct order.
The $endomorphsim field will reflect this value.

Method repr()**:** Create a call object representing this operator.

Usage:

```
MiesOperator$repr(
  skip_defaults = TRUE,
  show_params = TRUE,
  show_constructor_args = TRUE,
  ...
)
```

Arguments:

skip_defaults (logical(1))
Whether to skip construction arguments that have their default value. Default TRUE.

show_params (logical(1))
Whether to show ParamSet values. Default TRUE.

show_constructor_args (logical(1))
Whether to show construction args that are not ParamSet values. Default TRUE.

... (any)
Ignored.

Method `print()`**:** Print this operator.

Usage:

```
MiesOperator$print(verbose = FALSE, ...)
```

Arguments:

`verbose` (`logical(1)`)
Whether to show all construction arguments, even the ones at default values. Default `FALSE`.

`...` (any)
Ignored.

Method `prime()`**:** Prepare the `MiesOperator` to function on the given `ParamSet`. This must be called before `$operate()`. It may be called multiple times in the lifecycle of the `MiesOperator` object, and prior primings are forgotten when priming on a new `ParamSet`. The `ParamSet` on which the `MiesOperator` was last primed can be read from `$primed_ps`.

Usage:

```
MiesOperator$prime(param_set)
```

Arguments:

`param_set` (`ParamSet`)
The `ParamSet` to which all `values` tables passed to `$operate()` will need to conform to. May only contiain `Param` objects that conform to the classes listed in `$param_classes`.

Returns: invisible `self`.

Method `operate()`**:** Operate on the given individuals. This calls private `$.operate()`, which must be overloaded by an inheriting class, passing through all function arguments after performing some checks.

Usage:

```
MiesOperator$operate(values, ...)
```

Arguments:

`values` (`data.frame`)
Individuals to operate on. Must pass the check of the `Param` given in the last `$prime()` call and may not have any missing components.

`...` (any)
Depending on the concrete class, passed on to `$.operate()`.

Returns: `data.frame`: the result of the operation. If the input was a `data.table` instead of a `data.frame`, the output is also `data.table`.

Method `help()`**:** Run `utils::help()` for this object.

Usage:

```
MiesOperator$help(help_type = getOption("help_type"))
```

Arguments:

`help_type` (`character(1)`)
One of `"text"`, `"html"`, or `"pdf"`: The type of help page to open. Defaults to the `"help_type"` option.

Returns: `help_files_with_dopic` object, which opens the help page.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MiesOperator$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other base classes: `FiltorSurrogate`, `Filtor`, `MutatorDiscrete`, `MutatorNumeric`, `Mutator`, `OperatorCombination`, `RecombinatorPair`, `Recombinator`, `Scalor`, `SelectorScalar`, `Selector`

`mies_aggregate_generations` *Get Aggregated Performance Values by Generation*

Description

Get evaluated performance values from an `OptimInstance` aggregated for each generation. This may either concern all individuals that were alive at the end of a given generation (`survivors_only` `TRUE`) or at any point during a generation (`survivors_only` `FALSE`).

The result is a single `data.table` object with a dob column indicating the generation, as well as one column for each `aggregations` entry crossed with each objective of `inst`.

Usage

```
mies_aggregate_generations(
  inst,
  objectives = inst$archive$codomain$ids(),
 aggregations = list(min = min, mean = mean, max = max, median = stats::median, size =
    length),
  as_fitnesses = TRUE,
  survivors_only = TRUE,
  condition_on_budget_id = NULL
)
```

Arguments

`inst`	(`OptimInstance`) Optimization instance to evaluate.
`objectives`	(`character`) Objectives for which to calculate aggregates. Must be a subset of the codomain elements of `inst`, but when `as_fitnesses` is `TRUE`, elements that are neither being minimized nor maximized are ignored.

`aggregations` (named `list` of `function`)
List containing aggregation functions to be evaluated on a vector of objective falues for each generation. These functions should take a single argument and return a scalar value.

`as_fitnesses` (`logical(1)`)
Whether to transform performance values into "fitness" values that are always to be maximized. This means that values that objectives that should originally be minimized are multiplied with -1, and that parts of the objective codomain that are neither being minimized nor maximized are dropped. Default `TRUE`.

`survivors_only` (`logical(1)`)
Whether to ignore configurations that have `"eol"` set to the given generation, i.e. individuals that were killed during that generation. When this is `TRUE` (default), then only individuals that are alive at the *end* of a generation are considered; otherwise all individuals alive at any point of a generation are considered. If it is `TRUE`, this leads to individuals that have `"dob"` == `"eol"` being ignored.

`condition_on_budget_id`
(`character(1)` | `NULL`)
Budget component when doing multi-fidelity optimization. When this is given, then for each generation, only individuals with the highest value for this component are considered. If `survivors_only` is `TRUE`, this means the highest value of all survivors of a given generation, if it is `FALSE`, then it is the highest value of all individuals alive at any point of a generation. To ignore possible budget-parameters, set this to `NULL` (default). This is inparticular necessary when fidelity is not monotonically increasing (e.g. if it is categorical).

Value

a `data.table` with the column `"dob"`, indicating the generation, as well as further columns named by the items in `aggregations`. There is more on element in `objectives` (or more than one element not being minimized/maximized when `as_fitnesses` is `TRUE`), then columns are named `<aggregations element name>.<objective name>`. Otherwise, they are named by `<aggregations element name>` only. To get a guarantee that elements are only named after elements in `aggregations`, set `objectives` to a length 1 `character`.

Examples

```
library("bbotk")
lgr::threshold("warn")

# Define the objective to optimize
objective <- ObjectiveRFun$new(
  fun = function(xs) {
    z <- 10 - exp(-xs$x^2 - xs$y^2) + 2 * exp(-(2 - xs$x)^2 - (2 - xs$y)^2)
    list(Obj = z)
  },
  domain = ps(x = p_dbl(-2, 4), y = p_dbl(-2, 4)),
  codomain = ps(Obj = p_dbl(tags = "minimize"))
)
```

```
oi <- OptimInstanceSingleCrit$new(objective,
  terminator = trm("evals", n_evals = 6)
)

op <- opt("mies",
  lambda = 2, mu = 2,
  mutator = mut("gauss", sdev = 0.1),
  recombinator = rec("xounif"),
  parent_selector = sel("best")
)
set.seed(1)
op$optimize(oi)

# negates objectives that are minimized:
mies_aggregate_generations(oi)

# silly aggregation: first element
mies_aggregate_generations(oi, aggregations = list(first = function(x) x[1]))

# real objective values:
mies_aggregate_generations(oi, as_fitnesses = FALSE)

# Individuals that died are included:
mies_aggregate_generations(oi, survivors_only = FALSE)
```

---

mies_evaluate_offspring   *Evaluate Proposed Configurations Generated in a MIES Iteration*

Description

Calls `$eval_batch` of a given `OptimInstance` on a set of configurations as part of a MIES operation. The dob extra-info in the archive is also set properly to indicate a progressed generation.

This function can be used directly, but it is easier to use it within the `OptimizerMies` `Optimizer` if standard GA operation is desired.

Multifidelity evaluation is supported as described in `vignette("mies-multifid")`. For this, an extra component named after `budget_id` is appended to each individual, chosen from the `fidelity` argument and depending on the value of `survivor_budget`. `budget_id` should have the same values as given to the other `mies_*` functions.

Usage

```
mies_evaluate_offspring(
  inst,
  offspring,
  budget_id = NULL,
  fidelity = NULL,
  reevaluate_fidelity = NULL,
  fidelity_monotonic = TRUE
)
```

Arguments

inst
: (OptimInstance)
  Optimization instance to evaluate.

offspring
: (data.frame)
  Proposed configurations to be evaluated, must have columns named after the inst's search space, minus budget_id if not NULL.

budget_id
: (character(1) | NULL)
  Budget component when doing multi-fidelity optimization. This component of the search space is added to individuals according to fidelity. Should be NULL when no multi-fidelity optimization is performed (default).

fidelity
: (atomic(1) | NULL)
  Atomic scalar indicating the value to be assigned to the budget_id component of offspring. This value must be NULL if no multi-fidelity optimization is performed (the default).

reevaluate_fidelity
: (atomic(1))
  Fidelity with which to evaluate alive individuals from previous generations that have a budget value below (if fidelity_monotonic is TRUE) or different from the current fidelity value. Default NULL: Do not re-evaluate. Must be NULL when budget_id and fidelity are NULL. See also mies_step_fidelity.

fidelity_monotonic
: (logical(1))
  When reevaluate_fidelity is non-NULL, then this indicates whether individuals should only ever be re-evaluated when fidelity would be increased. Default TRUE. Ignored when reevaluate_fidelity is NULL

Value

invisible data.table: the performance values returned when evaluating the offspring values through eval_batch.

See Also

Other mies building blocks: mies_generate_offspring(), mies_get_fitnesses(), mies_init_population(), mies_select_from_archive(), mies_step_fidelity(), mies_survival_comma(), mies_survival_plus()

Examples

```
library("bbotk")
lgr::threshold("warn")

# Define the objective to optimize
objective <- ObjectiveRFun$new(
  fun = function(xs) {
    z <- exp(-xs$x^2 - xs$y^2) + 2 * exp(-(2 - xs$x)^2 - (2 - xs$y)^2)
    list(Obj = z)
  },
  domain = ps(x = p_dbl(-2, 4), y = p_dbl(-2, 4)),
```

```
  codomain = ps(Obj = p_dbl(tags = "maximize"))
)

# Get a new OptimInstance
oi <- OptimInstanceSingleCrit$new(objective,
  terminator = trm("evals", n_evals = 100)
)

mies_init_population(inst = oi, mu = 3)
# Initial state:
oi$archive

# 'offspring' is just a data.frame of values to evaluate.
# In general it should be created using 'mies_generate_offspring()'.
offspring = data.frame(x = 1:2, y = 2:1)

mies_evaluate_offspring(oi, offspring = offspring)

# This evaluated the given points and assigned them 'dob' 2.
oi$archive

# Note that at this point one would ordinarily call a 'mies_survival_*()'
# function.

###
# Advanced demo, making use of additional components and doing multi-fidelity
##

# declare 'y' the budget parameter. It will not be in the 'offspring'
# table any more.
budget_id = "y"
# but: offspring may contain any other value that is appended to 'oi'. These
# are ignored by the objective.
offspring = data.frame(x = 0:1, z = 3)

mies_evaluate_offspring(oi, offspring = offspring, budget_id = budget_id,
  fidelity = 1)

# This now has the additional column 'z'. Values of y for the new evaluations
# are 1.
oi$archive

offspring = data.frame(x = 2, z = 3)
# Increasing the fidelity will not cause re-evaluation of existing individuals
# when `reevaluate_fidelity` is not given.
mies_evaluate_offspring(oi, offspring = offspring, budget_id = budget_id,
  fidelity = 2)
oi$archive

offspring = data.frame(x = 3, z = 3)
# Depending on the effect of fidelity, this may however have a biasing effect,
# so it may be desirable to re-evaluate surviving individuals from previous
# generations. The 'reevaluate_fidelity' may even be different from 'fidelity'
```

```
mies_evaluate_offspring(oi, offspring = offspring, budget_id = budget_id,
  fidelity = 3, reevaluate_fidelity = 2)

# In this example, only individuals with 'y = 1' were re-evaluated, since
# 'fidelity_monotonic' is TRUE.
oi$archive
```

---

mies_filter_offspring *Filter Offspring*

Description

Uses a `Filtor` to extract a subset of individuals from a given set. The individuals are either returned directly (when `get_indivs` is `TRUE`) or in form of an index into the given individuals (when `get_indivs` is `FALSE`).

`Filtor`s must always select individuals without replacement, so selected individual indices are unique.

Usage

```
mies_filter_offspring(
  inst,
  individuals,
  lambda,
  filtor = NULL,
  budget_id = NULL,
  fidelity = NULL,
  get_indivs = TRUE
)
```

Arguments

`inst`	(`OptimInstance`) Optimization instance to evaluate.
`individuals`	(`data.frame` | `data.table`) Individuals to filter. Must have columns according to `filtor$primed_ps`, and must have at least `filtor$needed_input(lambda)` rows.
`lambda`	(`integer(1)`) Number of individuals to filter down to.
`filtor`	(`Filtor` | `NULL`) `Filtor` operator that filters. When `NULL` is given, then the `FiltorNull` operation is performed and the first `lambda` individuals are taken from `individuals`.
`budget_id`	(`character(1)` | `NULL`) Budget component when doing multi-fidelity optimization. This component of the search space is added to `individuals` according to `fidelity`. Should be `NULL` when no multi-fidelity optimization is performed (default).

`fidelity`	(`atomic` | `NULL`) scalar indicating the value of the `budget_id` component with which to evaluate individuals to be filtered. This value must be `NULL` when no multi-fidelity optimization is performed, but it may **also** be `NULL` when the maximum value of the `budget_id` found in `inst$archive` should be used (the default).
`get_indivs`	(`logical(1)`) Whether to return the `data.frame` or `data.table` of selected individuals, or an index into `individuals`.

Value

If `get_indivs` is `TRUE`: a `data.frame` or `data.table` (depending on the input type of `individuals`) of filtered configurations. If `get_indivs` is `FALSE`: an `integer` vector indexing the filtered individuals.

---

mies_generate_offspring

Generate Offspring Through Mutation and Recombination

Description

Generate new proposal individuals to be evaluated using `mies_evaluate_offspring()`.

Parent individuals are selected using `parent_selector`, then mutated using `mutator`, and thend recombined using `recombinator`. If only a subset of these operations is desired, then it is possible to set `mutator` or `recombinator` to the respective "null"-operators.

Usage

```
mies_generate_offspring(
  inst,
  lambda,
  parent_selector = NULL,
  mutator = NULL,
  recombinator = NULL,
  budget_id = NULL
)
```

Arguments

`inst`	(`OptimInstance`) Optimization instance to evaluate.
`lambda`	(`integer(1)`) Number of new individuals to generate. This is not necessarily the number with which `parent_selector` gets called, because `recombinator` could in principle need more than `lambda` input individuals to generate `lambda` output individuals.

parent_selector
: (Selector | NULL)
Selector operator that selects parent individuals depending on configuration values and objective results. When parent_selector$operate() is called, then objectives that are being minimized are multiplied with -1 (through mies_get_fitnesses()), since Selectors always try to maximize fitness. When this is NULL (default), then a SelectorBest us used.
The Selector must be primed on a superset of inst$search_space; this *includes* the "budget" component when performing multi-fidelity optimization. All components on which selector is primed on must occur in the archive.
The given Selector *may* return duplicates.

mutator
: (Mutator | NULL)
Mutator operation to apply to individuals selected out of inst using parent_selector.
The Mutator must be primed on a ParamSet similar to inst$search_space, but *without* the "budget" component when budget_id is given (multi-fidelity optimization). Such a ParamSet can be generated for example using mies_prime_operators.
When this is NULL (default), then a MutatorNull is used, effectively disabling mutation.

recombinator
: (Recombinator | NULL)
Recombinator operation to apply to individuals selected out of int using parent_selector after mutation using mutator. The Recombinator must be primed on a ParamSet similar to inst$search_space, but *without* the "budget" component when budget_id is given (multi-fidelity optimization). Such a ParamSet can be generated for example using mies_prime_operators.
When this is NULL (default), then a RecombinatorNull is used, effectively disabling recombination.

budget_id
: (character(1) | NULL)
Budget compnent when doing multi-fidelity optimization. This component of the search space is removed from individuals sampled from the archive in inst before giving it to mutator and recombinator. Should be NULL when not doing multi-fidelity.

Value

data.table: A table of configurations proposed as offspring to be evaluated using mies_evaluate_offspring().

See Also

Other mies building blocks: mies_evaluate_offspring(), mies_get_fitnesses(), mies_init_population(), mies_select_from_archive(), mies_step_fidelity(), mies_survival_comma(), mies_survival_plus()

Examples

```
set.seed(1)

library("bbotk")
lgr::threshold("warn")

# Define the objective to optimize
```

```
objective <- ObjectiveRFun$new(
  fun = function(xs) {
    z <- exp(-xs$x^2 - xs$y^2) + 2 * exp(-(2 - xs$x)^2 - (2 - xs$y)^2)
    list(Obj = z)
  },
  domain = ps(x = p_dbl(-2, 4), y = p_dbl(-2, 4)),
  codomain = ps(Obj = p_dbl(tags = "maximize"))
)

# Get a new OptimInstance
oi <- OptimInstanceSingleCrit$new(objective,
  terminator = trm("evals", n_evals = 100)
)

# Demo operators
m = mut("gauss", sdev = 0.1)
r = rec("xounif")
s = sel("random")
# Operators must be primed
mies_prime_operators(objective$domain, list(m), list(r), list(s))

# We would normally call mies_init_population, but for reproducibility
# we are going to evaluate three given points

oi$eval_batch(data.table::data.table(x = 0:2, y = 2:0, dob = 1, eol = NA_real_))

# Evaluated points:
oi$archive

# Use default operators: no mutation, no recombination, parent_selctor is
# sel("best") --> get one individual, the one with highest performance in the
# archive (x = 1, y = 1).
# (Note 'mies_generate_offspring()' does not modify 'oi')
mies_generate_offspring(oi, lambda = 1)

# Mutate the selected individual after selection. 'm' has 'sdev' set to 0.1,
# so the (x = 1, y = 1) is slightly permuted.
mies_generate_offspring(oi, lambda = 1, mutator = m)

# Recombination, then mutation.
# Even though lambda is 1, there will be two individuals selected with
# sel("best") and recombined, because rec("xounif") needs two parents. One
# of the crossover results is discarded (respecting that 'lambda' is 1),
# the other is mutated and returned.
mies_generate_offspring(oi, lambda = 1, mutator = m, recombinator = r)

# General application: select, recombine, then mutate.
mies_generate_offspring(oi, lambda = 5, parent_selector = s, mutator = m, recombinator = r)
```

mies_generation *Get the Last Generation that was Evaluated*

Description

Gets the last generation that was evaluated as counted by the "dob" column in the OptimInstance's Archive.

This accepts OptimInstances that were not evaluated with miesmuschel and are therefore missing the "dob" column, returning a value of 0. However, if the "dob" column is invalid (the inferred generation is not integer numeric or not non-negative), an error is thrown.

Usage

```
mies_generation(inst)
```

Arguments

inst	(OptimInstance) Optimization instance to evaluate.

Value

a scalar integer value indicating the last generation that was evaluated in inst. It is 0 when inst is empty, and also typically 0 if all evaluations in inst so far were performed outside of miesmuschel. Every call of mies_init_population that actually performs evaluations, as well as each call to mies_evaluate_offspring with non-empty offspring, increases the generation by 1.

Examples

```
library("bbotk")
lgr::threshold("warn")

# Define the objective to optimize
objective <- ObjectiveRFun$new(
  fun = function(xs) {
    z <- 10 - exp(-xs$x^2 - xs$y^2) + 2 * exp(-(2 - xs$x)^2 - (2 - xs$y)^2)
    list(Obj = z)
  },
  domain = ps(x = p_dbl(-2, 4), y = p_dbl(-2, 4)),
  codomain = ps(Obj = p_dbl(tags = "minimize"))
)

oi <- OptimInstanceSingleCrit$new(objective,
  terminator = trm("evals", n_evals = 6)
)

op <- opt("mies",
  lambda = 2, mu = 2,
  mutator = mut("gauss", sdev = 0.1),
  recombinator = rec("xounif"),
  parent_selector = sel("best")
)
set.seed(1)

mies_generation(oi)
```

```
op$optimize(oi)
mies_generation(oi)

oi$terminator = trm("evals", n_evals = 10)

op$optimize(oi)
mies_generation(oi)
```

---

mies_get_fitnesses *Get Fitness Values from OptimInstance*

Description

Get fitness values in the correct form as used by Selector operators from an OptimInstance. This works for both single-criterion and multi-criterion optimization, and entails multiplying objectives with -1 if they are being minimized, since Selector tries to maximize fitness.

Usage

```
mies_get_fitnesses(inst, rows)
```

Arguments

inst	(OptimInstance) Optimization instance to evaluate.
rows	optional (integer) Indices of rows within inst to consider. If this is not given, then the entire archive is used.

Value

numeric matrix with length(rows) (if rows is given, otherwise nrow(inst$archive$data)) rows and one column for each objective: fitnesses to be maximized.

See Also

Other mies building blocks: mies_evaluate_offspring(), mies_generate_offspring(), mies_init_population(), mies_select_from_archive(), mies_step_fidelity(), mies_survival_comma(), mies_survival_plus()

Examples

```
set.seed(1)
library("bbotk")
lgr::threshold("warn")

# Define the objective to optimize
objective <- ObjectiveRFun$new(
  fun = function(xs) {
```

```
    z <- exp(-xs$x^2 - xs$y^2) + 2 * exp(-(2 - xs$x)^2 - (2 - xs$y)^2)
    list(Obj = z)
  },
  domain = ps(x = p_dbl(-2, 4), y = p_dbl(-2, 4)),
  codomain = ps(Obj = p_dbl(tags = "maximize"))
)

# Get a new OptimInstance
oi <- OptimInstanceSingleCrit$new(objective,
  terminator = trm("evals", n_evals = 100)
)

mies_init_population(inst = oi, mu = 3)

oi$archive

mies_get_fitnesses(oi, c(2, 3))

###
# Multi-objective, and automatic maximization:
objective2 <- ObjectiveRFun$new(
  fun = function(xs) list(Obj1 = xs$x^2, Obj2 = -xs$y^2),
  domain = ps(x = p_dbl(-2, 4), y = p_dbl(-2, 4)),
  codomain = ps(
    Obj1 = p_dbl(tags = "minimize"),
    Obj2 = p_dbl(tags = "maximize")
  )
)
# Using MultiCrit!
oi <- OptimInstanceMultiCrit$new(objective2,
  terminator = trm("evals", n_evals = 100)
)

mies_init_population(inst = oi, mu = 3)

oi$archive

# Note Obj1 has a different sign than in the archive.
mies_get_fitnesses(oi, c(2, 3))
```

---

mies_get_generation_results
Get Performance Values by Generation

Description

Get evaluated performance values from an OptimInstance for all individuals that were alive at a given generation. Depending on `survivors_only`, all individuals alive at the *end* of a generation are returned, or all individuals alive at any point during a generation.

The resulting data.table object is formatted for easy manipulation to get relevant information about optimization progress. To get aggregated values per generation, use by = "dob".

Usage

```
mies_get_generation_results(
  inst,
  as_fitnesses = TRUE,
  survivors_only = TRUE,
  condition_on_budget_id = NULL
)
```

Arguments

inst
: (OptimInstance)
Optimization instance to evaluate.

as_fitnesses
: (logical(1))
Whether to transform performance values into "fitness" values that are always to be maximized. This means that values that objectives that should originally be minimized are multiplied with -1, and that parts of the objective codomain that are neither being minimized nor maximized are dropped. Default TRUE.

survivors_only
: (logical(1))
Whether to ignore configurations that have "eol" set to the given generation, i.e. individuals that were killed during that generation. When this is TRUE (default), then only individuals that are alive at the *end* of a generation are considered; otherwise all individuals alive at any point of a generation are considered. If it is TRUE, this leads to individuals that have "dob" == "eol" being ignored.

condition_on_budget_id
: (character(1) | NULL)
Budget component when doing multi-fidelity optimization. When this is given, then for each generation, only individuals with the highest value for this component are considered. If survivors_only is TRUE, this means the highest value of all survivors of a given generation, if it is FALSE, then it is the highest value of all individuals alive at any point of a generation. To ignore possible budget-parameters, set this to NULL (default). This is inparticular necessary when fidelity is not monotonically increasing (e.g. if it is categorical).

Value

a data.table with the column "dob", indicating the generation, as well as further columns named by the OptimInstance's objectives.

Examples

```
library("bbotk")
lgr::threshold("warn")

# Define the objective to optimize
objective <- ObjectiveRFun$new(
```

```
  fun = function(xs) {
    z <- 10 - exp(-xs$x^2 - xs$y^2) + 2 * exp(-(2 - xs$x)^2 - (2 - xs$y)^2)
    list(Obj = z)
  },
  domain = ps(x = p_dbl(-2, 4), y = p_dbl(-2, 4)),
  codomain = ps(Obj = p_dbl(tags = "minimize"))
)

oi <- OptimInstanceSingleCrit$new(objective,
  terminator = trm("evals", n_evals = 6)
)

op <- opt("mies",
  lambda = 2, mu = 2,
  mutator = mut("gauss", sdev = 0.1),
  recombinator = rec("xounif"),
  parent_selector = sel("best")
)
set.seed(1)
op$optimize(oi)

# negates objectives that are minimized:
mies_get_generation_results(oi)

# real objective values:
mies_get_generation_results(oi, as_fitnesses = FALSE)

# Individuals that died are included:
mies_get_generation_results(oi, survivors_only = FALSE)
```

---

mies_init_population *Initialize MIES Optimization*

Description

Set up an OptimInstance for MIES optimization. This adds the dob and eol columns to the instance's archive, and makes sure there are at least mu survivors (i.e. entries with eol set to NA) present. If there are already >= mu prior evaluations present, then the last mu of these remain alive (the other's eol set to 0); otherwise, up to mu new randomly sampled configurations are evaluated and added to the archive and have eol set to NA.

Usage

```
mies_init_population(
  inst,
  mu,
  initializer = generate_design_random,
  survival_selector = SelectorBest$new()$prime(inst$search_space),
  budget_id = NULL,
```

```
  fidelity = NULL,
  fidelity_new_individuals_only = FALSE,
  fidelity_monotonic = TRUE,
  additional_component_sampler = NULL
)
```

Arguments

`inst`
: (`OptimInstance`)
Optimization instance to evaluate.

`mu`
: (`integer(1)`)
Population target size, non-negative integer.

`initializer`
: (`function`)
Function that generates a `Design` object, with arguments `param_set` and `n`, functioning like `paradox::generate_design_random` or `paradox::generate_design_lhs`. Note that `paradox::generate_design_grid` can not be used and must be wrapped with a custom function that ensures that only `n` individuals are produced. The generated design must correspond to the `inst`'s `$search_space`; for components that are not in the objective's search space, the `additional_component_sampler` is used.

`survival_selector`
: (`Selector`)
Used when the given `OptimInstance` already contains more individuals than `mu`.
`Selector` operator that selects surviving individuals depending on configuration values and objective results, When `survival_selector$operate()` is called, then objectives that are being minimized are multiplied with -1 (through `mies_get_fitnesses`), since `Selector`s always try to maximize fitness.
The `Selector` must be primed on `inst$search_space`; this *includes* the "budget" component when performing multi-fidelity optimization. Default is `SelectorBest`.
The given `Selector` may *not* return duplicates.

`budget_id`
: (`character(1)` | `NULL`)
Budget component when doing multi-fidelity optimization. This component of the search space is added to `individuals` according to `fidelity`. Should be `NULL` when no multi-fidelity optimization is performed (default).

`fidelity`
: (`atomic(1)` | `NULL`)
Atomic scalar indicating the value to be assigned to the `budget_id` component of offspring. This value must be `NULL` if no multi-fidelity optimization is performed (the default).

`fidelity_new_individuals_only`
: (`logical(1)`)
When `fidelity` is not `NULL`: Whether to re-evaluate individuals that are already present in `inst` should they have a smaller (if `fidelity_monotonic` is `TRUE`) or different (if `fidelity_monotonic` is `FALSE`) value from the one given to `fidelity`. Default `FALSE`. Ignored when `fidelity` is `NULL`.

`fidelity_monotonic`
: (`logical(1)`)
Whether to only re-evaluate configurations for which the fidelity would increase. Default `TRUE`. Ignored when `fidelity` is `NULL` or when `fidelity_new_individuals_only` is `TRUE`.

`additional_component_sampler`
: (`Sampler` | `NULL`)
`Sampler` for components of individuals that are not part of `inst`'s `$search_space`. These components are never used for performance evaluation, but they may be useful for self-adaptive `OperatorCombination`s. See the description of `mies_prime_operators()` on how operators need to be primed to respect additional components.
It is possible that `additional_component_sampler` is used for *more* rows than `initializer`, which happens when the `inst`'s `$archive` contains prior evaluations that are alive, but does not contain columns pertaining to additional columns, or contains *all* these columns but there are rows that are `NA` valued. If only *some* of the columns are present, or if all these columns are present but there are rows that are only `NA` valued for some columns, then an error is thrown.
Default is `NULL`: no additional components.

Value

invisible `OptimInstance`: the input instance, modified by-reference.

See Also

Other mies building blocks: `mies_evaluate_offspring()`, `mies_generate_offspring()`, `mies_get_fitnesses()`, `mies_select_from_archive()`, `mies_step_fidelity()`, `mies_survival_comma()`, `mies_survival_plus()`

Examples

```
library("bbotk")
lgr::threshold("warn")

# Define the objective to optimize
objective <- ObjectiveRFun$new(
  fun = function(xs) {
    z <- exp(-xs$x^2 - xs$y^2) + 2 * exp(-(2 - xs$x)^2 - (2 - xs$y)^2)
    list(Obj = z)
  },
  domain = ps(x = p_dbl(-2, 4), y = p_dbl(-2, 4)),
  codomain = ps(Obj = p_dbl(tags = "maximize"))
)

# Get a new OptimInstance
oi <- OptimInstanceSingleCrit$new(objective,
  terminator = trm("evals", n_evals = 100)
)

mies_init_population(inst = oi, mu = 3)

# 3 evaluations, archive contains 'dob' and 'eol'
```

```
oi$archive

###
# Advanced demo, making use of additional components and fidelity
##

# Get a new OptimInstance
oi <- OptimInstanceSingleCrit$new(objective,
  terminator = trm("evals", n_evals = 100)
)

mies_init_population(inst = oi, mu = 3, budget_id = "y", fidelity = 2,
  additional_component_sampler = Sampler1DRfun$new(
    param = ParamDbl$new("additional", -1, 1), rfun = function(n) rep(-1, n)
  )
)

# 3 evaluations. We also have 'additional', sampled from rfun (always -1),
# which is ignored by the objective. Besides, we have "y", which is 2,
# according to 'fidelity'.
oi$archive
```

mies_prime_operators *Prime MIES Operators*

Description

Prime the given MiesOperators for an optimization run with the given search space.

In its simplest form, MIES optimization only optimizes the search space of the Objective to be optimized. However, more advanced optimization may handle a "budget" parameter for multi-fidelity optimization differently: It is still selected by Selectors, but not mutated or recombined and instead handled separately. It is also possible to add additional components to the search space that are not evaluated by the objective function, but that are used for self-adaption by other operators.

The mies_prime_operators() function uses the information that the user usually has readily at hand – the Objectives search space, the budget parameter, and additional components -- and primes [Mutator] objects in the right way:

- Selectors are primed on a union of search_space and additional_components
- Mutators and Recombinators are primed on the Selector's space with the budget_id Param removed.

mies_prime_operators() is called with an arbitrary number of MiesOperator arguments; typically one Mutator, one Recombinator and at least two Selector: one for survival selection, and one parent selection. Supplied MiesOperators are primed by-reference, but they are also returned as invisible list.

If neither additional components nor multi-fidelity optimization is used, it is also possible to use the $prime() function of hte MiesOperators directly, although using mies_prime_operators() gives flexibility for future extension.

Usage

```
mies_prime_operators(
  search_space,
  mutators = list(),
  recombinators = list(),
  selectors = list(),
  filtors = list(),
  ...,
  additional_components = NULL,
  budget_id = NULL
)
```

Arguments

`search_space` (ParamSet)
Search space of the Objective or OptimInstance to be optimized.

`mutators` (list of Mutator)
Mutator objects to prime. May be empty (default).

`recombinators` (list of Recombinator)
Recombinator objects to prime. May be empty (default).

`selectors` (list of Selector)
Selector objects to prime. May be empty (default).

`filtors` (list of Filtor)
Filtor objects to prime. May be empty (default).

`...` (any)
Must not be given. Other operators may be added in the future, so the following arguments should be passed by name.

`additional_components`
(ParamSet | NULL)
Additional components to optimize over, not included in `search_space`, but possibly used for self-adaption. This must be the ParamSet of `mies_init_population()`'s `additional_component_sampler` argument.

`budget_id` (`character(1)` | NULL)
Budget component used for multi-fidelity optimization.

Value

`invisible` named `list` with entries `$mutators` (`list` of Mutator, primed `mutators`), `$recombinators` (`list` of Recombinator, primed `recombinators`), and `$selectors` (`list` of Selector, primed `selectors`).

Examples

```
# Search space of a potential TuningInstance for optimization:
search_space = ps(x = p_dbl(), y = p_dbl())
# Additoinal search space components that are not part of the TuningInstance
additional_components = ps(z = p_dbl())
```

```
# Budget parameter not subject to mutation or recombination
budget_id = "y"

m = mut("gauss")
r = rec("xounif")
s1 = sel("best")
s2 = sel("random")
f = ftr("null")

mies_prime_operators(search_space, mutators = list(m),
  recombinators = list(r), selectors = list(s1, s2), filtors = list(f),
  additional_components = additional_components, budget_id = budget_id
)

# contain search_space without budget parameter, with additional_components
m$primed_ps
r$primed_ps

# contain also the budget parameter
s1$primed_ps
s2$primed_ps
f$primed_ps
```

---

mies_select_from_archive

Select Individuals from an OptimInstance

Description

Apply a `Selector` operator to a subset of configurations inside an `OptimInstance` and return the index within the archive (when `get_indivs` `FALSE`) or the configurations themselves (when `get_indivs` is `TRUE`).

It is not strictly necessary for the selector to select unique individuals / individuals without replacement.

Individuals are selected independently of whether they are "alive" or not. To select only from alive individuals, set `rows` to `inst$archive$data[, which(is.na(eol))]`.

Usage

```
mies_select_from_archive(
  inst,
  n_select,
  rows,
  selector = SelectorBest$new()$prime(inst$search_space),
  group_size = 1,
  get_indivs = TRUE
)
```

Arguments

inst
: (OptimInstance)
Optimization instance to evaluate.

n_select
: (integer(1))
Number of individuals to select.

rows
: optional (integer)
Indices of rows within inst to consider. If this is not given, then the entire archive is used.

selector
: (Selector)
Selector operator that selects individuals depending on configuration values and objective results. When selector$operate() is called, then objectives that are being minimized are multiplied with -1 (through mies_get_fitnesses()), since Selectors always try to maximize fitness. Defaults to SelectorBest.
The Selector must be primed on a superset of inst$search_space; this *includes* the "budget" component when performing multi-fidelity optimization. All components on which selector is primed on must occur in the archive.
The given Selector *may* return duplicates.

group_size
: (integer)
Sampling group size hint, indicating that the caller would prefer there to not be any duplicates within this group size. The Selector may or may not ignore this value, however. This may possibly happen because of certain configuration parameters, or because the input size is too small.
Must either be a scalar value or sum up to n_select. Must be non-negative. A scalar value of 0 is interpreted the same as 1.
Default is 1.

get_indivs
: (logical(1))
Whether to return configuration values from within the archive (TRUE) or just the indices within the archive (FALSE). Default is TRUE.

Value

integer | data.table: Selected individuals, either index into inst or subset of archive table, depending on get_indivs.

See Also

Other mies building blocks: mies_evaluate_offspring(), mies_generate_offspring(), mies_get_fitnesses(), mies_init_population(), mies_step_fidelity(), mies_survival_comma(), mies_survival_plus()

Examples

```
set.seed(1)
library("bbotk")
lgr::threshold("warn")

# Define the objective to optimize
objective <- ObjectiveRFun$new(
  fun = function(xs) {
```

```
    z <- exp(-xs$x^2 - xs$y^2) + 2 * exp(-(2 - xs$x)^2 - (2 - xs$y)^2)
    list(Obj = z)
  },
  domain = ps(x = p_dbl(-2, 4), y = p_dbl(-2, 4)),
  codomain = ps(Obj = p_dbl(tags = "maximize"))
)

# Get a new OptimInstance
oi <- OptimInstanceSingleCrit$new(objective,
  terminator = trm("evals", n_evals = 100)
)

s = sel("best")
s$prime(oi$search_space)

mies_init_population(inst = oi, mu = 6)

oi$archive

# Default: get individuals
mies_select_from_archive(oi, n_select = 2, rows = 1:6, selector = s)

# Alternatively: get rows within archive
mies_select_from_archive(oi, n_select = 2, rows = 1:6, selector = s,
  get_indivs = FALSE)

# Rows gotten from archive are relative from *all* rows, not from archive[rows]:
mies_select_from_archive(oi, n_select = 2, rows = 3:6, selector = s,
  get_indivs = FALSE)

##
# When using additional components: mies_select_from_archive learns about
# additional components from primed selector.

# Get a new OptimInstance
oi <- OptimInstanceSingleCrit$new(objective,
  terminator = trm("evals", n_evals = 100)
)

mies_init_population(inst = oi, mu = 6,
  additional_component_sampler = Sampler1DRfun$new(
    param = ParamDbl$new("additional", -1, 1), rfun = function(n) -1
  )
)

oi$archive

# Wrong: using selector primed only on search space. The resulting
# individuals do not have the additional component.
mies_select_from_archive(oi, n_select = 2, rows = 1:6, selector = s)

# Correct: selector must be primed on search space + additional component
mies_prime_operators(oi$search_space, selectors = list(s),
```

```
  additional_components = ps(additional = p_dbl(-1, 1)))

mies_select_from_archive(oi, n_select = 2, rows = 1:6, selector = s)
```

`mies_step_fidelity` *Re-Evaluate Existing Configurations with Higher Fidelity*

Description

As part of the "rolling-tide" multifidelity-setup, do reevaluation of configurations with higher fidelity that have survived lower-fidelity selection. The evaluations are done as part of the *current* generation, so the dob value is not increased.

This function should only be called when doing rolling-tide multifidelity, and should not be part of the MIES cycle otherwise.

Usage

```
mies_step_fidelity(
  inst,
  budget_id,
  fidelity,
  current_gen_only = FALSE,
  fidelity_monotonic = TRUE,
  additional_components = NULL
)
```

Arguments

`inst` (`OptimInstance`)
Optimization instance to evaluate.

`budget_id` (`character(1)`)
Budget component that is set to the `fidelity` value.

`fidelity` (`atomic(1)`)
Atomic scalar indicating the value to be assigned to the `budget_id` component of offspring.

`current_gen_only`
(`logical(1)`)
Whether to only re-evaluate survivors individuals generated in the latest generation (`TRUE`), or re-evaluate all currently alive individuals (`FALSE`). In any case, only individuals that were not already evaluated with the chosen fidelity are evaluated, so this will usually only have an effect when the fidelity of surviving individuals changed between generations.

`fidelity_monotonic`
(`logical(1)`)
Whether to only re-evaluate configurations for which the fidelity would increase. Default `TRUE`.

additional_components
: (ParamSet | NULL)
Additional components to optimize over, not included in search_space, but possibly used for self-adaption. This must be the ParamSet of mies_init_population()'s additional_component_sampler argument.

Value

invisible data.table: the performance values returned when evaluating the offspring values through eval_batch.

See Also

Other mies building blocks: mies_evaluate_offspring(), mies_generate_offspring(), mies_get_fitnesses(), mies_init_population(), mies_select_from_archive(), mies_survival_comma(), mies_survival_plus()

Examples

```
library("bbotk")
lgr::threshold("warn")

# Define the objective to optimize
objective <- ObjectiveRFun$new(
  fun = function(xs) {
    z <- exp(-xs$x^2 - xs$y^2) + 2 * exp(-(2 - xs$x)^2 - (2 - xs$y)^2)
    list(Obj = z)
  },
  domain = ps(x = p_dbl(-2, 4), y = p_dbl(-2, 4)),
  codomain = ps(Obj = p_dbl(tags = "maximize"))
)

# Get a new OptimInstance
oi <- OptimInstanceSingleCrit$new(objective,
  terminator = trm("evals", n_evals = 100)
)

budget_id = "y"

# Create an initial population with fidelity ("y") value 1
mies_init_population(oi, mu = 2, budget_id = budget_id, fidelity = 1)

oi$archive

# Re-evaluate these individuals with higher fidelity
mies_step_fidelity(oi, budget_id = budget_id, fidelity = 2)

oi$archive

# The following creates a new generation without killing the initial
# generation
offspring = data.frame(x = 0:1)
mies_evaluate_offspring(oi, offspring = offspring, budget_id = budget_id,
```

```
  fidelity = 3)

oi$archive

# Re-evaluate only individuals from last generation by setting current_gen_only
mies_step_fidelity(oi, budget_id = budget_id, fidelity = 4,
  current_gen_only = TRUE)

oi$archive

# Default: Re-evaluate all that *increase* fidelity: Only the initial
# population is re-evaluated here.
mies_step_fidelity(oi, budget_id = budget_id, fidelity = 3)

oi$archive

# To also re-evaluate individuals with *higher* fidelity, use
# 'fidelity_monotonic = FALSE'. This does not re-evaluate the points that already have
# the requested fidelity, however.
mies_step_fidelity(oi, budget_id = budget_id, fidelity = 3, fidelity_monotonic = FALSE)

oi$archive
```

`mies_survival_comma` *Choose Survivors According to the "Mu , Lambda" ("Comma") Strategy*

Description

Choose survivors during a MIES iteration using the "Comma" survival strategy, i.e. selecting survivors from the latest generation only, using a `Selector` operator, and choosing "elites" from survivors from previous generations using a different `Selector` operator.

When `n_elite` is greater than the number of alive individuals from previous generations, then all these individuals from previous generations survive. In this case, it is possible that more than `mu - n_elite` individuals from the current generation survive. Similarly, when `mu` is greater than the number of alive individuals from the last generation, then all these individuals survive.

Usage

```
mies_survival_comma(inst, mu, survival_selector, n_elite, elite_selector, ...)
```

Arguments

`inst` (`OptimInstance`)
Optimization instance to evaluate.

`mu` (`integer(1)`)
Population target size, non-negative integer.

survival_selector
: (Selector)
Selector operator that selects surviving individuals depending on configuration values and objective results. When survival_selector$operate() is called, then objectives that are being minimized are multiplied with -1 (through mies_get_fitnesses), since Selectors always try to maximize fitness.
The Selector must be primed on inst$search_space; this *includes* the "budget" component when performing multi-fidelity optimization.
The given Selector may *not* return duplicates.

n_elite
: (integer(1))
Number of individuals to carry over from previous generations. n_elite individuals will be selected by elite_selector, while mu - n_elite will be selected by survival_selector from the most recent generation. n_elite may be 0 (no elitism), in which case only individuals from the newest generation survive. n_elite must be strictly smaller than mu to permit any optimization progress.

elite_selector
: (Selector)
Selector operator that selects "elites", i.e. surviving individuals from previous generations, depending on configuration values and objective results. When elite_selector$operate() is called, then objectives that are being minimized are multiplied with -1 (through mies_get_fitnesses()), since Selectors always try to maximize fitness.
The Selector must be primed on inst$search_space; this *includes* the "budget" component when performing multi-fidelity optimization.
The given Selector may *not* return duplicates.

...
: (any)
Ignored, for compatibility with other mies_survival_* functions.

Value

invisible data.table: The value of inst$archive$data, changed in-place with eol set to the current generation for non-survivors.

See Also

Other mies building blocks: mies_evaluate_offspring(), mies_generate_offspring(), mies_get_fitnesses(), mies_init_population(), mies_select_from_archive(), mies_step_fidelity(), mies_survival_plus()

Examples

```
set.seed(1)
library("bbotk")
lgr::threshold("warn")

# Define the objective to optimize
objective <- ObjectiveRFun$new(
  fun = function(xs) {
    z <- exp(-xs$x^2 - xs$y^2) + 2 * exp(-(2 - xs$x)^2 - (2 - xs$y)^2)
    list(Obj = z)
```

```
  },
  domain = ps(x = p_dbl(-2, 4), y = p_dbl(-2, 4)),
  codomain = ps(Obj = p_dbl(tags = "maximize"))
)

# Get a new OptimInstance
oi <- OptimInstanceSingleCrit$new(objective,
  terminator = trm("evals", n_evals = 100)
)

mies_init_population(inst = oi, mu = 3)
# Usually the offspring is generated using mies_generate_offspring()
# Here shorter for demonstration purposes.
offspring = generate_design_random(oi$search_space, 3)$data
mies_evaluate_offspring(oi, offspring = offspring)

# State before: different generations of individuals. Alive individuals have
# 'eol' set to 'NA'.
oi$archive

s = sel("best")
s$prime(oi$search_space)
mies_survival_comma(oi, mu = 3, survival_selector = s,
  n_elite = 2, elite_selector = s)

# sel("best") lets only the best individuals survive.
# mies_survival_comma selects from new individuals (generation 2 in this case)
# and old individuals (all others) separately: n_elite = 2 from old,
# mu - n_elite = 1 from new.
# The surviving individuals have 'eol' set to 'NA'
oi$archive
```

`mies_survival_plus` *Choose Survivors According to the "Mu + Lambda" ("Plus") Strategy*

Description

Choose survivors during a MIES iteration using the "Plus" survival strategy, i.e. combining all alive individuals from the latest and from prior generations indiscriminately and choosing survivors using a survival `Selector` operator.

When `mu` is greater than the number of alive individuals, then all individuals survive.

Usage

```
mies_survival_plus(inst, mu, survival_selector, ...)
```

Arguments

`inst` (`OptimInstance`)
Optimization instance to evaluate.

mu (integer(1))
Population target size, non-negative integer.

survival_selector
(Selector)
Selector operator that selects surviving individuals depending on configuration values and objective results. When survival_selector$operate() is called, then objectives that are being minimized are multiplied with -1 (through mies_get_fitnesses), since Selectors always try to maximize fitness.
The Selector must be primed on inst$search_space; this *includes* the "budget" component when performing multi-fidelity optimization.
The given Selector may *not* return duplicates.

... (any)
Ignored, for compatibility with other mies_survival_* functions.

Value

invisible data.table: The value of inst$archive$data, changed in-place with eol set to the current generation for non-survivors.

See Also

Other mies building blocks: mies_evaluate_offspring(), mies_generate_offspring(), mies_get_fitnesses(), mies_init_population(), mies_select_from_archive(), mies_step_fidelity(), mies_survival_comma()

Examples

```
set.seed(1)
library("bbotk")
lgr::threshold("warn")

# Define the objective to optimize
objective <- ObjectiveRFun$new(
  fun = function(xs) {
    z <- exp(-xs$x^2 - xs$y^2) + 2 * exp(-(2 - xs$x)^2 - (2 - xs$y)^2)
    list(Obj = z)
  },
  domain = ps(x = p_dbl(-2, 4), y = p_dbl(-2, 4)),
  codomain = ps(Obj = p_dbl(tags = "maximize"))
)

# Get a new OptimInstance
oi <- OptimInstanceSingleCrit$new(objective,
  terminator = trm("evals", n_evals = 100)
)

mies_init_population(inst = oi, mu = 3)
offspring = generate_design_random(oi$search_space, 2)$data
mies_evaluate_offspring(oi, offspring = offspring)

# State before: different generations of individuals. Alive individuals have
# 'eol' set to 'NA'.
```

```
oi$archive

s = sel("best")
s$prime(oi$search_space)
mies_survival_plus(oi, mu = 3, survival_selector = s)

# sel("best") lets only the three best individuals survive.
# The others have 'eol = 2' (the current generation).
oi$archive
```

---

mlr_terminators_budget

Terminator that Limits Total Budget Component Evaluation

Description

Terminator that terminates after the sum (or similar aggregate) of a given "budget" search space component croses a threshold.

Dictionary

This Terminator can be created with the short access form trm() (trms() to get a list), or through the dictionary mlr_terminators in the following way:

```
# preferred
trm("budget")
trms("budget")  # takes vector IDs, returns list of Terminators

# long form
mlr_terminators$get("budget")
```

Configuration Parameters

- budget :: numeric(1)
  Total budget available, after which to stop. Not initialized and should be set to the desired value during construction.
- aggregate :: function
  Function taking a vector of values of the budget search space component, returning a scalar value to be compared to the budget configuration parameter. If this function returns a value greater or equal to budget the termination criterion is matched. Calling this function with NULL must return the lower bound of the budget value; percentage progress is reported as the progress from this lower bound to the value of budget. Initialized to sum().

Super class

bbotk::Terminator -> TerminatorBudget

Methods

Public methods:

- TerminatorBudget$new()
- TerminatorBudget$is_terminated()
- TerminatorBudget$clone()

Method new()**:** Initialize the TerminatorBudget object.

Usage:

```
TerminatorBudget$new()
```

Method is_terminated()**:** Is TRUE if when the termination criterion is matched, FALSE otherwise.

Usage:

```
TerminatorBudget$is_terminated(archive)
```

Arguments:

archive Archive Archive to check.

Returns: logical(1): Whether to terminate.

Method clone()**:** The objects of this class are cloneable with this method.

Usage:

```
TerminatorBudget$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
library("bbotk")
# Evaluate until sum of budget component of evaluated configs is >= 100
trm("budget", budget = 100)

# Evaluate until sum of two to the power of budget component is >= 100
trm("budget", budget = 1024, aggregate = function(x) sum(2 ^ x))
```

---

mlr_terminators_gens *Terminator that Counts OptimizerMies Generations*

Description

Terminator that terminates after a given number of generations have passed in OptimizerMies.

If OptimizerMies is started on an archive that already has evaluated configurations, these evaluations count as generation 0. If an initial, randomly sampled generation is generated by OptimizerMies, it has generation number 1. Setting generation to 1 therefore terminates after the evaluation of the initial sample, *unless* no initial sample is generated by OptimizerMies and instead found in the archive. generation set to 0 avoids any evaluation within OptimizerMies (but is ignored if no dob column is in the archive).

When doing multi-fidelity optimization, and fidelity of a configuration is increased because of a step in the fidelity schedule, or because they were sampled new and survived, then this fidelity refinement happens as part of an already started generation. This means termination at this fidelity refinement step is avoided.

Dictionary

This Terminator can be created with the short access form trm() (trms() to get a list), or through the dictionary mlr_terminators in the following way:

```
# preferred
trm("gens")
trms("gens")  # takes vector IDs, returns list of Terminators

# long form
mlr_terminators$get("gens")
```

Configuration Parameters

- generations :: integer(1)
  Number of generations to evaluate, after which to stop. Not initialized and should be set to the desired value during construction.

Super class

bbotk::Terminator -> TerminatorGenerations

Methods

Public methods:

- TerminatorGenerations$new()
- TerminatorGenerations$is_terminated()
- TerminatorGenerations$clone()

Method new()**:** Initialize the TerminatorGenerations object.

Usage:

```
TerminatorGenerations$new()
```

Method is_terminated()**:** Is TRUE if when the termination criterion is matched, FALSE otherwise.

Usage:

```
TerminatorGenerations$is_terminated(archive)
```

Arguments:

archive Archive Archive to check.

Returns: logical(1): Whether to terminate.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
TerminatorGenerations$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
library("bbotk")
trm("gens", generations = 10)
```

`mut` *Short Access Forms for Operators*

Description

These functions complement dict_mutators, dict_recombinators, dict_selectors with functions in the spirit of mlr3::mlr_sugar.

Usage

```
mut(.key, ...)

muts(.keys, ...)

rec(.key, ...)

recs(.key, ...)

sel(.key, ...)

sels(.key, ...)

scl(.key, ...)

scls(.key, ...)

ftr(.key, ...)

ftrs(.key, ...)
```

Arguments

`.key`	`(character(1))` Key passed to the respective dictionary to retrieve the object.

...	(named list()) Named arguments passed to the constructor, to be set as parameters in the paradox::ParamSet, or to be set as public field. See mlr3misc::dictionary_sugar_get() for more details.
.keys	(character()) Keys passed to the respective dictionary to retrieve multiple objects.

Value

- Mutator for mut()
- list of Mutator for muts()
- Recombinator for rec().
- list of Recombinator for recs().
- Selector for sel().
- list of Selector for sels().
- Scalor for scl().
- list of Scalor for scls().

See Also

Other dictionaries: dict_filtors, dict_mutators, dict_recombinators, dict_scalors, dict_selectors

Examples

```
mut("gauss", sdev = 0.5)
rec("xounif")
sel("random")
scl("nondom")
```

Mutator *Mutator Base Class*

Description

Base class representing mutation operations, inheriting from MiesOperator.

Mutations get a table of individuals as input and return a table of modified individuals as output. Individuals are acted on as individuals: every line of output corresponds to the same line of input, and presence or absence of other input lines does not affect the result.

Mutation operations are performed in ES algorithms to facilitate exploration of the search space around individuals.

Inheriting

Mutator is an abstract base class and should be inherited from. Inheriting classes should implement the private $.mutate() function. The user of the object calls $operate(), and the arguments are passed on to private $.mutate() after checking that the operator is primed, and that the values argument conforms to the primed domain. Typically, the $initialize() function should also be overloaded, and optionally the $prime() function; they should call their super equivalents.

In many cases, it is advisable to inherit from one of the abstract subclasses, such as MutatorNumeric, or MutatorDiscrete.

Super class

miesmuschel::MiesOperator -> Mutator

Methods

Public methods:

- Mutator$new()
- Mutator$clone()

Method new()**:** Initialize base class components of the Mutator.

Usage:

```
Mutator$new(
  param_classes = c("ParamLgl", "ParamInt", "ParamDbl", "ParamFct"),
  param_set = ps(),
  packages = character(0),
  dict_entry = NULL,
  own_param_set = quote(self$param_set)
)
```

Arguments:

param_classes (character)
: Classes of parameters that the operator can handle. May contain any of "ParamLgl", "ParamInt", "ParamDbl", "ParamFct". Default is all of them.
The $param_classes field will reflect this value.

param_set (ParamSet | list of expression)
: Strategy parameters of the operator. This should be created by the subclass and given to super$initialize(). If this is a ParamSet, it is used as the MiesOperator's ParamSet directly. Otherwise it must be a list of expressions e.g. created by alist() that evaluate to ParamSets, possibly referencing self and private. These ParamSet are then combined using a ParamSetCollection. Default is the empty ParamSet.
The $param_set field will reflect this value.

packages (character) Packages that need to be loaded for the operator to function. This should be declared so these packages can be loaded when operators run on parallel instances. Default is character(0).
The $packages field will reflect this values.

dict_entry (character(1) | NULL)
: Key of the class inside the Dictionary (usually one of dict_mutators, dict_recombinators,

dict_selectors), where it can be retrieved using a short access function. May be NULL if the operator is not entered in a dictionary.
The $dict_entry field will reflect this value.

own_param_set (language)
An expression that evaluates to a ParamSet indicating the configuration parameters that are entirely owned by this operator class (and not proxied from a construction argument object). This should be quote(self$param_set) (the default) when the param_set argument is not a list of expressions.

Method clone()**:** The objects of this class are cloneable with this method.

Usage:

```
Mutator$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other base classes: FiltorSurrogate, Filtor, MiesOperator, MutatorDiscrete, MutatorNumeric, OperatorCombination, RecombinatorPair, Recombinator, Scalor, SelectorScalar, Selector

Other mutators: MutatorDiscrete, MutatorNumeric, OperatorCombination, dict_mutators_cmpmaybe, dict_mutators_erase, dict_mutators_gauss, dict_mutators_maybe, dict_mutators_null, dict_mutators_proxy, dict_mutators_sequential, dict_mutators_unif

---

MutatorDiscrete *Discrete Mutator Base Class*

Description

Base class for mutation operations on discrete individuals, inheriting from Mutator.

MutatorDiscrete operators perform mutation on discrete (logical and factor valued) individuals. Inheriting operators implement the private $.mutate_discrete() function that is called once for each individual and is given a character vector.

Inheriting

MutatorDiscrete is an abstract base class and should be inherited from. Inheriting classes should implement the private $.mutate_discrete() function. During $operate(), the $.mutate_discrete() function is called once for each individual, with the parameters values (the individual as a single character vector), and levels (a list of character containing the possible values for each element of values). Typically, $initialize() should also be overloaded.

Super classes

miesmuschel::MiesOperator -> miesmuschel::Mutator -> MutatorDiscrete

Methods

Public methods:

- MutatorDiscrete$new()
- MutatorDiscrete$clone()

Method `new()`**:** Initialize base class components of the `MutatorNumeric`.

Usage:

```
MutatorDiscrete$new(
  param_classes = c("ParamLgl", "ParamFct"),
  param_set = ps(),
  packages = character(0),
  dict_entry = NULL,
  own_param_set = quote(self$param_set)
)
```

Arguments:

`param_classes` (`character`)
: Classes of parameters that the operator can handle. May contain any of `"ParamLgl"`, `"ParamFct"`. Default is both of them.
The `$param_classes` field will reflect this value.

`param_set` (`ParamSet` | `list` of `expression`)
: Strategy parameters of the operator. This should be created by the subclass and given to `super$initialize()`. If this is a `ParamSet`, it is used as the `MiesOperator`'s `ParamSet` directly. Otherwise it must be a `list` of expressions e.g. created by `alist()` that evaluate to `ParamSet`s, possibly referencing `self` and `private`. These `ParamSet` are then combined using a `ParamSetCollection`. Default is the empty `ParamSet`.
The `$param_set` field will reflect this value.

`packages` (`character`) Packages that need to be loaded for the operator to function. This should be declared so these packages can be loaded when operators run on parallel instances. Default is `character(0)`.
The `$packages` field will reflect this values.

`dict_entry` (`character(1)` | `NULL`)
: Key of the class inside the `Dictionary` (usually one of `dict_mutators`, `dict_recombinators`, `dict_selectors`), where it can be retrieved using a short access function. May be `NULL` if the operator is not entered in a dictionary.
The `$dict_entry` field will reflect this value.

`own_param_set` (`language`)
: An expression that evaluates to a `ParamSet` indicating the configuration parameters that are entirely owned by this operator class (and not proxied from a construction argument object). This should be `quote(self$param_set)` (the default) when the `param_set` argument is not a list of expressions.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MutatorDiscrete$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other base classes: FiltorSurrogate, Filtor, MiesOperator, MutatorNumeric, Mutator, OperatorCombination, RecombinatorPair, Recombinator, Scalor, SelectorScalar, Selector

Other mutators: MutatorNumeric, Mutator, OperatorCombination, dict_mutators_cmpmaybe, dict_mutators_erase, dict_mutators_gauss, dict_mutators_maybe, dict_mutators_null, dict_mutators_proxy, dict_mutators_sequential, dict_mutators_unif

---

`MutatorNumeric` *Numeric Mutator Base Class*

Description

Base class for mutation operations on numeric and integer valued individuals, inheriting from `Mutator`.

`MutatorNumeric` operators perform mutation on numeric (integer and real valued) individuals. Inheriting operators implement the private `$.mutate_numeric()` function that is called once for each individual and is given a numeric vector.

Inheriting

`MutatorNumeric` is an abstract base class and should be inherited from. Inheriting classes should implement the private `$.mutate_numeric()` function. During `$operate()`, the `$.mutate_numeric()` function is called once for each individual, with the parameters `values` (the individual as a single `numeric` vector), `lowers` and `uppers` (`numeric` vectors, the lower and upper bounds for each component of `values`). Typically, `$initialize()` should also be overloaded.

`MutatorNumeric`s that perform real-valued operations, such as e.g. `MutatorGauss`, operate on integers by widening the lower and upper bounds of integer components by 0.5, applying their operation, and rounding resulting values to the nearest integer (while always staying inside bounds).

Super classes

`miesmuschel::MiesOperator` -> `miesmuschel::Mutator` -> `MutatorNumeric`

Methods

Public methods:

- `MutatorNumeric$new()`
- `MutatorNumeric$clone()`

Method `new()`**:** Initialize base class components of the `MutatorNumeric`.

Usage:

```
MutatorNumeric$new(
  param_classes = c("ParamInt", "ParamDbl"),
  param_set = ps(),
  packages = character(0),
  dict_entry = NULL,
  own_param_set = quote(self$param_set)
)
```

Arguments:

`param_classes` (`character`)
: Classes of parameters that the operator can handle. May contain any of "ParamInt", "ParamDbl". Default is both of them.
  The $param_classes field will reflect this value.

`param_set` (ParamSet | list of expression)
: Strategy parameters of the operator. This should be created by the subclass and given to super$initialize(). If this is a ParamSet, it is used as the MiesOperator's ParamSet directly. Otherwise it must be a list of expressions e.g. created by alist() that evaluate to ParamSets, possibly referencing self and private. These ParamSet are then combined using a ParamSetCollection. Default is the empty ParamSet.
  The $param_set field will reflect this value.

`packages` (`character`) Packages that need to be loaded for the operator to function. This should be declared so these packages can be loaded when operators run on parallel instances. Default is character(0).
  The $packages field will reflect this values.

`dict_entry` (character(1) | NULL)
: Key of the class inside the Dictionary (usually one of dict_mutators, dict_recombinators, dict_selectors), where it can be retrieved using a short access function. May be NULL if the operator is not entered in a dictionary.
  The $dict_entry field will reflect this value.

`own_param_set` (`language`)
: An expression that evaluates to a ParamSet indicating the configuration parameters that are entirely owned by this operator class (and not proxied from a construction argument object). This should be quote(self$param_set) (the default) when the param_set argument is not a list of expressions.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MutatorNumeric$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other base classes: FiltorSurrogate, Filtor, MiesOperator, MutatorDiscrete, Mutator, OperatorCombination, RecombinatorPair, Recombinator, Scalor, SelectorScalar, Selector

Other mutators: MutatorDiscrete, Mutator, OperatorCombination, dict_mutators_cmpmaybe, dict_mutators_erase, dict_mutators_gauss, dict_mutators_maybe, dict_mutators_null, dict_mutators_proxy, dict_mutators_sequential, dict_mutators_unif

OperatorCombination *Self-Adaptive Operator Combinations*

Description

Combines multiple operators and makes operator-configuration parameters self-adaptive.

The OperatorCombination operators combine operators for different subspaces of individuals by wraping other MiesOperators given during construction. Different MiesOperators are assigned to different components or sets of components and operate on them independently of the rest of the components or the other operators. An operator can be assigned to a single component by giving it in operators with the name of the component, or to multiple components by giving it in operators with the name of a *group*. Groups are created by the groups argument, but several default groups that catch components by type exist.

Details

Operators can be made *self-adaptive* by coupling their configuration parameter values to values in individuals. This is done by giving functions in adaptions; these functions are executed for each individual before an operator is applied, and the result given to a named operator configuration parameter.

OperatorCombination is the base class from which MutatorCombination and RecombinatorCombination inherit. The latter two are to be used for Mutator and Recombinator objects, respectively.

Besides groups created with the groups construction argument, there are special groups that all unnamed operators fall into based on their Param class: "ParamInt", "ParamDbl", "ParamFct", and "ParamLgl". A component of an individual that is not named directly in operators or made part of a group in groups is automatically in one of these special groups. There is furthermore a special catch-all group "ParamAny", which catches all components that are are not operated directly, not in a group, and not in another special group that is itself named directly or in a group. I.e., all components that would otherwise have no assigned operation.

RecombinatorCombination can only combine operators where $n_indivs_in and $n_indivs_out can be combined. This is currently supported either when $n_indivs_in and $n_indivs_out for each operator are the same (but $n_indivs_in may be unequal $n_indivs_out in eacho of them); or when $n_indivs_in is equal to $n_indivs_out for each operator and the set of all $n_indivs_in that occur contains 1 and one more integer. $n_indivs_in and $n_indivs_out for the resulting RecombinatorCombination operator will be set the maximum of occuring $n_indivs_in and $n_indivs_out, respectively.

Supported Operand Types

Supported Param classes are calculated based on the supported classes of the wrapped operators. They are frequently just the set union of supported classes, unless inference can be drawn from type-specific groups that an operator is assigned to. If e.g. an operator that supports ParamDbl and ParamInt is assigned to group "ParamInt", and an operator that supports ParamLgl is assigned to component "a", then the result will support ParamLgl and ParamInt only.

Configuration Parameters

The OperatorCombination has the configuration parameters of all encapsulated MiesOperators, minus the configuration parameters that are named in the adaptions. Configuration parameter names are prefixed with the name of the MiesOperator in the operators list.

Dictionary

This Mutator can be created with the short access form mut() (muts() to get a list), or through the the dictionary dict_mutators in the following way:

```
# preferred:
mut("combine", <operators>, ...)
muts("combine", <operators>, ...)  # takes vector IDs, returns list of Mutators

# long form:
dict_mutators$get("combine", <operators>, ...)
```

This Recombinator can be created with the short access form rec() (recs() to get a list), or through the the dictionary dict_recombinators in the following way:

```
# preferred:
rec("combine", <operators>, ...)
recs("combine", <operators>, ...) # takes vector IDs, returns list of Recombinators

# long form:
dict_recombinators$get("combine", <operators>, ...)
```

Super class

miesmuschel::MiesOperator -> OperatorCombination

Active bindings

operators (named list of MiesOperator)
: List of operators to apply to components of individuals, as set during construction. Read-only.

groups (named list of character)
: List of groups that operators can act on, as set during construction. Read-only.

adaptions (named list of function)
: List of functions used for self-adaption of operators, as set during construction. Read-only.

binary_fct_as_logical (logical(1))
: Whether to treat binary ParamFct components of ParamSets as ParamLgl with respect to the special groups "ParamLgl" and "ParamFct", as set during construction. Read-only.

on_type_not_present (character(1))
: Action to perform during $prime() when an operator is assigned to a type special group but there is no component available that falls in this group. See the construction argument. Can be changed during the object's lifetime.

`on_name_not_present` (character(1))
: Action to perform during `$prime()` when an operator is assigned to a specifically named component, but the component is not present. See the construction argument. Can be changed during the object's lifetime.

Methods

Public methods:

- OperatorCombination$new()
- OperatorCombination$prime()
- OperatorCombination$clone()

Method `new()`**:** Initialize the `OperatorCombination` object.

Usage:

```
OperatorCombination$new(
  operators,
  groups = list(),
  adaptions = list(),
  binary_fct_as_logical = FALSE,
  on_type_not_present = "warn",
  on_name_not_present = "stop",
  granularity = 1,
  dict_entry = NULL,
  dict_shortaccess = NULL
)
```

Arguments:

`operators` (named `list` of `MiesOperator`)
: List of operators to apply to components of individuals. Names are either names of individual components, or group names which are either as defined through `groups` or special groups. Individual components can only be member of either a (non-special) group or named in `operators`, so a name that occurs in `operators` may not be a member of a group as defined in `groups`.
The `$operators` field will reflect this value.

`groups` (named `list` of `character`)
: List of groups that operators can act on. Names of this list define new groups. The content of each list element contains the names of components or special groups (a `Param` subclass name or `"ParamAny"`) to subsume under the group. Individual components can only be member of either a (non-special) group or named in `operators`, so a name that occurs in `operators` may not be a member of a group as defined in `groups`. The default is the empty list.
The `$groups` field will reflect this value.

`adaptions` (named `list` of `function`)
: List of functions used for self-adaption of operators. The names of the list must be names of configuration parameters of wrapped operators, prefixed with the corresponding name in the `operators` list. This is the same name as the configuration parameter would otherwise have if exposed by the `OperatorCombination` object. The values in the list must be functions that receive a single input, the individual or individuals being operated on, as a `data.table`.

It must return a value that is then assigned to the configuration parameter of the operator to which it pertains. Note that MutatorCombination adaption functions are always called with a data.table containing a single row, while RecombinatorCombination adaption functions are called with data.tables with multiple rows according to $n_indivs_in. In both cases, the return value must be a scalar. The default is the empty list.
The $adaption field will reflect this value.

binary_fct_as_logical (logical(1))
: Whether to treat binary ParamFct components of ParamSets as ParamLgl with respect to the special groups "ParamLgl" and "ParamFct". This does *not* perform any conversion, so a MiesOperator assigned to the "ParamLgl" special group when binary_fct_as_logical is TRUE and there are binary ParamFcts present will receive a factorial value and must also support ParamFct in this case. This is checked during $prime(), but not during construction. Default is FALSE.
The $binary_fct_as_logical field will reflect this value.

on_type_not_present (character(1))
: Action to perform during $prime() when an operator is assigned to a type special group but there is no component available that falls in this group, either because no components of the respective type are present, or because all these components are also directly named in operators or in groups. One of "quiet" (do nothing), "warn" (give warning, default), or "stop" (generate an error).
The writable $on_type_not_present field will reflect this value.

on_name_not_present (character(1))
: Action to perform during $prime() when an operator is assigned to a specifically named component, but the component is not present. One of "quiet" (do nothing), "warn" (give warning), or "stop" (generate an error, default).
The writable $on_name_not_present field will reflect this value.

granularity (integer(1))
: At what granularity to query adaptions for sets of individuals. Functions in adaptions are always called once per granularity individuals in input values, and the function argument in these calls will then have granularity number of rows. This is used internally, it is set to 1 for MutatorCombination, and to $n_indivs_in for RecombinatorCombination.

dict_entry (character(1) | NULL)
: Key of the class inside the Dictionary (usually one of dict_mutators, dict_recombinators, dict_selectors), where it can be retrieved using a short access function. May be NULL if the operator is not entered in a dictionary.
The $dict_entry field will reflect this value.

dict_shortaccess (character(1) | NULL)
: Name of the Dictionary short access function in which the operator is registered. This is used to inform the user about how to construct a given object. Should ordinarily be one of "mut", "rec", "sel".
The $dict_shortaccess field will reflect this value.

Method prime()**:** See MiesOperator method. Primes both this operator, as well as the wrapped operators given to operators during construction. Priming of wrapped operators happens according to component assignments to wrapped operators.

Usage:

```
OperatorCombination$prime(param_set)
```

Arguments:

param_set (ParamSet)
    Passed to MiesOperator$prime().

Returns: invisible self.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
OperatorCombination$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Super classes

miesmuschel::MiesOperator -> miesmuschel::OperatorCombination -> MutatorCombination

Methods

Public methods:

- MutatorCombination$new()
- MutatorCombination$clone()

Method new(): Initialize the MutatorCombination object.

Usage:

```
MutatorCombination$new(
  operators = list(),
  groups = list(),
  adaptions = list(),
  binary_fct_as_logical = FALSE,
  on_type_not_present = "warn",
  on_name_not_present = "stop"
)
```

Arguments:

operators see above.

groups see above.

adaptions see above.

binary_fct_as_logical see above.

on_type_not_present see above.

on_name_not_present see above.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
MutatorCombination$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Super classes

miesmuschel::MiesOperator -> miesmuschel::OperatorCombination -> RecombinatorCombination

Active bindings

n_indivs_in (integer(1))
: Number of individuals to consider at the same time. When operating, the number of input individuals must be divisible by this number.

n_indivs_out (integer(1))
: Number of individuals produced for each group of $n_indivs_in individuals.

Methods

Public methods:

- RecombinatorCombination$new()
- RecombinatorCombination$clone()

Method new()**:** Initialize the RecombinatorCombination object.

Usage:

```
RecombinatorCombination$new(
  operators = list(),
  groups = list(),
  adaptions = list(),
  binary_fct_as_logical = FALSE,
  on_type_not_present = "warn",
  on_name_not_present = "stop"
)
```

Arguments:

operators see above.

groups see above.

adaptions see above.

binary_fct_as_logical see above.

on_type_not_present see above.

on_name_not_present see above.

Method clone()**:** The objects of this class are cloneable with this method.

Usage:

```
RecombinatorCombination$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other base classes: FiltorSurrogate, Filtor, MiesOperator, MutatorDiscrete, MutatorNumeric, Mutator, RecombinatorPair, Recombinator, Scalor, SelectorScalar, Selector

Other mutators: MutatorDiscrete, MutatorNumeric, Mutator, dict_mutators_cmpmaybe, dict_mutators_erase, dict_mutators_gauss, dict_mutators_maybe, dict_mutators_null, dict_mutators_proxy, dict_mutators_sequential, dict_mutators_unif

Other mutator wrappers: dict_mutators_cmpmaybe, dict_mutators_maybe, dict_mutators_proxy, dict_mutators_sequential

Other recombinators: RecombinatorPair, Recombinator, dict_recombinators_cmpmaybe, dict_recombinators_conve dict_recombinators_cvxpair, dict_recombinators_maybe, dict_recombinators_null, dict_recombinators_proxy dict_recombinators_sbx, dict_recombinators_sequential, dict_recombinators_swap, dict_recombinators_xona dict_recombinators_xounif

Other recombinator wrappers: dict_recombinators_cmpmaybe, dict_recombinators_maybe, dict_recombinators_proxy, dict_recombinators_sequential

Examples

```
set.seed(1)
data = data.frame(x = 0, y = 0, a = TRUE, b = "a",
  stringsAsFactors = FALSE)  # necessary for R <= 3.6
p = ps(x = p_dbl(-1, 1), y = p_dbl(-1, 1), a = p_lgl(), b = p_fct(c("a", "b")))

# Demo operators:
m0 = mut("null")  # no mutation
msmall = mut("gauss", sdev = 0.1)  # mutates to small value around 0
mbig = mut("gauss", sdev = 100)  # likely mutates to +1 or -1
mflip = mut("unif", can_mutate_to_same = FALSE)  # flips TRUE/"a" to FALSE/"b"

# original:
data

# operators by name
op = mut("combine", operators = list(x = msmall, y = mbig, a = m0, b = mflip))
op$prime(p)
op$operate(data)

# operators by type
op = mut("combine",
  operators = list(ParamDbl = msmall, ParamLgl = m0, ParamFct = mflip)
)
op$prime(p)
op$operate(data)

# the binary ParamFct 'b' counts as 'ParamLgl' when
# 'binary_fct_as_logical' is set to 'TRUE'.
op = mut("combine",
  operators = list(ParamDbl = msmall, ParamLgl = m0),
  binary_fct_as_logical = TRUE
)
op$prime(p)
```

```
op$operate(data)

# operators by type; groups can be mixed types
op = mut("combine",
  operators = list(group1 = m0, group2 = msmall, group3 = mflip),
  groups = list(group1 = c("a", "x"), group2 = "y", group3 = "b")
)
op$prime(p)
op$operate(data)

# Special type-groups can be used inside groups.
op = mut("combine",
  operators = list(group1 = m0, b = mflip),
  groups = list(group1 = c("ParamDbl", "a"))
)
op$prime(p)
op$operate(data)

# Type-groups only capture all parameters that were not caught by name.
# The special 'ParamAny' group captures all that is left.
op = mut("combine",
  operators = list(ParamAny = m0, ParamDbl = msmall, x = mbig)
)
op$prime(p)
op$operate(data)

# Configuration parameters are named by names in the 'operators' list.
op$param_set

###
# Self-adaption:
# In this example, the 'ParamDbl''s operation is changed depending on the
# value of 'b'.
op = mut("combine",
  operators = list(ParamAny = m0, ParamLgl = mflip, ParamDbl = msmall),
  adaptions = list(ParamDbl.sdev = function(x) if (x$a) 100 else 0.1)
)
op$prime(p)

data2 = data[c(1, 1, 1, 1), ]
data2$a = c(TRUE, TRUE, FALSE, FALSE)

data2
# Note the value of x$a gets used line-wise, and that it is used *before*
# being flipped here. So the first two lines get large mutations, even though
# they have 'a' 'FALSE' after the operation.
op$operate(data2)
```

OptimizerMies *Mixed Integer Evolution Strategies Optimizer*

Description

Perform optimization using evolution strategies. `OptimizerMies` and `TunerMies` implement a standard ES optimization algorithm, performing initialization first, followed by a loop of performance evaluation, survival selection, parent selection, mutation, and recombination to generate new individuals to be evaluated. Currently, two different survival modes ("comma" and "plus") are supported. Multi-fidelity optimization, similar to the "rolling-tide" algorithm described in Fieldsend (2014), is supported. The modular design and reliance on `MiesOperator` objects to perform central parts of the optimization algorithm makes this `Optimizer` highly flexible and configurable. In combination with `OperatorCombination` mutators and recombinators, an algorithm as presented in Li (2013) can easily be implemented.

`OptimizerMies` implements a standard evolution strategies loop:

1. Prime operators, using `mies_prime_operators()`
2. Initialize and evaluate population, using `mies_init_population()`
3. Generate offspring by selecting parents, recombining and mutating them, using `mies_generate_offspring()`
4. Evaluate performance, using `mies_evaluate_offspring()`
5. Select survivors, using either `mies_survival_plus()` or `mies_survival_comma()`, depending on the `survival_strategy` configuration parameter
6. Optionally, evaluate survivors with higher fidelity if the multi-fidelity functionality is being used
7. Jump to 3.

Terminating

As with all optimizers, `Terminator`s are used to end optimization after a specific number of evaluations were performed, time elapsed, or other conditions are satisfied. Of particular interest is `TerminatorGenerations`, which terminates after a number of generations were evaluated in `OptimizerMies`. The initial population counts as generation 1, its offspring as generation 2 etc.; fidelity refinements (step 6. in the algorithm description above) are always included in their generation, `TerminatorGenerations` avoids terminating right before they are evaluated. Other terminators may, however, end the optimization process at any time.

Multi-Fidelity

`miesmuschel` provides a simple multi-fidelity optimization mechanism that allows both the refinement of fidelity as the optimization progresses, as well as fidelity refinement within each generation. When `multi_fidelity` is `TRUE`, then one search space component of the `OptimInstance` must have the `"budget"` tag, which is then optimized as the "budget" component. This means that the value of this component is determined by the `fidelity`/`fidelity_offspring` parameters, which are functions that get called whenever individuals get evaluated. The `fidelity` function is evaluated before step 2 and before every occurrence of step 6 in the algorithm, it returns the value of the budget search space component that all individuals that survive the current generation should be evaluated with. `fidelity_offspring` is called before step 4 and determines the fidelity that newly sampled offspring individuals should be evaluated with; it may be desirable to set this to a lower value than `fidelity` to save budget when preliminarily evaluating newly sampled individuals that may or may not perform well compared to already sampled individuals. Individuals that survive the generation and are not removed in step 5 will be re-evaluated with the fidelity-value in step 6 before the next loop iteration.

fidelity and fidelity_offspring must have arguments inst, budget_id, last_fidelity and last_fidelity_offspring. inst is the OptimInstance bein optimized, the functions can use it to determine the progress of the optimization, e.g. query the current generation with mies_generation. budget_id identifies the search space component being used as budget parameter. last_fidelity and last_fidelity_offspring contain the last values given by fidelity / fidelity_offspring. Should the offspring-fidelity (as returned by fidelity_offspring always be the same as the parent generation fidelity (as returned by fidelity), for example, then fidelity_offspring can be set to a function that just returns last_fidelity; this is actually the behaviour that fidelity_offspring is initialized with.

OptimizerMies avoids re-evaluating individuals if the fidelity parameter does not change. This means that setting fidelity and fidelity_offspring to the same value avoids re-evaluating individuals in step 6. When fidelity_monotonic is TRUE, re-evaluation is also avoided should the desired fidelity parameter value decrease. When fidelity_current_gen_only is TRUE, then step 6 only re-evaluates individuals that were created in the current generation (in the previous step 4) and sets the fidelity for individuals that are created in step 6, but it does not re-evaluate individuals that survived from earlier generations or were already in the OptimInstance when optimization started; it is recommended to leave this value at TRUE which it is initialized with.

Additional Components

The search space over which the optimization is performed is fundamentally tied to the Objective, and therefore to the OptimInstance given to OptimizerMies$optimize(). However, some advanced Evolution Strategy based algorithms may need to make use of additional search space components that are independent of the particular objective. An example is self-adaption as implemented in OperatorCombination, where one or several components can be used to adjust operator behaviour. These additional components are supplied to the optimizer through the additional_component_sampler configuration parameter, which takes a Sampler object. This object both has an associated ParamSet which represents the additional components that are present, and it provides a method for generating the initial values of these components. The search space that is seen by the MiesOperators is then the union of the OptimInstance's ParamSet, and the Sampler's ParamSet.

Configuration Parameters

OptimizerMies has the configuration parameters of the mutator, recombinator, parent_selector, survival_selector, init_selector, and, if given, elite_selector operator given during construction, and prefixed according to the name of the argument (mutator's configuration parameters are prefixed "mutator." etc.). When using the construction arguments' default values, they are all "proxy" operators: MutatorProxy, RecombinatorProxy and SelectorProxy. This means that the respective configuration parameters become mutator.operation, recombinator.operation etc., so the operators themselves can be set via configuration parameters in this case.

Further configuration parameters are:

- lambda :: integer(1)
  Offspring size: Number of individuals that are created and evaluated anew for each generation. This is equivalent to the lambda parameter of mies_generate_offspring(), see there for more information. Must be set by the user.

- mu :: integer(1)
  Population size: Number of individuals that are sampled in the beginning, and which are selected with each survival step. This is equivalent to the mu parameter of mies_init_population(), see there for more information. Must be set by the user.
- survival_strategy :: character(1)
  May be "plus", or, if the elite_selector construction argument is not NULL, "comma": Choose whether mies_survival_plus() or mies_survival_comma() is used for survival selection. Initialized to "plus".
- n_elite :: integer(1)
  Only if the elite_selector construction argument is not NULL, and only valid when survival_strategy is "comma": Number of elites, i.e. individuals from the parent generation, to keep during "Comma" survival. This is equivalent to the n_elite parameter of mies_survival_comma(), see there for more information.
- initializer :: function
  Function that generates the initial population as a Design object, with arguments param_set and n, functioning like paradox::generate_design_random or paradox::generate_design_lhs. This is equivalent to the initializer parameter of mies_init_population(), see there for more information. Initialized to generate_design_random().
- additional_component_sampler :: Sampler | NULL
  Additional components that may be part of individuals as seen by mutation, recombination, and selection MiesOperators, but that are not part of the search space of the OptimInstance being optimized. This is equivalent to the additional_component_sampler parameter of mies_init_population(), see there for more information. Initialized to NULL (no additional components).
- fidelity :: function
  Only if the multi_fidelity construction argument is TRUE: Function that determines the value of the "budget" component of surviving individuals being evaluated when doing multi-fidelity optimization. It must have arguments named inst, budget_id, last_fidelity and last_fidelity_offspring, see the "Multi-Fidelity"-section for more details. Its return value is given to mies_init_population() and mies_step_fidelity(). When this configuration parameter is present (i.e. multi_fidelity is TRUE), then it is initialized to a function returning the value 1.
- fidelity_offspring :: function
  Only if the multi_fidelity construction argument is TRUE: Function that determines the value of the "budget" component of newly sampled offspring individuals being evaluated when doing multi-fidelity optimization. It must have arguments named inst, budget_id, last_fidelity and last_fidelity_offspring, see the "Multi-Fidelity"-section for more details. Its return value is given to mies_evaluate_offspring(). When this configuration parameter is present (i.e. multi_fidelity is TRUE), then it is initialized to a function returning the value of last_fidelity, i.e. the value returned by the last call to the fidelity configuration parameter. This is the recommended value when fidelity should not change within a generation, since this means that survivor selection is performed with individuals that were evaluated with the same fidelity (at least if fidelity_current_gen_only is also set to FALSE) .
- fidelity_current_gen_only :: logical(1)
  Only if the multi_fidelity construction argument is TRUE: When doing fidelity refinement in mies_step_fidelity(), whether to refine all individuals with different budget component, or

only individuals created in the current generation. This is equivalent to the `current_gen_only` parameter of `mies_step_fidelity()`, see there for more information.
When this configuration parameter is present (i.e. `multi_fidelity` is `TRUE`), then it is initialized to `FALSE`, the recommended value.

- `fidelity_monotonic` :: `logical(1)`
  Only if the `multi_fidelity` construction argument is `TRUE`: Whether to only do fidelity refinement in `mies_step_fidelity()` for individuals for which the when budget component value would *increase*. This is equivalent to the `monotonic` parameter of `mies_step_fidelity()`, see there for more information.
  When this configuration parameter is present (i.e. `multi_fidelity` is `TRUE`), then it is initialized to `TRUE`. When optimization is performed on problems that have a categorical "budget" parameter, then this value should be set to `FALSE`.

Super class

`bbotk::Optimizer` -> `OptimizerMies`

Active bindings

`mutator` (`Mutator`)
: Mutation operation to perform during `mies_generate_offspring()`.

`recombinator` (`Recombinator`)
: Recombination operation to perform during `mies_generate_offspring()`.

`parent_selector` (`Selector`)
: Parent selection operation to perform during `mies_generate_offspring()`.

`survival_selector` (`Selector`)
: Survival selection operation to use in `mies_survival_plus()` or `mies_survival_comma()`.

`elite_selector` (`Selector` | NULL)
: Elite selector used in `mies_survival_comma()`.

`init_selector` (`Selector`)
: Selection operation to use when there are more than `mu` individuals present at the beginning of the optimization.

`param_set` (`ParamSet`)
: Configuration parameters of the optimization algorithm.

Methods

Public methods:

- `OptimizerMies$new()`
- `OptimizerMies$clone()`

Method `new()`**:** Initialize the `OptimizerMies` object.

Usage:

```
OptimizerMies$new(
  mutator = MutatorProxy$new(),
  recombinator = RecombinatorProxy$new(),
```

```
  parent_selector = SelectorProxy$new(),
  survival_selector = SelectorProxy$new(),
  elite_selector = NULL,
  init_selector = survival_selector,
  multi_fidelity = FALSE
)
```

Arguments:

`mutator` (Mutator)
: Mutation operation to perform during `mies_generate_offspring()`, see there for more information. Default is `MutatorProxy`, which exposes the operation as a configuration parameter of the optimizer itself.
  The `$mutator` field will reflect this value.

`recombinator` (Recombinator)
: Recombination operation to perform during `mies_generate_offspring()`, see there for more information. Default is `RecombinatorProxy`, which exposes the operation as a configuration parameter of the optimizer itself. Note: The default `RecombinatorProxy` has `$n_indivs_in` set to 2, so to use recombination operations with more than two inputs, or to use population size of 1, it may be necessary to construct this argument explicitly.
  The `$recombinator` field will reflect this value.

`parent_selector` (Selector)
: Parent selection operation to perform during `mies_generate_offspring()`, see there for more information. Default is `SelectorProxy`, which exposes the operation as a configuration parameter of the optimizer itself.
  The `$parent_selector` field will reflect this value.

`survival_selector` (Selector)
: Survival selection operation to use in `mies_survival_plus()` or `mies_survival_comma()` (depending on the `survival_strategy` configuration parameter), see there for more information. Default is `SelectorProxy`, which exposes the operation as a configuration parameter of the optimizer itself.
  The `$survival_selector` field will reflect this value.

`elite_selector` (Selector | NULL)
: Elite selector used in `mies_survival_comma()`, see there for more information. "Comma" selection is only available when this argument is not `NULL`. Default `NULL`.
  The `$elite_selector` field will reflect this value.

`init_selector` (Selector)
: Survival selection operation to give to the `survival_selector` argument of `mies_init_population()`; it is used if the `OptimInstance` being optimized already contains more (alive) individuals than `mu`. Default is the value given to `survival_selector`. The `$init_selector` field will reflect this value.

`multi_fidelity` (logical(1))
: Whether to enable multi-fidelity optimization. When this is `TRUE`, then the `OptimInstance` being optimized must contain a `Param` tagged `"budget"`, which is then used as the "budget" search space component, determined by `fidelity` and `fidelity_offspring` instead of by the `MiesOperator`s themselves. For multi-fidelity optimization, the `fidelity`, `fidelity_offspring`, `fidelity_current_gen_only`, and `fidelity_monotonic` configuration parameters must be given to determine multi-fidelity behaviour. (While the initial values for most of these are probably good for most cases in which more budget implies higher fidelity, at least the `fidelity` configuration parameter should be adjusted in most cases). Default is `FALSE`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
OptimizerMies$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Super classes

`mlr3tuning::Tuner` -> `mlr3tuning::TunerFromOptimizer` -> `TunerMies`

Methods

Public methods:

- `TunerMies$new()`
- `TunerMies$clone()`

Method `new()`**:** Initialize the `TunerMies` object.

Usage:

```
TunerMies$new(
  mutator = MutatorProxy$new(),
  recombinator = RecombinatorProxy$new(),
  parent_selector = SelectorProxy$new(),
  survival_selector = SelectorProxy$new(),
  elite_selector = NULL,
  init_selector = survival_selector,
  multi_fidelity = FALSE
)
```

Arguments:

`mutator` (`Mutator`)

`recombinator` (`Recombinator`)

`parent_selector` (`Selector`)

`survival_selector` (`Selector`)

`elite_selector` (`Selector` | `NULL`)

`init_selector` (`Selector`)

`multi_fidelity` (`logical(1)`)

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
TunerMies$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Fieldsend, E J, Everson, M R (2014). “The rolling tide evolutionary algorithm: A multiobjective optimizer for noisy optimization problems.” *IEEE Transactions on Evolutionary Computation*, **19**(1), 103–117.

Li, Rui, Emmerich, TM M, Eggermont, Jeroen, B"ack, Thomas, Sch"utz, Martin, Dijkstra, Jouke, Reiber, HC J (2013). “Mixed integer evolution strategies for parameter optimization.” *Evolutionary computation*, **21**(1), 29–64.

Examples

```
lgr::threshold("warn")

op.m <- mut("gauss", sdev = 0.1)
op.r <- rec("xounif", p = .3)
op.parent <- sel("random")
op.survival <- sel("best")

#####
# Optimizing a Function
#####

library("bbotk")

# Define the objective to optimize
objective <- ObjectiveRFun$new(
  fun = function(xs) {
    z <- exp(-xs$x^2 - xs$y^2) + 2 * exp(-(2 - xs$x)^2 - (2 - xs$y)^2)
    list(Obj = z)
  },
  domain = ps(x = p_dbl(-2, 4), y = p_dbl(-2, 4)),
  codomain = ps(Obj = p_dbl(tags = "maximize"))
)

# Get a new OptimInstance
oi <- OptimInstanceSingleCrit$new(objective,
  terminator = trm("evals", n_evals = 100)
)

# Create OptimizerMies object
mies_opt <- opt("mies", mutator = op.m, recombinator = op.r,
  parent_selector = op.parent, survival_selector = op.survival,
  mu = 10, lambda = 5)

# mies_opt$optimize performs MIES optimization and returns the optimum
mies_opt$optimize(oi)

#####
# Optimizing a Machine Learning Method
#####
```

```
# Note that this is a short example, aiming at clarity and short runtime.
# The settings are not optimal for hyperparameter tuning. The resampling
# in particular should not be "holdout" for small datasets where this gives
# a very noisy estimate of performance.

library("mlr3")
library("mlr3tuning")

# The Learner to optimize
learner = lrn("classif.rpart")

# The hyperparameters to optimize
learner$param_set$values[c("cp", "maxdepth")] = list(to_tune())

# Get a TuningInstance
ti = TuningInstanceSingleCrit$new(
  task = tsk("iris"),
  learner = learner,
  resampling = rsmp("holdout"),
  measure = msr("classif.acc"),
  terminator = trm("gens", generations = 10)
)

# Create TunerMies object
mies_tune <- tnr("mies", mutator = op.m, recombinator = op.r,
  parent_selector = op.parent, survival_selector = op.survival,
  mu = 10, lambda = 5)

# mies_tune$optimize performs MIES optimization and returns the optimum
mies_tune$optimize(ti)
```

ParamSetShadow *ParamSetShadow*

Description

Wraps another ParamSet and shadows out a subset of its Params. The original ParamSet can still be accessed through the $origin field; otherwise, the ParamSetShadow behaves like a ParamSet where the shadowed Params are not present.

Super class

paradox::ParamSet -> ParamSetShadow

Active bindings

params (named list of Param) List of Param that are members of the wrapped ParamSet with the shadowed Params removed.

params_unid (named list of Param) List of Param that are members of the wrapped ParamSet
: with the shadowed Params removed. This is a field mostly for internal usage that has the $ids set to invalid values but avoids cloning overhead.

deps (data.table)
: Table of dependencies, as in ParamSet. The dependencies that are related to shadowed parameters are not exposed. This data.table should be seen as read-only and not modified in-place; instead, the $origin's $deps should be modified.

values (named list)
: List of values, as in ParamSet, with the shadowed values removed.

set_id (data.table)
: Id of the wrapped ParamSet. Changing this value will also change the wrapped ParamSet's $set_id accordingly.

origin (ParamSet)
: ParamSet being wrapped. This object can be modified by reference to influence the ParamSetShadow object itself.

Methods

Public methods:

- ParamSetShadow$new()
- ParamSetShadow$add()
- ParamSetShadow$subset()
- ParamSetShadow$add_dep()
- ParamSetShadow$clone()

Method new()**:** Initialize the ParamSetShadow object.

Usage:

```
ParamSetShadow$new(set, shadowed)
```

Arguments:

set (ParamSet)
: ParamSet to wrap.

shadowed (character)
: Ids of Params to shadow from sets, must be a subset of set$ids().

Method add()**:** Adds a single param or another set to this set, all params are cloned.

This calls the underlying ParamSet's $add() function.

Param with ids that also occur in the underlying ParamSet but are shadowed can *not* be added and instead will result in an error.

Usage:

```
ParamSetShadow$add(p)
```

Arguments:

p (Param | ParamSet)

Returns: invisible(self).

Method `subset()`**:** Reduces the parameters to the ones of passed ids.

This calls the underlying `ParamSet`'s `$subset()` function.

Usage:

```
ParamSetShadow$subset(ids)
```

Arguments:

`ids` (`character`)

Returns: `invisible(self)`.

Method `add_dep()`**:** Adds a dependency to the unterlying `ParamSet`.

Usage:

```
ParamSetShadow$add_dep(id, on, cond)
```

Arguments:

`id` (`character(1)`)

`on` (`character(1)`)

`cond` (`Condition`)

Returns: `invisible(self)`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ParamSetShadow$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
p1 = ps(x = p_dbl(0, 1), y = p_lgl())
p1$values = list(x = 0.5, y = TRUE)
print(p1)

p2 = ParamSetShadow$new(p1, "x")
print(p2$values)

p2$values$y = FALSE
print(p2)

print(p2$origin$values)
```

`rank_nondominated` *Perform Nondominated Sorting*

Description

Assign elements of `fitnesses` to nondominated fronts.

The first nondominated front is the set of individuals that is not dominated by any other individual with respect to any fitness dimension, i.e. where no other individual exists that has all fitness values greater or equal, with at least one fitness value strictly greater.

The n'th nondominated front is the set of individuals that is not dominated by any other individual that is not in any nondominated front with smaller n.

Fitnesses are *maximized*, so the individuals in lower numbered nondominated fronts tend to have higher fitness values.

Usage

```
rank_nondominated(fitnesses, epsilon = 0)
```

Arguments

`fitnesses`	(`numeric matrix`) fitness matrix, with one row per individual and one column per objective
`epsilon`	(`numeric`) Epsilon-vaue for non-dominance. A value is epsilon-dominated by another if it is at least `epsilon` smaller than the other in all dimensions, and more than `epsilon` smaller than the other in one dimension. `epsilon` may be a scalar, in which case it is used for all dimensions or a vector, in which case its length must match the number of dimensions. Default 0.

Value

`list`: `$front`: Vector assigning each individual in `fitnesses` its nondominated front. `$domcount`: Length N vector counting the number of individuals that dominate the given individual.

`Recombinator` *Recombinator Base Class*

Description

Base class representing recombination operations, inheriting from `MiesOperator`.

Recombinators get a table of individuals as input and return a table of modified individuals as output. Individuals are acted on by groups: every `$n_indivs_out` lines of output corresponds to a group of `$n_indivs_in` lines of input, and presence or absence of other input groups does not affect the result.

Recombination operations are performed in ES algorithms to facilitate exploration of the search space that combine partial solutions.

Inheriting

Recombinator is an abstract base class and should be inherited from. Inheriting classes should implement the private $.recombine() function. The user of the object calls $operate(), which calls $.recombine() for each $n_indivs_in sized group of individuals after checking that the operator is primed, that the values argument conforms to the primed domain. $.recombine() should then return a table of $n_indivs_out individuals for each call. Typically, the $initialize() function should also be overloaded, and optionally the $prime() function; they should call their super equivalents.

Super class

miesmuschel::MiesOperator -> Recombinator

Active bindings

n_indivs_in (integer(1))
: Number of individuals to consider at the same time. When operating, the number of input individuals must be divisible by this number.

n_indivs_out (integer(1))
: Number of individuals produced for each group of $n_indivs_in individuals.

Methods

Public methods:

- Recombinator$new()
- Recombinator$clone()

Method new()**:** Initialize base class components of the Recombinator.

Usage:

```
Recombinator$new(
  param_classes = c("ParamLgl", "ParamInt", "ParamDbl", "ParamFct"),
  param_set = ps(),
  n_indivs_in = 2,
  n_indivs_out = n_indivs_in,
  packages = character(0),
  dict_entry = NULL,
  own_param_set = quote(self$param_set)
)
```

Arguments:

param_classes (character)
: Classes of parameters that the operator can handle. May contain any of "ParamLgl", "ParamInt", "ParamDbl", "ParamFct". Default is all of them.
The $param_classes field will reflect this value.

param_set (ParamSet | list of expression)
: Strategy parameters of the operator. This should be created by the subclass and given to super$initialize(). If this is a ParamSet, it is used as the MiesOperator's ParamSet directly. Otherwise it must be a list of expressions e.g. created by alist() that evaluate

to ParamSets, possibly referencing self and private. These ParamSet are then combined using a ParamSetCollection. Default is the empty ParamSet.
The $param_set field will reflect this value.

n_indivs_in (integer(1))
Number of individuals to consider at the same time. When operating, the number of input individuals must be divisible by this number. Default 2.
The $n_indivs_in field will reflect this value.

n_indivs_out (integer(1))
Number of individuals that result for each n_indivs_in lines of input. The number of results from the recombinator will be nrow(values) / n_indivs_in * n_indivs_out. Default equal to n_indivs_in.
The $n_indivs_out field will reflect this value.

packages (character) Packages that need to be loaded for the operator to function. This should be declared so these packages can be loaded when operators run on parallel instances. Default is character(0).
The $packages field will reflect this values.

dict_entry (character(1) | NULL)
Key of the class inside the Dictionary (usually one of dict_mutators, dict_recombinators, dict_selectors), where it can be retrieved using a short access function. May be NULL if the operator is not entered in a dictionary.
The $dict_entry field will reflect this value.

own_param_set (language)
An expression that evaluates to a ParamSet indicating the configuration parameters that are entirely owned by this operator class (and not proxied from a construction argument object). This should be quote(self$param_set) (the default) when the param_set argument is not a list of expressions.

Method clone()**:** The objects of this class are cloneable with this method.

Usage:

```
Recombinator$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other base classes: FiltorSurrogate, Filtor, MiesOperator, MutatorDiscrete, MutatorNumeric, Mutator, OperatorCombination, RecombinatorPair, Scalor, SelectorScalar, Selector

Other recombinators: OperatorCombination, RecombinatorPair, dict_recombinators_cmpmaybe, dict_recombinators_convex, dict_recombinators_cvxpair, dict_recombinators_maybe, dict_recombinators_nul dict_recombinators_proxy, dict_recombinators_sbx, dict_recombinators_sequential, dict_recombinators_swa dict_recombinators_xonary, dict_recombinators_xounif

RecombinatorPair *Pair Recombinator Base Class*

Description

Base class for recombination that covers the common case of combining two individuals, where two (typically complementary) child individuals could be taken as the result, such as bitwise crossover or SBX crossover.

This is a relatively lightweight class, it adds the `keep_complement` active binding and sets `$n_indivs_in` and `$n_indivs_out` appropriately.

Inheriting

`RecombinatorPair` is an abstract base class and should be inherited from. Inheriting classes should implement the private `$.recombine_pair()` function. During `$operate()`, the `$.recombine_pair()` function is called with the same input as the `$.recombine()` function of the `Recombinator` class. It should return a `data.table` of two individuals.

Constructors of inheriting classes should have a `keep_complement` argument.

Super classes

`miesmuschel::MiesOperator` -> `miesmuschel::Recombinator` -> `RecombinatorPair`

Active bindings

`keep_complement (logical(1))`
: Whether the operation keeps both resulting individuals of the operation or discards the complement.

Methods

Public methods:

- `RecombinatorPair$new()`
- `RecombinatorPair$clone()`

Method `new()`**:** Initialize base class components of the `RecombinatorPair`.

Usage:

```
RecombinatorPair$new(
  keep_complement = TRUE,
  param_classes = c("ParamLgl", "ParamInt", "ParamDbl", "ParamFct"),
  param_set = ps(),
  packages = character(0),
  dict_entry = NULL,
  own_param_set = quote(self$param_set)
)
```

Arguments:

keep_complement (logical(1))
Whether the operation should keep both resulting individuals (TRUE), or only the first and discard the complement (FALSE). Default TRUE. The $keep_complement field will reflect this value.

param_classes (character)
Classes of parameters that the operator can handle. May contain any of "ParamLgl", "ParamInt", "ParamDbl", "ParamFct". Default is all of them.
The $param_classes field will reflect this value.

param_set (ParamSet | list of expression)
Strategy parameters of the operator. This should be created by the subclass and given to super$initialize(). If this is a ParamSet, it is used as the MiesOperator's ParamSet directly. Otherwise it must be a list of expressions e.g. created by alist() that evaluate to ParamSets, possibly referencing self and private. These ParamSet are then combined using a ParamSetCollection. Default is the empty ParamSet.
The $param_set field will reflect this value.

packages (character) Packages that need to be loaded for the operator to function. This should be declared so these packages can be loaded when operators run on parallel instances. Default is character(0).
The $packages field will reflect this values.

dict_entry (character(1) | NULL)
Key of the class inside the Dictionary (usually one of dict_mutators, dict_recombinators, dict_selectors), where it can be retrieved using a short access function. May be NULL if the operator is not entered in a dictionary.
The $dict_entry field will reflect this value.

own_param_set (language)
An expression that evaluates to a ParamSet indicating the configuration parameters that are entirely owned by this operator class (and not proxied from a construction argument object). This should be quote(self$param_set) (the default) when the param_set argument is not a list of expressions.

Method clone()**:** The objects of this class are cloneable with this method.

Usage:

```
RecombinatorPair$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other base classes: FiltorSurrogate, Filtor, MiesOperator, MutatorDiscrete, MutatorNumeric, Mutator, OperatorCombination, Recombinator, Scalor, SelectorScalar, Selector

Other recombinators: OperatorCombination, Recombinator, dict_recombinators_cmpmaybe, dict_recombinators_convex, dict_recombinators_cvxpair, dict_recombinators_maybe, dict_recombinators_nul dict_recombinators_proxy, dict_recombinators_sbx, dict_recombinators_sequential, dict_recombinators_swa dict_recombinators_xonary, dict_recombinators_xounif

repr — *Create a 'call' Object Representation*

Description

repr() creates a call object representing obj, if possible. Evaluating the call should come close to recreating the original object.

In the most trivial cases, it should be possible to recreate objects from their representation by evaluating them using eval(). Important exceptions are:

- Functions are represented by their source code, if available, and by their AST if not. This drops the context from their environments and recreated objects will not work if they contain functions that depend on specific environments
- environments are not represented.
- R6 objects are only represented if they have a $repr() function. This function may have arbitrary arguments, and should have a ... argument to capture ignored arguments.

Objects that can not be represented are currently mapped to the call `stop("<###>")`, where ### is a short description of the non-representable object.

Usage

```
repr(obj, ...)
```

Arguments

obj	(any) Object to create a representation of.
...	(any) Further arguments to be passed to class methods. Currently in use are: • skip_defaults (logical(1)) whether to skip construction arguments that have their default value. Default TRUE. • show_params (logical(1)) whether to show ParamSet values. Default TRUE. • show_constructor_args (logical(1)) whether to show construction args that are not ParamSet values. Default TRUE.

Value

call: A call that, when evaluated, tries to re-create the object.

SamplerRandomWeights *Sampler for Projection Weights*

Description

Sampler for a single `ParamUty` that samples weight-matrices as used by `ScalorFixedProjection`.

Super class

`paradox::Sampler` -> `SamplerRandomWeights`

Active bindings

`nobjectives` (`numeric(1)`)
: Number of objectives for which weights are generated.

`nweights` (`numeric(1)`)
: Number of weight vectors generated for each configuration.

`weights_component_id` (`numeric(1)`)
: search space component identifying the weights by which to scalarize.

Methods

Public methods:

- `SamplerRandomWeights$new()`
- `SamplerRandomWeights$clone()`

Method `new()`**:** Initialize the `SamplerRandomWeights` object.

Usage:

```
SamplerRandomWeights$new(
  nobjectives = 2,
  nweights = 1,
  weights_component_id = "scalarization_weights"
)
```

Arguments:

`nobjectives` (`numeric(1)`)
: Number of objectives for which weights are generated.

`nweights` (`numeric(1)`)
: Number of weight vectors generated for each configuration.

`weights_component_id` (`character(1)`)
: Id of the `ParamUty`. Default is `"scalarization_weights"`. Can be changed arbitrarily but should match the `ScalorFixedProjection`'s `weights_component_id`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SamplerRandomWeights$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
set.seed(1)
```

Scalarizer *Scalarizer*

Description

Scalarizer objects are functions taking a fitness-matrix fitnesses (Nindivs x Nobjectives, with higher values indicating higher desirability) and a list of weight matrices weights (Nindivs elements of Nobjectives x Nweights matrices; positive weights indicate a positive contribution to scale) and returns a matrix of scalarizations (Nindivs x Nweights, with higher values indicating greater desirability).

Any other function conforming to these requirements can also be used in place of a Scalarizer, but the provided Scalarizer functions cover the most common use cases.

Scalarizers are constructed from constructor-functions, such as scalarizer_linear() or scalarizer_chebyshev().

See Also

Other Scalarizers: scalarizer_chebyshev(), scalarizer_linear()

---

scalarizer_chebyshev *Chebyshev Scalarizer*

Description

Constructs a Scalarizer that does Chebyshev scalarization, as employed in ParEGO by Knowles (2006).

The Chebyshev scalarization for a single individual with fitness values f and given weight vector w is min(w * f) + rho * sum(w * f), where rho is a hyperparameter given during construction.

Usage

```
scalarizer_chebyshev(rho = 0.05)
```

Arguments

rho	(numeric(1)) Small positive value.

Value

a Scalarizer object.

References

Knowles, Joshua (2006). "ParEGO: A hybrid algorithm with on-line landscape approximation for expensive multiobjective optimization problems." *IEEE Transactions on Evolutionary Computation*, **10**(1), 50–66.

See Also

Other Scalarizers: Scalarizer, scalarizer_linear()

Examples

```
# fitnesses: three rows (i.e. thee indivs) with two objective values each
fitnesses <- matrix(0:5, ncol = 2)

# weights: contains one matrix for each row of 'fitnesses' (i.e. each indiv)
# which get multiplied with their respective row.
weights <- list(
 matrix(c(1, 0, 0, 1), ncol = 2),
 matrix(c(1, 2, 0, 0), ncol = 2),
 matrix(c(0, 1, 0, 1), ncol = 2)
)

sc <- scalarizer_chebyshev()

# The resulting row-vectors are the different scalarizations according to the
# columns in the 'weights' matrices.
sc(fitnesses, weights)

sc <- scalarizer_chebyshev(rho = 0.1)
sc(fitnesses, weights)
```

---

scalarizer_linear *Linear Scalarizer*

Description

Constructs a linear Scalarizer, which performs linear scalarization for ScalorFixedProjection.

Usage

```
scalarizer_linear()
```

Value

a Scalarizer object.

See Also

Other Scalarizers: Scalarizer, scalarizer_chebyshev()

Examples

```
# fitnesses: three rows (i.e. thee indivs) with two objective values each
fitnesses <- matrix(0:5, ncol = 2)

# weights: contains one matrix for each row of 'fitnesses' (i.e. each indiv)
# which get multiplied with their respective row.
weights <- list(
 matrix(c(1, 0, 0, 1), ncol = 2),
 matrix(c(1, 2, 0, 0), ncol = 2),
 matrix(c(0, 1, 0, 1), ncol = 2)
)

sc <- scalarizer_linear()

# The resulting row-vectors are the different scalarizations according to the
# columns in the 'weights' matrices.
sc(fitnesses, weights)
```

Scalor *Scalor Base Class*

Description

Base class representing ranking operations, inheriting from `MiesOperator`.

A `Scalor` gets a table of individuals as input, along with information on the individuals' performance values and returns a vector of a possible scalarization of individuals' fitness (or other qualities).

`Scalor`s can be used by `Selector`s as a basis to select individuals by. This way it is possible to have tournament selection (`SelectorTournament`) or elite selection (`SelectorBest`) based on different, configurable qualities of individuals.

Unlike most other operator types inheriting from `MiesOperator`, the `$operate()` function has two arguments, which are passed on to `$.scale()`

- `values` :: `data.frame`
  Individuals to operate on. Must pass the check of the `Param` given in the last `$prime()` call and may not have any missing components.
- `fitnesses` :: `numeric` | `matrix`
  Fitnesses for each individual given in `values`. If this is a `numeric`, then its length must be equal to the number of rows in `values`. If this is a `matrix`, if number of rows must be equal to the number of rows in `values`, and it must have one column when doing single-crit optimization and one column each for each "criterion" when doing multi-crit optimization.
  Note that fitness values are always *maximized*, both in single- and multi-criterion optimization, so objective output is multiplied with `-1` if it is tagged as `"minimize"`.

The return value of an operation should be a numeric vector with one finite value for each entry of `values`, assigning high values to individuals in some way more "desirable" than others with low values.

Inheriting

Scalor is an abstract base class and should be inherited from. Inheriting classes should implement the private $.scale() function. The user of the object calls $operate(), and the arguments are passed on to private $.scale() after checking that the operator is primed, that the values argument conforms to the primed domain and that other values match. Typically, the $initialize() function should also be overloaded, and optionally the $prime() function; they should call their super equivalents.

Super class

miesmuschel::MiesOperator -> Scalor

Active bindings

supported (character)
: Optimization supported by this Scalor, can be "single-crit", "multi-crit", or both.

Methods

Public methods:

- Scalor$new()
- Scalor$clone()

Method new()**:** Initialize base class components of the Mutator.

Usage:

```
Scalor$new(
  param_classes = c("ParamLgl", "ParamInt", "ParamDbl", "ParamFct"),
  param_set = ps(),
  supported = c("single-crit", "multi-crit"),
  packages = character(0),
  dict_entry = NULL,
  own_param_set = quote(self$param_set)
)
```

Arguments:

param_classes (character)
: Classes of parameters that the operator can handle. May contain any of "ParamLgl", "ParamInt", "ParamDbl", "ParamFct". Default is all of them.
The $param_classes field will reflect this value.

param_set (ParamSet | list of expression)
: Strategy parameters of the operator. This should be created by the subclass and given to super$initialize(). If this is a ParamSet, it is used as the MiesOperator's ParamSet directly. Otherwise it must be a list of expressions e.g. created by alist() that evaluate to ParamSets, possibly referencing self and private. These ParamSet are then combined using a ParamSetCollection. Default is the empty ParamSet.
The $param_set field will reflect this value.

`supported` (`character`)
: Subset of `"single-crit"` and `"multi-crit"`, indicating wether single and / or multi-criterion optimization is supported. Default both of them.
The `$supported` field will reflect this value.

`packages` (`character`) Packages that need to be loaded for the operator to function. This should be declared so these packages can be loaded when operators run on parallel instances. Default is `character(0)`.
The `$packages` field will reflect this values.

`dict_entry` (`character(1)` | `NULL`)
: Key of the class inside the `Dictionary` (usually one of `dict_mutators`, `dict_recombinators`, `dict_selectors`), where it can be retrieved using a short access function. May be `NULL` if the operator is not entered in a dictionary.
The `$dict_entry` field will reflect this value.

`own_param_set` (`language`)
: An expression that evaluates to a `ParamSet` indicating the configuration parameters that are entirely owned by this operator class (and not proxied from a construction argument object). This should be `quote(self$param_set)` (the default) when the `param_set` argument is not a list of expressions.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Scalor$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other base classes: `FiltorSurrogate`, `Filtor`, `MiesOperator`, `MutatorDiscrete`, `MutatorNumeric`, `Mutator`, `OperatorCombination`, `RecombinatorPair`, `Recombinator`, `SelectorScalar`, `Selector`

Other scalors: `dict_scalors_aggregate`, `dict_scalors_domcount`, `dict_scalors_fixedprojection`, `dict_scalors_hypervolume`, `dict_scalors_nondom`, `dict_scalors_one`, `dict_scalors_proxy`, `dict_scalors_single`

`Selector` *Selector Base Class*

Description

Base class representing selection operations, inheriting from `MiesOperator`.

A `Selector` gets a table of individuals as input, along with information on the individuals' performance values and the number of individuals to select, and returns a vector of integers indicating which individuals were selected.

Selection operations are performed in ES algorithms to facilitate concentration towards individuals that perform well with regard to the fitness measure.

Fitness values are always *maximized*, both in single- and multi-criterion optimization.

Unlike most other operator types inheriting from `MiesOperator`, the `$operate()` function has three arguments, which are passed on to `$.select()`

- `values` :: `data.frame`
  Individuals to operate on. Must pass the check of the `Param` given in the last `$prime()` call and may not have any missing components.
- `fitnesses` :: `numeric` | `matrix`
  Fitnesses for each individual given in `values`. If this is a `numeric`, then its length must be equal to the number of rows in `values`. If this is a `matrix`, if number of rows must be equal to the number of rows in `values`, and it must have one column when doing single-crit optimization and one column each for each "criterion" when doing multi-crit optimization.
  The `fitnesses`-value passed on to `$.select()` is always a `matrix`.
- `n_select` :: `integer(1)`
  Number of individuals to select. Some `Selector`s select individuals with replacement, for which this value may be greater than the number of rows in `values`.
- `group_size` :: `integer`
  Sampling group size hint, indicating that the caller would prefer there to not be any duplicates within this group size, e.g. because the `Selector` is called to select individuals to be given to a `Recombinator` with a certain `n_indivs_in`, or because it is called as a `survival_selector` in `mies_survival_comma()` or `mies_survival_plus()`. The `Selector` may or may not ignore this value, however. This may possibly happen because of certain configuration parameters, or because the input size is too small.
  Must either be a scalar value or sum up to `n_select`. Must be non-negative. A scalar value of 0 is interpreted the same as 1.
  If not given, this value defaults to 1.

The return value for an operation will be a numeric vector of integer values of length `n_select` indexing the individuals that were selected. Some `Selector`s select individuals with replacement, for which the return value may contain indices more than once.

Inheriting

`Selector` is an abstract base class and should be inherited from. Inheriting classes should implement the private `$.select()` function. The user of the object calls `$operate()`, and the arguments are passed on to private `$.select()` after checking that the operator is primed, that the `values` argument conforms to the primed domain and that other values match. Typically, the `$initialize()` function should also be overloaded, and optionally the `$prime()` function; they should call their `super` equivalents.

Super class

`miesmuschel::MiesOperator` -> `Selector`

Active bindings

`supported` (character)
: Optimization supported by this `Selector`, can be `"single-crit"`, `"multi-crit"`, or both.

Methods

Public methods:

- `Selector$new()`
- `Selector$clone()`

Method `new()`**:** Initialize base class components of the `Selector`.

Usage:

```
Selector$new(
  is_deterministic = FALSE,
  param_classes = c("ParamLgl", "ParamInt", "ParamDbl", "ParamFct"),
  param_set = ps(),
  supported = c("single-crit", "multi-crit"),
  packages = character(0),
  dict_entry = NULL,
  own_param_set = quote(self$param_set)
)
```

Arguments:

`is_deterministic` (`logical(1)`)
: Whether the `Selector` is deterministic. Setting this to `TRUE` adds a configuration parameter `shuffle_selection` (initialized to `TRUE`) that causes the selection to be shuffled.

`param_classes` (`character`)
: Classes of parameters that the operator can handle. May contain any of `"ParamLgl"`, `"ParamInt"`, `"ParamDbl"`, `"ParamFct"`. Default is all of them.
The `$param_classes` field will reflect this value.

`param_set` (`ParamSet` | `list` of `expression`)
: Strategy parameters of the operator. This should be created by the subclass and given to `super$initialize()`. If this is a `ParamSet`, it is used as the `MiesOperator`'s `ParamSet` directly. Otherwise it must be a `list` of expressions e.g. created by `alist()` that evaluate to `ParamSet`s, possibly referencing `self` and `private`. These `ParamSet` are then combined using a `ParamSetCollection`. Default is the empty `ParamSet`.
The `$param_set` field will reflect this value.

`supported` (`character`)
: Subset of `"single-crit"` and `"multi-crit"`, indicating wether single and / or multi-criterion optimization is supported. Default both of them.
The `$supported` field will reflect this value.

`packages` (`character`) Packages that need to be loaded for the operator to function. This should be declared so these packages can be loaded when operators run on parallel instances. Default is `character(0)`.
The `$packages` field will reflect this values.

`dict_entry` (`character(1)` | `NULL`)
: Key of the class inside the `Dictionary` (usually one of `dict_mutators`, `dict_recombinators`, `dict_selectors`), where it can be retrieved using a short access function. May be `NULL` if the operator is not entered in a dictionary.
The `$dict_entry` field will reflect this value.

`own_param_set` (`language`)
: An expression that evaluates to a `ParamSet` indicating the configuration parameters that are

entirely owned by this operator class (and not proxied from a construction argument object). This should be `quote(self$param_set)` (the default) when the `param_set` argument is not a list of expressions.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Selector$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other base classes: `FiltorSurrogate`, `Filtor`, `MiesOperator`, `MutatorDiscrete`, `MutatorNumeric`, `Mutator`, `OperatorCombination`, `RecombinatorPair`, `Recombinator`, `Scalor`, `SelectorScalar`

Other selectors: `SelectorScalar`, `dict_selectors_best`, `dict_selectors_maybe`, `dict_selectors_null`, `dict_selectors_proxy`, `dict_selectors_random`, `dict_selectors_sequential`, `dict_selectors_tournament`

`SelectorScalar` *Selector making use of Scalors*

Description

Base class inheriting from `Selector` for selection operations that make use of scalar values, generated by `Scalor`.

Inheriting

`SelectorScaling` is an abstract base class and should be inherited from. Inheriting classes should implement the private `$.select_scalar()` function. During `$operate()`, the `$.select_scalar()` function is called, it should have three arguments, similar to `Selector`'s `$.select()` function. `values` and `n_select` are as given to `$.select()` of the `Selector`. The `fitnesses` argument is first scaled by the associated `Scalor` and then passed on as a `numeric` vector.

Typically, `$initialize()` should also be overloaded when inheriting.

Super classes

`miesmuschel::MiesOperator` -> `miesmuschel::Selector` -> `SelectorScalar`

Active bindings

`scalor` (`Scalor`)
    `Scalor` used to scalarize fitnesses for selection.

Methods

Public methods:

- `SelectorScalar$new()`
- `SelectorScalar$prime()`
- `SelectorScalar$clone()`

Method `new()`**:** Initialize base class components of the `SelectorScalar`.

Usage:

```
SelectorScalar$new(
  scalor = ScalorSingleObjective$new(),
  is_deterministic = FALSE,
  param_classes = c("ParamLgl", "ParamInt", "ParamDbl", "ParamFct"),
  param_set = ps(),
  supported = scalor$supported,
  packages = character(0),
  dict_entry = NULL
)
```

Arguments:

`scalor` (`Scalor`)
: `Scalor` to use to generate scalar values from multiple objectives, if multi-objective optimization is performed. Initialized to `ScalorSingleObjective`: Doing single-objective optimization normally, throwing an error if used in multi-objective setting: In that case, a `Scalor` needs to be explicitly chosen.

`is_deterministic` (`logical(1)`)
: Whether the `Selector` is deterministic. Setting this to `TRUE` adds a configuration parameter `shuffle_selection` (initialized to `TRUE`) that causes the selection to be shuffled.

`param_classes` (`character`)
: Classes of parameters that the operator can handle. May contain any of `"ParamLgl"`, `"ParamInt"`, `"ParamDbl"`, `"ParamFct"`. Default is all of them.
The `$param_classes` field will reflect this value.

`param_set` (`ParamSet` | `list` of `expression`)
: Strategy parameters of the operator. This should be created by the subclass and given to `super$initialize()`. If this is a `ParamSet`, it is used as the `MiesOperator`'s `ParamSet` directly. Otherwise it must be a `list` of expressions e.g. created by `alist()` that evaluate to `ParamSet`s, possibly referencing `self` and `private`. These `ParamSet` are then combined using a `ParamSetCollection`. Default is the empty `ParamSet`.
The `$param_set` field will reflect this value.

`supported` (`character`)
: Subset of `"single-crit"` and `"multi-crit"`, indicating wether single and / or multi-criterion optimization is supported. Default to the supported set of `scalor`.
The `$supported` field will reflect this value.

`packages` (`character`) Packages that need to be loaded for the operator to function. This should be declared so these packages can be loaded when operators run on parallel instances. Default is `character(0)`.
The `$packages` field will reflect this values.

dict_entry (character(1) | NULL)
Key of the class inside the Dictionary (usually one of dict_mutators, dict_recombinators, dict_selectors), where it can be retrieved using a short access function. May be NULL if the operator is not entered in a dictionary.
The $dict_entry field will reflect this value.

Method prime(): See MiesOperator method. Primes both this operator, as well as the wrapped operator given to scalor during construction.

Usage:

```
SelectorScalar$prime(param_set)
```

Arguments:

param_set (ParamSet)
Passed to MiesOperator$prime().

Returns: invisible self.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
SelectorScalar$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other base classes: FiltorSurrogate, Filtor, MiesOperator, MutatorDiscrete, MutatorNumeric, Mutator, OperatorCombination, RecombinatorPair, Recombinator, Scalor, Selector

Other selectors: Selector, dict_selectors_best, dict_selectors_maybe, dict_selectors_null, dict_selectors_proxy, dict_selectors_random, dict_selectors_sequential, dict_selectors_tournament

---

terminator_get_generations *Get the Numger of Generations that a Terminator Allows*

Description

Get the number of generations of a TerminatorGenerations. When the TerminatorGenerations is wrapped in a TerminatorCombo, then the minimum number of generations allowed by it are retrieved. This is the minimum of all terminator_get_generations if $any is set to TRUE, and the maximum if $any is set to FALSE.

The number of generations allowed by other Terminators is infinity.

Usage

```
terminator_get_generations(x)
```

Arguments

`x`	(`Terminator`) `Terminator` to query.

Value

`numeric(1)`: The theoretical maximum number of generations allowed by the `Terminator`.

Index

∗ **Scalarizers**
  Scalarizer, 163
  scalarizer_chebyshev, 163
  scalarizer_linear, 164
∗ **base classes**
  Filtor, 92
  FiltorSurrogate, 94
  MiesOperator, 97
  Mutator, 132
  MutatorDiscrete, 134
  MutatorNumeric, 136
  OperatorCombination, 138
  Recombinator, 156
  RecombinatorPair, 159
  Scalor, 165
  Selector, 167
  SelectorScalar, 170
∗ **datasets**
  dict_filtors, 6
  dict_mutators, 18
  dict_recombinators, 35
  dict_scalors, 59
  dict_selectors, 74
∗ **dictionaries**
  dict_filtors, 6
  dict_mutators, 18
  dict_recombinators, 35
  dict_scalors, 59
  dict_selectors, 74
  mut, 131
∗ **filtor wrappers**
  dict_filtors_maybe, 7
  dict_filtors_proxy, 11
∗ **filtors**
  dict_filtors_maybe, 7
  dict_filtors_null, 10
  dict_filtors_proxy, 11
  dict_filtors_surprog, 13
  dict_filtors_surtour, 16
  Filtor, 92
  FiltorSurrogate, 94
∗ **mies building blocks**
  mies_evaluate_offspring, 103
  mies_generate_offspring, 107
  mies_get_fitnesses, 111
  mies_init_population, 114
  mies_select_from_archive, 119
  mies_step_fidelity, 122
  mies_survival_comma, 124
  mies_survival_plus, 126
∗ **mutator wrappers**
  dict_mutators_cmpmaybe, 19
  dict_mutators_maybe, 25
  dict_mutators_proxy, 29
  dict_mutators_sequential, 30
  OperatorCombination, 138
∗ **mutators**
  dict_mutators_cmpmaybe, 19
  dict_mutators_erase, 22
  dict_mutators_gauss, 23
  dict_mutators_maybe, 25
  dict_mutators_null, 27
  dict_mutators_proxy, 29
  dict_mutators_sequential, 30
  dict_mutators_unif, 33
  Mutator, 132
  MutatorDiscrete, 134
  MutatorNumeric, 136
  OperatorCombination, 138
∗ **optimizers**
  OptimizerMies, 145
∗ **recombinator wrappers**
  dict_recombinators_cmpmaybe, 36
  dict_recombinators_maybe, 42
  dict_recombinators_proxy, 47
  dict_recombinators_sequential, 51
  OperatorCombination, 138
∗ **recombinators**

dict_recombinators_cmpmaybe, 36
dict_recombinators_convex, 38
dict_recombinators_cvxpair, 40
dict_recombinators_maybe, 42
dict_recombinators_null, 45
dict_recombinators_proxy, 47
dict_recombinators_sbx, 49
dict_recombinators_sequential, 51
dict_recombinators_swap, 54
dict_recombinators_xonary, 55
dict_recombinators_xounif, 57
OperatorCombination, 138
Recombinator, 156
RecombinatorPair, 159
∗ **scalor wrappers**
dict_scalors_aggregate, 59
dict_scalors_fixedprojection, 64
dict_scalors_proxy, 70
∗ **scalors**
dict_scalors_aggregate, 59
dict_scalors_domcount, 62
dict_scalors_fixedprojection, 64
dict_scalors_hypervolume, 66
dict_scalors_nondom, 67
dict_scalors_one, 69
dict_scalors_proxy, 70
dict_scalors_single, 72
Scalor, 165
∗ **selector wrappers**
dict_selectors_maybe, 76
dict_selectors_proxy, 81
dict_selectors_sequential, 85
∗ **selectors**
dict_selectors_best, 74
dict_selectors_maybe, 76
dict_selectors_null, 80
dict_selectors_proxy, 81
dict_selectors_random, 83
dict_selectors_sequential, 85
dict_selectors_tournament, 87
Selector, 167
SelectorScalar, 170

Archive, *66*, *110*, *129, 130*

bbotk::Optimizer, *149*
bbotk::Terminator, *128*, *130*
bitwise crossover, *159*

call, *99*, *161*
Condition, *155*
crate_env, 5

data.table, *98–102*, *104*, *106–108*, *113*, *120*, *123*, *125*, *127*, *140, 141*, *154*
Design, *22*, *115*, *148*
dict_filtors, 6, *8*, *10*, *15*, *17*, *19*, *35*, *59*, *74*, *78*, *132*
dict_filtors_maybe, 7, *11*, *13*, *16*, *18*, *94*, *97*
dict_filtors_null, *9*, 10, *13*, *16*, *18*, *94*, *97*
dict_filtors_proxy, *9*, *11*, 11, *16*, *18*, *94*, *97*
dict_filtors_surprog, *9*, *11*, *13*, 13, *18*, *94*, *97*
dict_filtors_surtour, *9*, *11*, *13*, *16*, 16, *94*, *97*
dict_mutators, *7*, 18, *20*, *22*, *24, 25*, *28, 29*, *31*, *34, 35*, *59*, *74*, *94*, *96*, *99*, *131–133*, *135*, *137*, *139*, *141*, *158*, *160*, *167*, *169*, *172*
dict_mutators_cmpmaybe, 19, *23, 24*, *27, 28*, *30*, *32*, *34*, *134*, *136, 137*, *144*
dict_mutators_combine (OperatorCombination), 138
dict_mutators_erase, *21*, 22, *24*, *27, 28*, *30*, *32*, *34*, *134*, *136, 137*, *144*
dict_mutators_gauss, *21*, *23*, 23, *27, 28*, *30*, *32*, *34*, *134*, *136, 137*, *144*
dict_mutators_maybe, *21*, *23, 24*, 25, *28*, *30*, *32*, *34*, *134*, *136, 137*, *144*
dict_mutators_null, *21*, *23, 24*, *27*, 27, *30*, *32*, *34*, *134*, *136, 137*, *144*
dict_mutators_proxy, *21*, *23, 24*, *27, 28*, 29, *32*, *34*, *134*, *136, 137*, *144*
dict_mutators_sequential, *21*, *23, 24*, *27, 28*, *30*, 30, *34*, *134*, *136, 137*, *144*
dict_mutators_unif, *21*, *23, 24*, *27, 28*, *30*, *32*, 33, *134*, *136, 137*, *144*
dict_recombinators, *7*, *19*, 35, *36*, *39*, *41*, *43*, *45*, *47*, *50*, *52*, *54*, *56*, *58, 59*, *74*, *94*, *96*, *99*, *131–133*, *135*, *137*, *139*, *141*, *158*, *160*, *167*, *169*, *172*
dict_recombinators_cmpmaybe, 36, *40*, *42*, *44*, *46*, *49, 50*, *53*, *55*, *57, 58*, *144*, *158*, *160*
dict_recombinators_combine (OperatorCombination), 138
dict_recombinators_convex, *38*, 38, *42*, *44*, *46*, *49, 50*, *53*, *55*, *57, 58*, *144*, *158*,

160
dict_recombinators_cvxpair, 38, 40, 40, 44, 46, 49, 50, 53, 55, 57, 58, 144, 158, 160
dict_recombinators_maybe, 38, 40, 42, 42, 46, 49, 50, 53, 55, 57, 58, 144, 158, 160
dict_recombinators_null, 38, 40, 42, 44, 45, 49, 50, 53, 55, 57, 58, 144, 158, 160
dict_recombinators_proxy, 38, 40, 42, 44, 46, 47, 50, 53, 55, 57, 58, 144, 158, 160
dict_recombinators_sbx, 38, 40, 42, 44, 46, 49, 49, 53, 55, 57, 58, 144, 158, 160
dict_recombinators_sequential, 38, 40, 42, 44, 46, 49, 50, 51, 55, 57, 58, 144, 158, 160
dict_recombinators_swap, 38, 40, 42, 44, 46, 49, 50, 53, 54, 57, 58, 144, 158, 160
dict_recombinators_xonary, 38, 40, 42, 44, 46, 49, 50, 53, 55, 55, 58, 144, 158, 160
dict_recombinators_xounif, 38, 40, 42, 44, 46, 49, 50, 53, 55, 57, 57, 144, 158, 160
dict_scalors, 7, 19, 35, 59, 60, 62, 64, 66, 68, 69, 71, 73, 74, 132
dict_scalors_aggregate, 59, 63, 65, 67, 68, 70, 72, 73, 167
dict_scalors_domcount, 61, 62, 65, 67, 68, 70, 72, 73, 167
dict_scalors_fixedprojection, 61, 63, 64, 67, 68, 70, 72, 73, 167
dict_scalors_hypervolume, 61, 63, 65, 66, 68, 70, 72, 73, 167
dict_scalors_nondom, 61, 63, 65, 67, 67, 70, 72, 73, 167
dict_scalors_one, 61, 63, 65, 67, 68, 69, 72, 73, 167
dict_scalors_proxy, 61, 63, 65, 67, 68, 70, 70, 73, 167
dict_scalors_single, 61, 63, 65, 67, 68, 70, 72, 72, 167
dict_selectors, 7, 12, 19, 35, 59, 74, 75, 80, 82, 83, 86, 88, 94, 96, 99, 131, 132, 134, 135, 137, 141, 158, 160, 167, 169, 172
dict_selectors_best, 74, 79, 81, 82, 84, 87, 89, 170, 172
dict_selectors_maybe, 76, 76, 81, 82, 84, 87, 89, 170, 172
dict_selectors_null, 76, 79, 80, 82, 84, 87, 89, 170, 172
dict_selectors_proxy, 76, 79, 81, 81, 84, 87, 89, 170, 172
dict_selectors_random, 76, 79, 81, 82, 83, 87, 89, 170, 172
dict_selectors_sequential, 76, 79, 81, 82, 84, 85, 89, 170, 172
dict_selectors_tournament, 76, 79, 81, 82, 84, 87, 87, 170, 172
Dictionary, 7, 19, 35, 59, 74, 94, 96–99, 133, 135, 137, 141, 158, 160, 167, 169, 172
dictionary, 8, 10, 12, 15, 17, 20, 22, 24, 25, 28, 29, 31, 34, 36, 39, 41, 43, 45, 47, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 69, 71, 73, 75, 78, 80, 82, 83, 86, 88, 128, 130–132, 139
dist_crowding, 89
domhv_contribution, 90
domhv_improvement, 91

environment, 161
environmentName(), 5
eval(), 161

Filtor, 7–13, 15–18, 78, 92, 92, 94, 97, 101, 106, 118, 134, 136, 137, 144, 158, 160, 167, 170, 172
FiltorMaybe (dict_filtors_maybe), 7
FiltorNull, 9, 106
FiltorNull (dict_filtors_null), 10
FiltorProxy, 10
FiltorProxy (dict_filtors_proxy), 11
FiltorSurrogate, 9, 11, 13, 14, 16–18, 94, 94, 101, 134, 136, 137, 144, 158, 160, 167, 170, 172
FiltorSurrogateProgressive (dict_filtors_surprog), 13
FiltorSurrogateTournament (dict_filtors_surtour), 16
ftr (mut), 131
ftr(), 8, 10, 15, 17, 78
ftrs (mut), 131

ftrs(), *8*, *10*, *15*, *17*, *78*
function, *64*

generate_design_random(), *22*, *148*

invisible, *9*, *13*, *21*, *26*, *30*, *32*, *37*, *44*, *48*, *53*, *61*, *65*, *72*, *79*, *82*, *87*, *96*, *100*, *104*, *116*, *117*, *123*, *142*, *172*

Learner, *95*
LearnerRegr, *13*

mies_aggregate_generations, 101
mies_evaluate_offspring, 103, *108*, *110*, *111*, *116*, *120*, *123*, *125*, *127*
mies_evaluate_offspring(), *107*, *108*, *148*
mies_filter_offspring, 106
mies_generate_offspring, *38*, *55*, *104*, 107, *111*, *116*, *120*, *123*, *125*, *127*
mies_generate_offspring(), *75*, *80*, *87*, *147*, *149*, *150*
mies_generation, 109, *147*
mies_get_fitnesses, *66*, *104*, *108*, 111, *115*, *116*, *120*, *123*, *125*, *127*
mies_get_fitnesses(), *108*, *120*, *125*
mies_get_generation_results, 112
mies_init_population, *104*, *108*, *110*, *111*, 114, *120*, *123*, *125*, *127*
mies_init_population(), *22*, *148*, *150*
mies_prime_operators, *108*, 117
mies_prime_operators(), *116*
mies_select_from_archive, *104*, *108*, *111*, *116*, 119, *123*, *125*, *127*
mies_step_fidelity, *104*, *108*, *111*, *116*, *120*, 122, *125*, *127*
mies_step_fidelity(), *148*, *149*
mies_survival_comma, *104*, *108*, *111*, *116*, *120*, *123*, 124, *127*
mies_survival_comma(), *87*, *148–150*, *168*
mies_survival_plus, *104*, *108*, *111*, *116*, *120*, *123*, *125*, 126
mies_survival_plus(), *87*, *148–150*, *168*
miesmuschel *(*miesmuschel-package*)*, 4
miesmuschel-package, 4
miesmuschel::Filtor, *8*, *10*, *12*, *15*, *17*, *95*
miesmuschel::FiltorSurrogate, *15*, *17*
miesmuschel::MiesOperator, *8*, *10*, *12*, *15*, *17*, *20*, *22*, *24*, *26*, *28*, *29*, *31*, *34*, *36*, *39*, *41*, *43*, *46*, *48*, *50*, *52*, *54*, *56*, *60*, *63*, *64*, *66*, *68*, *69*, *71*, *73*, *75*, *78*, *80*, *82*, *84*, *86*, *88*, *93*, *95*, *133*, *134*, *136*, *139*, *142*, *143*, *157*, *159*, *166*, *168*, *170*
miesmuschel::Mutator, *20*, *22*, *24*, *26*, *28*, *29*, *31*, *34*, *134*, *136*
miesmuschel::MutatorDiscrete, *34*
miesmuschel::MutatorNumeric, *24*
miesmuschel::OperatorCombination, *142*, *143*
miesmuschel::Recombinator, *36*, *39*, *41*, *43*, *46*, *48*, *50*, *52*, *54*, *56*, *159*
miesmuschel::RecombinatorPair, *41*, *50*, *54*
miesmuschel::Scalor, *60*, *63*, *64*, *66*, *68*, *69*, *71*, *73*
miesmuschel::Selector, *75*, *78*, *80*, *82*, *84*, *86*, *88*, *170*
miesmuschel::SelectorScalar, *75*, *88*
MiesOperator, *9*, *12*, *21*, *26*, *30–32*, *37*, *44*, *48*, *52*, *53*, *61*, *65*, *71*, *79*, *82*, *87*, *92*, *94*, *96*, *97*, 97, *117*, *132*, *134*, *136–142*, *144*, *146–148*, *150*, *156*, *158*, *160*, *165*, *167*, *168*, *170*, *172*
mlr3::Learner, *94*
mlr3::LearnerRegr, *15*, *17*, *18*, *95*
mlr3::mlr_sugar, *131*
mlr3misc::dictionary_sugar_get(), *132*
mlr3tuning::Tuner, *151*
mlr3tuning::TunerFromOptimizer, *151*
mlr_terminators, *128*, *130*
mlr_terminators_budget, 128
mlr_terminators_gens, 129
mut, *7*, *19*, *35*, *59*, *74*, 131
mut(), *20*, *22*, *24*, *25*, *28*, *29*, *31*, *34*, *139*
Mutator, *19–32*, *34*, *94*, *97*, *101*, *108*, *117*, *118*, *132*, 132, *134*, *136–139*, *144*, *149–151*, *158*, *160*, *167*, *170*, *172*
MutatorCmpMaybe, *22*, *33*
MutatorCmpMaybe *(*dict_mutators_cmpmaybe*)*, 19
MutatorCombination, *27*, *97*, *141*
MutatorCombination *(*OperatorCombination*)*, 138
MutatorDiscrete, *21*, *23*, *24*, *27*, *28*, *30*, *32*, *34*, *94*, *97*, *101*, *133*, *134*, 134, *137*, *144*, *158*, *160*, *167*, *170*, *172*
MutatorDiscreteUniform

(dict_mutators_unif), 33
MutatorErase (dict_mutators_erase), 22
MutatorGauss, 136
MutatorGauss (dict_mutators_gauss), 23
MutatorMaybe, 27
MutatorMaybe (dict_mutators_maybe), 25
MutatorNull, 20, 26, 108
MutatorNull (dict_mutators_null), 27
MutatorNumeric, 21, 23, 24, 27, 28, 30, 32, 34, 94, 97, 101, 133, 134, 136, 136, 144, 158, 160, 167, 170, 172
MutatorProxy, 147, 150
MutatorProxy (dict_mutators_proxy), 29
MutatorSequential (dict_mutators_sequential), 30
muts (mut), 131
muts(), 20, 22, 24, 25, 28, 29, 31, 34, 139

Objective, 117, 118, 147
OperatorCombination, 21, 23, 24, 27, 28, 30, 32, 34, 38, 40, 42, 44, 46, 49, 50, 53, 55, 57, 58, 94, 97, 101, 116, 134, 136, 137, 138, 146, 147, 158, 160, 167, 170, 172
OptimInstance, 101, 103, 104, 106, 107, 110–116, 118–120, 122, 124, 126, 146–148, 150
Optimizer, 4, 103, 146
OptimizerMies, 4, 29, 47, 81, 103, 129, 130, 145

paradox::generate_design_grid, 115
paradox::generate_design_lhs, 22, 115, 148
paradox::generate_design_random, 22, 115, 148
paradox::ParamSet, 132, 153
paradox::Sampler, 162
Param, 8, 10, 12, 19, 22, 24, 25, 27, 29, 31, 34, 36, 39, 41, 43, 45, 47, 49, 52, 54, 56, 58, 62, 64, 66, 68, 69, 71, 72, 75, 78, 80, 81, 83, 86, 88, 92, 95, 100, 117, 138, 140, 150, 153, 154, 165, 168
ParamDbl, 10, 12, 22, 24, 27, 29, 39, 41, 45, 47, 49, 54, 56, 58, 62, 64, 66, 68, 69, 71, 72, 75, 80, 81, 83, 88, 138
ParamFct, 10, 12, 22, 27, 29, 34, 45, 47, 54, 58, 62, 64, 66, 68, 69, 71, 72, 75, 80, 81, 83, 88, 139, 141
ParamInt, 10, 12, 22, 24, 27, 29, 45, 47, 49, 54, 58, 62, 64, 66, 68, 69, 71, 72, 75, 80, 81, 83, 88, 138
ParamLgl, 10, 12, 22, 27, 29, 34, 45, 47, 54, 58, 62, 64, 66, 68, 69, 71, 72, 75, 80, 81, 83, 88, 138, 139, 141
ParamSet, 9, 12, 21, 23, 26, 30, 32, 37, 44, 48, 49, 53, 61, 65, 71, 79, 82, 87, 93, 94, 96–100, 108, 118, 123, 133–135, 137, 139, 141, 142, 147, 149, 153–155, 157, 158, 160, 161, 166, 167, 169, 171, 172
ParamSetCollection, 93, 96, 99, 133, 135, 137, 158, 160, 166, 169, 171
ParamSetShadow, 153
ParamUty, 64, 162

R6, 161
rank_nondominated, 62, 156
rec (mut), 131
rec(), 36, 39, 41, 43, 45, 47, 50, 52, 54, 56, 58, 139
rec(cmpmaybe), 49
Recombinator, 36–58, 87, 94, 97, 101, 108, 117, 118, 132, 134, 136–139, 144, 149–151, 156, 159, 160, 167, 168, 170, 172
RecombinatorCmpMaybe, 54, 57
RecombinatorCmpMaybe (dict_recombinators_cmpmaybe), 36
RecombinatorCombination, 45, 46, 97, 138, 141
RecombinatorCombination (OperatorCombination), 138
RecombinatorConvex (dict_recombinators_convex), 38
RecombinatorConvexPair (dict_recombinators_cvxpair), 40
RecombinatorCrossoverNary (dict_recombinators_xonary), 55
RecombinatorCrossoverUniform, 54
RecombinatorCrossoverUniform (dict_recombinators_xounif), 57
RecombinatorMaybe, 45, 46
RecombinatorMaybe (dict_recombinators_maybe), 42
RecombinatorNull, 37, 44, 51, 108

RecombinatorNull
(dict_recombinators_null), 45
RecombinatorPair, *38*, *40*, *42*, *44*, *46*, *49, 50*, *53*, *55*, *57, 58*, *94*, *97*, *101*, *134*, *136*, *137*, *144*, *158*, 159, *167*, *170*, *172*
RecombinatorProxy, *147*, *150*
RecombinatorProxy
(dict_recombinators_proxy), 47
RecombinatorSequential
(dict_recombinators_sequential), 51
RecombinatorSimulatedBinaryCrossover
(dict_recombinators_sbx), 49
RecombinatorSwap, *57*
RecombinatorSwap
(dict_recombinators_swap), 54
recs (mut), 131
recs(), *36*, *39*, *41*, *43*, *45*, *47*, *50*, *52*, *54*, *56*, *58*, *139*
repr, 161

Sampler, *116*, *147, 148*
SamplerRandomWeights, 162
SBX crossover, *159*
Scalarizer, *64*, *163*, 163, *164*
scalarizer_chebyshev, *163*, 163, *164*
scalarizer_chebyshev(), *64*, *163*
scalarizer_linear, *163, 164*, 164
scalarizer_linear(), *64*, *163*
Scalor, *59–73*, *75*, *88*, *94*, *97*, *101*, *132*, *134*, *136, 137*, *144*, *158*, *160*, *165*, 165, *170–172*
ScalorAggregate
(dict_scalors_aggregate), 59
ScalorDomcount (dict_scalors_domcount), 62
ScalorFixedProjection, *162*, *164*
ScalorFixedProjection
(dict_scalors_fixedprojection), 64
ScalorHypervolume
(dict_scalors_hypervolume), 66
ScalorNondom (dict_scalors_nondom), 67
ScalorOne, *72*
ScalorOne (dict_scalors_one), 69
ScalorProxy (dict_scalors_proxy), 70
ScalorSingleObjective, *71*, *75*, *88*, *171*
ScalorSingleObjective
(dict_scalors_single), 72

scl (mut), 131
scl(), *60*, *62*, *64*, *66*, *68, 69*, *71*, *73*
scls (mut), 131
scls(), *60*, *62*, *64*, *66*, *68, 69*, *71*, *73*
sel (mut), 131
sel(), *12*, *75*, *80*, *82, 83*, *86*, *88*
sel(best), *39*, *55*
sel(tournament), *39*, *55*
Selector, *12*, *15*, *18*, *72*, *74–89*, *94–97*, *101*, *108*, *111*, *115*, *117–120*, *124–127*, *132*, *134*, *136, 137*, *144*, *149–151*, *158*, *160*, *165*, *167*, 167, *168–172*
SelectorBest, *70*, *81*, *108*, *115*, *120*, *165*
SelectorBest (dict_selectors_best), 74
SelectorMaybe (dict_selectors_maybe), 76
SelectorNull (dict_selectors_null), 80
SelectorProxy, *147*, *150*
SelectorProxy (dict_selectors_proxy), 81
SelectorRandom, *79*
SelectorRandom (dict_selectors_random), 83
SelectorScalar, *76*, *79*, *81, 82*, *84*, *87*, *89*, *94*, *97*, *101*, *134*, *136, 137*, *144*, *158*, *160*, *167*, *170*, 170
SelectorSequential
(dict_selectors_sequential), 85
SelectorTournament, *165*
SelectorTournament
(dict_selectors_tournament), 87
sels (mut), 131
sels(), *12*, *75*, *80*, *82, 83*, *86*, *88*
short access function, *94*, *96*, *99*, *134, 135*, *137*, *141*, *158*, *160*, *167*, *169*, *172*

Terminator, *128–130*, *146*, *172, 173*
terminator_get_generations, 172
TerminatorBudget
(mlr_terminators_budget), 128
TerminatorCombo, *172*
TerminatorGenerations, *146*, *172*
TerminatorGenerations
(mlr_terminators_gens), 129
trm(), *128*, *130*
trms(), *128*, *130*
Tuner, *4*
TunerMies, *4*
TunerMies (OptimizerMies), 145

utils::help(), *100*