Package ‘neighbours’

December 17, 2025

Type Package

Title Neighbourhood Functions for Local-Search Algorithms

Version 0.1-5

Date 2025-12-17

Maintainer Enrico Schumann <es@enricoschumann.net>

Description Neighbourhood functions are key components of local-search algorithms such as Simulated Annealing or Threshold Accepting. These functions take a solution and return a slightly-modified copy of it, i.e. a neighbour. The package provides a function neighbourfun() that constructs such neighbourhood functions, based on parameters such as admissible ranges for elements in a solution. Supported are numeric and logical solutions. The algorithms were originally created for portfolio-optimisation applications, but can be used for other models as well. Several recipes for neighbour computations are taken from ``Numerical Methods and Optimization in Finance'' by M. Gilli, D. Maringer and E. Schumann (2019, ISBN:978-0128150658).

License GPL-3

URL https://enricoschumann.net/R/packages/neighbours/ , https://sr.ht/~enricoschumann/neighbours/ , https://github.com/enricoschumann/neighbours

Depends R (>= 3.3)

Suggests NMOF, quadprog, tinytest

NeedsCompilation no

Author Enrico Schumann [aut, cre] (ORCID: <https://orcid.org/0000-0001-7601-6576>)

Repository CRAN

Date/Publication 2025-12-17 10:20:14 UTC

Contents

`compare_vectors` *Compare Vectors*

Description

Compare numeric or logical vectors.

Usage

```
compare_vectors(..., sep = "", diff.char = "|")
```

Arguments

`...`	vectors of the same length
`sep`	a string
`diff.char`	a single character

Details

The function compares vectors with one another. The main purpose is to print a useful representation of differences (and return differences, usually invisibly).

The function is still experimental and will likely change.

Value

depends on how the function is called; typically a list

Author(s)

Enrico Schumann

See Also

`neighbourfun`

Examples

```
x <- runif(5) > 0.5
nb <- neighbourfun(type = "logical")

compare_vectors(x, nb(x))
## 01010
##  |
## 00010
## The vectors differ in  1  place.

nb <- neighbourfun(type = "logical", stepsize = 2)
compare_vectors(x, nb(x))
## 01010
## |   |
## 11011
## The vectors differ in  2  places.
```

`neighbourfun` *Neighbourhood Functions*

Description

Create neighbourhood functions, including constraints.

Usage

```
neighbourfun(min = 0, max = 1, kmin = NULL, kmax = NULL,
             stepsize, sum = TRUE, random = TRUE, update = FALSE,
             type = "numeric", active = TRUE, length = NULL,
             A = NULL, ...)

neighborfun (min = 0, max = 1, kmin = NULL, kmax = NULL,
             stepsize, sum = TRUE, random = TRUE, update = FALSE,
             type = "numeric", active = TRUE, length = NULL,
             A = NULL, ...)
```

Arguments

`min`	a numeric vector. A scalar is recycled to `length`, if specified.
`max`	a numeric vector. A scalar is recycled to `length`, if specified.
`kmin`	`NULL` or `integer`: the minimum number of `TRUE` values in logical vectors
`kmax`	`NULL` or `integer`: the maximum number of `TRUE` values in logical vectors
`stepsize`	numeric. For numeric neighbourhoods, the (average) stepsize. For logical neighbourhoods, the number of elements that are changed.
`sum`	logical or numeric. If specified and of length 1, only zero-sum changes will be applied to a numeric vector (i.e. the sum over all elements in a solution remains unchanged).

`random`	logical. Should the stepsize be random or fixed?
`active`	a vector: either the positions of elements that may be changed, or a logical vector. The default is a length-one logical vector, which means that all elements may be changed.
`update`	either `logical` (the default `FALSE`) or a string, specifying the type of updating. Currently supported is `"Ax"`, in which case the matrix `A` must be specified. See Examples.
`A`	a numeric matrix
`type`	string: either `"numeric"`, `"logical"` or `"permute"`
`length`	integer: the length of a vector
`...`	other arguments

Details

The function returns a closure with arguments `x` and `...`, which can be used for local-search algorithms.

Three types of solution vectors are supported:

- `numeric` a neighbour is created by adding or subtracting typically small numbers to random elements of a solution
- `logical` `TRUE` and `FALSE` values are switched
- `permute` elements of `x` are exchanged. Works with atomic and generic vectors (aka lists).

`neighborfun` is an alias for `neighbourfun`.

Value

A function (closure) with arguments `x` and `...`.

Note on algorithms

There are different strategies to implement constraints in local-search algorithms, and ultimately only experiments show which strategy works well for a given problem class. The algorithms used by `neighbourfun` always require a feasible initial solution, and then remain within the space of feasible solutions. See Gilli et al. (2019), Section 12.5, for a brief discussion.

Author(s)

Maintainer: Enrico Schumann <es@enricoschumann.net>

References

Gilli, M., Maringer, D. and Schumann, E. (2019) *Numerical Methods and Optimization in Finance*. 2nd edition. Elsevier.
doi:10.1016/C2017001621X

Schumann, E. (2025) *Financial Optimisation with R (***NMOF** *Manual).*
`https://enricoschumann.net/NMOF.htm#NMOFmanual`

See Also

implementations of algorithms of the local-search family, such as Simulated Annealing (`SAopt` in **NMOF**) or Threshold Accepting (`TAopt` in **NMOF**)

Examples

```
## a LOGICAL neighbourhood
x <- logical(8)
x[1:3] <- TRUE

N <- neighbourfun(type = "logical", kmin = 3, kmax = 3)

cat(ifelse(x, "o", "."), "  | initial solution ",
    sep = "", fill = TRUE)
for (i in 1:5) {
    x <- N(x)
    cat(ifelse(x, "o", "."), sep = "", fill = TRUE)
}
## ooo.....  | initial solution
## oo....o.
## o...o.o.
## o.o.o...
## oo..o...
## oo....o.




## UPDATING a numeric neighbourhood
##   the vector is 'x' is used in the product 'Ax'
A <- array(rnorm(100*25), dim = c(100, 25))
N <- neighbourfun(type = "numeric",
                  stepsize = 0.05,
                  update = "Ax",
                  A = A)
x <- rep(1/25, 25)
attr(x, "Ax") <- A %*% x
for (i in 1:10)
    x <- N(x, A)

all.equal(A %*% x, attr(x, "Ax"))




## a PERMUTATION neighbourhood
x <- 1:5

N <- neighbourfun(type = "permute")
N(x)
## [1] 1 2 5 4 3
##         ^   ^

N <- neighbourfun(type = "permute", stepsize = 5)
```

```
N(x)

## 'x' is not restricted to integers
x <- letters[1:5]
N(x)


## a useful way to STORE/SPECIFY PARAMETERS, e.g. in config files
settings <- list(type = "numeric",
                 min = 0.0,
                 max = 0.2)
do.call(neighbourfun, settings)
```

`next_subset` *Select Next Subset*

Description

Select next subset of size k from a set of size n.

Usage

```
next_subset(a, n, k)
```

Arguments

a	a numeric vector (integers)
n	an integer: the size of the set to choose from
k	an integer: the subset size

Details

Given a subset a of size k taken from a set of size n, compute the next subset by alphabetical order.

Uses algorithm NEXKSB of Nijenhuis and Wilf (1975).

Value

a numeric vector (the next subset) or `NULL` (when there is no next subset)

Author(s)

Enrico Schumann

References

Nijenhuis, A. and Wilf, H. S. (1975) *Combinatorial Algorithms for Computers and Calculators*. Academic Press.

See Also

`choose` computes the number of combinations
`combn` creates all combinations
`expand.grid`

Examples

```
n <- 4
k <- 2
t(combn(n, k))
##      [,1] [,2]
## [1,]    1    2
## [2,]    1    3
## [3,]    1    4
## [4,]    2    3
## [5,]    2    4
## [6,]    3    4

a <- 1:k
print(a)
while (!is.null(a))
    print(a <- next_subset(a, n = n, k = k))
## [1] 1 2
## [1] 1 3
## [1] 1 4
## [1] 2 3
## [1] 2 4
## [1] 3 4
```

Index