Package ‘CEGO’

February 19, 2015

Version 1.0.1108
Title Combinatorial Efficient Global Optimization
Author Martin Zaefferer <mzaefferer@gmail.com>
Maintainer Martin Zaefferer <mzaefferer@gmail.com>
Description Model building, surrogate model based optimization and Efficient Global Optimization in combinatorial or mixed search spaces.
License GPL (>= 3)
Type Package
LazyLoad yes
Depends R (>= 3.0.0)
Imports MASS, stats, DEoptim
Suggests nloptr
ByteCompile TRUE
Date 06.01.2015
Date/Publication 2015-01-06 17:35:45
NeedsCompilation yes
Repository CRAN

R topics documented:

CEGO-package .......................................................... 3
benchmarkGeneratorFSP ........................................... 3
benchmarkGeneratorNKL ............................................ 4
benchmarkGeneratorQAP ............................................ 5
benchmarkGeneratorTSP ............................................ 6
benchmarkGeneratorWT ............................................ 7
combinatorialKriging ............................................. 8
combinatorialLM .................................................. 10
combinatorialRBFN .............................................. 12
distancePermutationAdjacency ................................. 13
R topics documented:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>distancePermutationChebyshev</td>
<td>14</td>
</tr>
<tr>
<td>distancePermutationCos</td>
<td>15</td>
</tr>
<tr>
<td>distancePermutationEuclidean</td>
<td>16</td>
</tr>
<tr>
<td>distancePermutationHamming</td>
<td>17</td>
</tr>
<tr>
<td>distancePermutationInsert</td>
<td>17</td>
</tr>
<tr>
<td>distancePermutationInterchange</td>
<td>18</td>
</tr>
<tr>
<td>distancePermutationLCSeq</td>
<td>19</td>
</tr>
<tr>
<td>distancePermutationLCStr</td>
<td>20</td>
</tr>
<tr>
<td>distancePermutationLee</td>
<td>21</td>
</tr>
<tr>
<td>distancePermutationLevenshtein</td>
<td>22</td>
</tr>
<tr>
<td>distancePermutationLex</td>
<td>23</td>
</tr>
<tr>
<td>distancePermutationManhattan</td>
<td>24</td>
</tr>
<tr>
<td>distancePermutationPosition</td>
<td>25</td>
</tr>
<tr>
<td>distancePermutationPosition2</td>
<td>26</td>
</tr>
<tr>
<td>distancePermutationR</td>
<td>27</td>
</tr>
<tr>
<td>distancePermutationSwap</td>
<td>28</td>
</tr>
<tr>
<td>infillExpectedImprovement</td>
<td>29</td>
</tr>
<tr>
<td>landscapeGeneratorGaussian</td>
<td>29</td>
</tr>
<tr>
<td>landscapeGeneratorMUL</td>
<td>31</td>
</tr>
<tr>
<td>landscapeGeneratorUNI</td>
<td>32</td>
</tr>
<tr>
<td>lexicographicPermutationOrderNumber</td>
<td>33</td>
</tr>
<tr>
<td>longestIncreasingSubsequence</td>
<td>33</td>
</tr>
<tr>
<td>mutationBinary</td>
<td>34</td>
</tr>
<tr>
<td>mutationBinaryFast</td>
<td>35</td>
</tr>
<tr>
<td>mutationPermutationInterchange</td>
<td>35</td>
</tr>
<tr>
<td>mutationPermutationSwap</td>
<td>36</td>
</tr>
<tr>
<td>optim2Opt</td>
<td>36</td>
</tr>
<tr>
<td>optimCEGO</td>
<td>37</td>
</tr>
<tr>
<td>optimEA</td>
<td>39</td>
</tr>
<tr>
<td>optimInterface</td>
<td>41</td>
</tr>
<tr>
<td>optimRS</td>
<td>43</td>
</tr>
<tr>
<td>predict.CKriging</td>
<td>44</td>
</tr>
<tr>
<td>predict.CLM</td>
<td>45</td>
</tr>
<tr>
<td>predict.CRBFN</td>
<td>45</td>
</tr>
<tr>
<td>recombinationBinaryUniformCrossoverFast</td>
<td>46</td>
</tr>
<tr>
<td>recombinationPermutationCycleCrossover</td>
<td>46</td>
</tr>
<tr>
<td>solutionFunctionGeneratorBinary</td>
<td>47</td>
</tr>
<tr>
<td>solutionFunctionGeneratorPermutation</td>
<td>47</td>
</tr>
</tbody>
</table>

Index 49
benchmarkGeneratorFSP

Description

Combinatorial Efficient Global Optimization

Details

Model building, surrogate model based optimization and Efficient Global Optimization in combinatorial or mixed search spaces.

Package: CEGO
Type: Package
Version: 1.0.1108
Date: 06.01.2015
License: GPL (>= 3)
LazyLoad: yes

Author(s)

Martin Zaefferer <mzaeffeer@gmail.com>

References


See Also

Interface of main function: optimCEGO

benchmarkGeneratorFSP

Create Flow shop Scheduling Problem (FSP) Benchmark

Description

Creates a benchmark function for the Flow shop Scheduling Problem.
Usage

benchmarkGeneratorFSP(a, n, m)

Arguments

a matrix of processing times for each step and each machine
n number of jobs
m number of machines

Value

the function of type cost=f(permutation)

See Also

benchmarkGeneratorQAP, benchmarkGeneratorTSP, benchmarkGeneratorWT

Examples

n=10
m=4
# create a matrix of processing times
A <- matrix(sample(100,replace=TRUE),n,m)
# create FSP objective function
fun <- benchmarkGeneratorFSP(A,n,m)
# evaluate
fun(1:n)
fun(n:1)

benchmarkGeneratorNKL    NK-Landscape Benchmark Creation

Description

Function that generates a NK-Landscapes.

Usage

benchmarkGeneratorNKL(N = 10, K = 1, PI = 1:K, g = NULL)

Arguments

N length of the bit strings
K number of neighbours contributing to fitness of one position
PI vector, giving relative positions of each neighbour in the bit-string
g set of fitness functions for each possible combination of string components. Will be randomly determined if not specified. Should have N rows, and $2^K(K+1)$ columns.
Value

the function of type cost=f(bitstring). Returned fitness values will be negative, for purpose of minimization.

Examples

```r
fun <- benchmarkGeneratorQAP(6, 2)
fun(c(1, 0, 0, 1, 1, 0, 1))
fun(c(1, 0, 1, 1, 0, 0, 0))
fun(c(0, 1, 0, 0, 1, 1))
fun <- benchmarkGeneratorQAP(6, 3)
fun(c(1, 0, 1, 0, 0))
fun <- benchmarkGeneratorQAP(6, 2, c(-1, 1))
fun(c(1, 0, 1, 1, 0, 0))
fun <- benchmarkGeneratorQAP(6, 2, c(-1, 1), g=matrix(runif(48), 6))
fun(c(1, 0, 1, 1, 0, 0))
fun(sample(c(0, 1), 6, TRUE))
```

**Description**

Creates a benchmark function for the Quadratic Assignment Problem.

**Usage**

```r
benchmarkGeneratorQAP(a, b)
```

**Arguments**

- `a` : flow matrix
- `b` : distance matrix

**Value**

the function of type cost=f(permutation)

**See Also**

`benchmarkGeneratorFSP`, `benchmarkGeneratorTSP`, `benchmarkGeneratorWT`
Examples

```r
set.seed(1)
n=5
# create a flow matrix
A <- matrix(0,n,n)
for(i in 1:n){
  for(j in i:n){
    if(i==j){
      A[i,j] <- sample(100,1)
    }
  }
}
# create a distance matrix
locations <- matrix(runif(n*2)*10,2)
B <- as.matrix(dist(locations))
# create QAP objective function
fun <- benchmarkGeneratorQAP(A,B)
# evaluate
fun(1:n)
fun(n:1)
```

benchmarkGeneratorTSP  
Create (Asymmetric) Travelling Salesperson Problem (TSP) Benchmark

Description

Creates a benchmark function for the (Asymmetric) Travelling Salesperson Problem. Path (Do not return to start of tour. Start and end of tour not fixed.) or Cycle (Return to start of tour). Symmetry depends on supplied distance matrix.

Usage

```r
benchmarkGeneratorTSP(distanceMatrix, type = "Cycle")
```

Arguments

- `distanceMatrix`: Matrix that collects the distances between travelled locations.
- `type`: Can be "Cycle" (return to start, default) or "Path" (no return to start).

Value

The function of type \( \text{cost}=f(\text{permutation}) \)

See Also

`benchmarkGeneratorQAP`, `benchmarkGeneratorFSP`, `benchmarkGeneratorWT`
**benchmarkGeneratorWT**

Create single-machine total Weighted Tardiness (WT) Problem Benchmark

**Description**

Creates a benchmark function for the single-machine total Weighted Tardiness Problem.

**Usage**

`benchmarkGeneratorWT(p, w, d)`

**Arguments**

- `p` processing times
- `w` weights
- `d` due dates

**Value**

the function of type `cost=f(permutation)`

**See Also**

`benchmarkGeneratorQAP`, `benchmarkGeneratorTSP`, `benchmarkGeneratorFSP`
Examples

n=6
#processing times
p <- sample(100,n,replace=TRUE)
#weights
w <- sample(10,n,replace=TRUE)
#due dates
RDD <- c(0.2, 0.4, 0.6, 0.8, 1.0)
TF <- c(0.2, 0.4, 0.6, 0.8, 1.0)
i <- 1
j <- 1
P <- sum(p)
d <- runif(n,min=P*(1-TF[i]-RDD[j]/2),max=P*(1-TF[i]+RDD[j]/2))
#create WT objective function
fun <- benchmarkGeneratorWT(p,w,d)
fun(1:n)
fun(n:1)

combinatorialKriging  Combinatorial Kriging

Description

Implementation of a Kriging model for mixed or combinatorial input spaces. It is based on employing suitable distance measures for the samples in input space.

Usage

combinatorialKriging(x, y, distanceFunction, control = list())

Arguments

x list of samples in input space
y column vector of observations for each sample
distanceFunction
a suitable distance function of type f(x1,x2), returning a scalar distance value, preferably between 0 and 1. Maximum distances larger 1 are no problem, but may yield scaling bias when different measures are compared. Should be non-negative and symmetric. It can also be a list of several distance functions. In this case, Maximum Likelihood Estimation (MLE) is used to determine the most suited distance measure.

control (list), with the options for the model building procedure:
lower lower boundary for theta, default is 1e-6
upper upper boundary for theta, default is 100
corr function to be used for correlation modelling, default is fcorrGauss
algTheta algorithm used to find theta (as well as p and lambda), default is “L-BFGS-B”. Else, any from the list of possible method values in optimInterface
can be chosen.
budgetAlgLtheta budget for the above mentioned algorithm, default is 100. The value will be multiplied with the length of the model parameter vector to be optimized. optimizeP boolean that specifies whether the exponents (p) should be optimized. Else they will be set to two.
useLlambda whether or not to use the regularization constant lambda (nugget effect). Default is FALSE. lambdaLower lower boundary for lambda, default is -6 lambdaUpper upper boundary for lambda. default is 0 distances a distance matrix. If available, this matrix is used for model building, instead of calculating the distance matrix using the parameters distanceFunction. Default is NULL.

Details

The basic Kriging implementation is based on the work of Forrester et al. (2008). For adaptation of Kriging to mixed or combinatorial spaces, as well as choosing distance measures with Maximum Likelihood Estimation, see the other two references (Zaefferer et al., 2014).

Value

an object of class CKriging containing the options and determined parameters for the model:
x (see parameters)
y (see parameters)
lower (see parameters)
upper (see parameters)
algLtheta (see parameters)
budgetAlgLtheta (see parameters)
optimizeP (see parameters)
theta activity or width parameter theta, a parameter of the correlation function determined with MLE
log10Theta log10 theta (i.e. log10(theta))
lambda regularization constant (nugget) lambda
log10Lambda log10 of regularization constant (nugget) lambda (i.e. log10(lambda))
p exponent p, parameter of the correlation function determined with MLE (if optimizeP is TRUE)
yMu vector of observations y, minus MLE of mu
SSQ Maximum Likelihood Estimate (MLE) of model parameter sigma^2
mu MLE of model parameter mu
Psi correlation matrix Psi
Psinv inverse of Psi
nevals number of Likelihood evaluations during MLE of theta/lambda/p
distanceFunctionIndexMLE If a list of several distance measures (distanceFunction) was given, this parameter contains the index value of the measure chosen with MLE.

References

Zaefferer, Martin; Stork, Joerg; Friese, Martina; Fischbach, Andreas; Naujoks, Boris; Bartz-Beielstein, Thomas. (2014). Efficient global optimization for combinatorial problems. In Proceedings of the
combinatorialLM

Distance based Linear Model

Description

A simple linear model based on arbitrary distances. Comparable to a nearest neighbor, but potentially able to extrapolate into regions of improvement. Used as a simple baseline by Zaefferer et al. (2014). Basically just generates a proper CLM object for the prediction function.
combinatorialLM

Usage

combinatorialLM(x, y, distanceFunction, control = list())

Arguments

x  list of samples in input space
y  matrix, vector of observations for each sample
distanceFunction  a suitable distance function of type f(x1,x2), returning a scalar distance value, preferably between 0 and 1. Maximum distances larger 1 are no problem, but may yield scaling bias when different measures are compared. Should be non-negative and symmetric.
control  currently unused, defaults to list()

Value

a fit (list, CLM), with the options and found parameters for the model which has to be passed to the predictor function:
x  samples in input space (see parameters)
y  observations for each sample (see parameters)
distanceFunction  distance function (see parameters)

References


See Also

predict.CLM

Examples

# set random number generator seed
set.seed(1)
# simple test landscape
fn <- landscapeGeneratorUNI(1:5,distancePermutationHamming)
# generate data for training and test
x <- unique(replicate(40,sample(5),FALSE))
xtest <- x[-(1:15)]
x <- x[1:15]
# determine true objective function values
y <- sapply(x,fn)
ytest <- sapply(xtest,fn)
# build model
fit <- combinatorialLM(x,y,distancePermutationHamming)
# predicted obj. function values
combinatorialRBFN

Description

Implementation of a Radial Basis Function Network (RBFN) model for mixed or combinatorial input spaces. It is based on employing suitable distance measures for the samples in input space. For reference, see the paper by Moraglio and Kattan (2011).

Usage

combinatorialRBFN(x, y, distanceFunction, control = list())

Arguments

x list of samples in input space
y column vector of observations for each sample
distanceFunction a suitable distance function of type f(x1,x2), returning a scalar distance value, preferably between 0 and 1. Maximum distances larger 1 are no problem, but may yield scaling bias when different measures are compared. Should be non-negative and symmetric.
control (list), with the options for the model building procedure:
beta Parameter of the radial basis function: exp(-beta*D), where D is the distance matrix. If beta is not specified, the heuristic in fbeta will be used to determine it, which is default behavior.
fbeta Function f(x) to calculate the beta parameter, x is the maximum distance observed in the input data. Default function is $1/(2*(x^2))$.
distances a distance matrix. If available, this matrix is used for model building, instead of calculating the distance matrix using the parameters distanceFunction. Default is NULL.

Value

a fit (list, CRBFN), with the options and found parameters for the model which has to be passed to the predictor function:
SSQ Variance of the observations (y)
centers Centers of the RBFN model, samples in input space (see parameters)
w Model parameters (weights) w
Phi Gram matrix
Phinv (Pseudo)-Inverse of Gram matrix
**distancePermutationAdjacency**

w0 Mean of observations (y)
dMax Maximum observed distance
d Matrix of distances between all samples
beta See parameters
fbeta See parameters
distanceFunction See parameters

**References**


**See Also**

predict.CRBFN

**Examples**

```r
#set random number generator seed
set.seed(1)
#simple test landscape
fn <- landscapeGeneratorUNI(1:5,distancePermutationHamming)
#generate data for training and test
x <- unique(replicate(40,sample(5),FALSE))
xtest <- x[-(1:15)]
x <- x[1:15]
#determin true objective function values
y <- sapply(x,fn)
ytest <- sapply(xtest,fn)
#build model
fit <- combinatorialRBFN(x,y,distancePermutationHamming)
#predicted obj. function values
ypred <- predict(fit,xtest)$f
#uncertainty estimate
spred <- predict(fit,xtest,TRUE)$s
#plot
plot(ytest,ypred,xlab="true value",ylab="predicted value",
     pch=20,xlim=c(0.3,1),ylim=c(min(ypred)-0.1,max(ypred)+0.1))
abline(0,1,lt=2)
```

**Description**

Adjacency Distance for Permutations

Bi-directional adjacency distance for permutations, depending on how often two elements are neighbours in both permutations x and y.
Usage

distancePermutationAdjacency(x, y)

Arguments

x first permutation (integer vector)
y second permutation (integer vector)

Value

numeric distance value

\[ d(x, y) \]

, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

References


Examples

x <- 1:5
y <- 5:1
distancePermutationAdjacency(x,y)
p <- replicate(10,sample(1:5),simplify=FALSE)
distanceMatrix(p,distancePermutationAdjacency)

---

distancePermutationChebyshev

Chebyshev Distance for Permutations

Description

Chebyshev distance for permutations. Specific to permutations is only the scaling to values of 0 to 1:

\[ d(x, y) = \frac{\max(|x - y|)}{(n - 1)} \]

where n is the length of the permutations x and y.

Usage

distancePermutationChebyshev(x, y)
Arguments

- `x` first permutation (integer vector)
- `y` second permutation (integer vector)

Value

numeric distance value

\[ d(x, y) \]

, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

Examples

```r
x <- 1:5
y <- c(5,1,2,3,4)
distancePermutationChebyshev(x, y)
p <- replicate(10, sample(1:5), simplify = FALSE)
distanceMatrix(p, distancePermutationChebyshev)
```

---

**Cosine Distance for Permutations**

Description

The Cosine distance for permutations is derived from the Cosine similarity measure which has been applied in fields like text mining. It is based on the scalar product of two vectors (here: permutations).

Usage

```
distancePermutationCos(x, y)
```

Arguments

- `x` first permutation (integer vector)
- `y` second permutation (integer vector)

Value

numeric distance value

\[ d(x, y) \]

, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)
distancePermutationEuclidean

Euclidean Distance for Permutations

Description
Euclidean distance for permutations, scaled to values between 0 and 1:

\[ d(x, y) = \frac{1}{r \sqrt{n \sum_{i=1}^{n} (x_i - y_i)^2}} \]

where \( n \) is the length of the permutations \( x \) and \( y \), and scaling factor \( r = \sqrt{2 \times 4 \times n \times (n + 1) \times (2 \times n + 1) / 6} \) (if \( n \) is odd) or \( r = \sqrt{2 \times 4 \times n \times (2 \times n - 1) \times (2 \times n + 1) / 3} \) (if \( n \) is even).

Usage
distancePermutationEuclidean(x, y)

Arguments
- x: first permutation (integer vector)
- y: second permutation (integer vector)

Value
numeric distance value \( d(x, y) \), scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

Examples
x <- 1:5
y <- c(5,1,2,3,4)
distancePermutationEuclidean(x, y)
p <- replicate(10, sample(1:5), simplify=FALSE)
distanceMatrix(p, distancePermutationEuclidean)

References
**distancePermutationHamming**

*Hamming Distance for Permutations*

**Description**

Hamming distance for permutations, scaled to values between 0 and 1. That is, the number of unequal elements of two permutations, divided by the permutations length.

**Usage**

```
distancePermutationHamming(x, y)
```

**Arguments**

- `x` first permutation (integer vector)
- `y` second permutation (integer vector)

**Value**

numeric distance value 

\[ d(x, y) \]

, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

**Examples**

```
x <- 1:5
y <- c(5,1,2,3,4)
distancePermutationHamming(x,y)
p <- replicate(10,sample(1:5),simplify=FALSE)
distanceMatrix(p,distancePermutationHamming)
```

---

**distancePermutationInsert**

*Insert Distance for Permutations*

**Description**

The Insert Distance is an edit distance. It counts the minimum number of insert operations required to transform one permutation into another. An insert operation shifts one element to a new position. All other elements move accordingly to make place for the element. E.g., the following shows a single insert move that sorts the corresponding permutation: 1 4 2 3 5 \(\rightarrow\) 1 2 3 4 5. This distance is also called Ulam’s metric and is the same as the distance measure based on the longest common subsequence (*distancePermutationLCSeq*). Note, that the implementations differ. Insert distance scales better for longer permutations, whereas LCSeq works better for shorter permutations.
Usage

distancePermutationInsert(x, y)

Arguments

x
first permutation (integer vector)

y
second permutation (integer vector)

Value

numeric distance value

\[ d(x, y) \]

, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

References


Examples

```r
x <- 1:5
y <- c(5,1,2,3,4)
distancePermutationInsert(x,y)
p <- replicate(10,sample(1:5),simplify=FALSE)
distanceMatrix(p,distancePermutationInsert)
```

---

**Interchange Distance for Permutations**

Description

The interchange distance is an edit-distance, counting how many edit operation (here: interchanges, i.e., transposition of two arbitrary elements) have to be performed to transform permutation x into permutation y.

Usage

```
distancePermutationInterchange(x, y)
```
**distancePermutationLCSeq**

**Arguments**

- **x**: first permutation (integer vector)
- **y**: second permutation (integer vector)

**Value**

A numeric distance value, denoted as $d(x, y)$, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations).

**References**


**Examples**

```r
x <- 1:5
y <- c(1, 4, 3, 2, 5)
distancePermutationInterchange(x, y)
p <- replicate(10, sample(1:5), simplify=FALSE)
distanceMatrix(p, distancePermutationInterchange)
```

---

**distancePermutationLCSeq**

*Longest Common Subsequence Distance for Permutations*

**Description**

Distance of permutations. Based on the longest sequence (breaks allowed) of permutation elements in increasing order that both permutations have in common. This distance is also called Ulam’s metric and is the same as the distance measure based on the insert edit-operation (**distancePermutationInsert**). Note, that the implementations differ. Insert distance scales better for longer permutations, whereas LCSeq is faster for shorter permutations.

**Usage**

```r
distancePermutationLCSeq(x, y)
```

**Arguments**

- **x**: first permutation (integer vector)
- **y**: second permutation (integer vector)
**distancePermutationLCStr**

**Description**

Distance of permutations. Based on the longest string of adjacent elements that two permutations have in common.

**Usage**

```r
distancePermutationLCStr(x, y)
```

**Arguments**

- `x`: first permutation (integer vector)
- `y`: second permutation (integer vector)

**Value**

- numeric distance value $d(x, y)$, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

**Examples**

```r
x <- 1:5
y <- c(1, 2, 5, 3, 4)
distancePermutationLCSeq(x, y)
p <- replicate(10, sample(1:5), simplify=FALSE)
distanceMatrix(p, distancePermutationLCSeq)
```

**References**

distancePermutationLee

References


Examples

```r
x <- 1:5
y <- c(5,1,2,3,4)
distancePermutationLCStr(x, y)
p <- replicate(10, sample(1:5), simplify=FALSE)
distanceMatrix(p, distancePermutationLCStr)
```

---

**Lee Distance for Permutations**

**Description**

Usually a string distance, with slightly different definition. Adapted to permutations as:

\[
d(x, y) = \sum_{i=1}^{n} \min(|x_i - y_i|, n - |x_i - y_i|)
\]

where \(n\) is the length of the permutations \(x\) and \(y\).

**Usage**

```r
distancePermutationLee(x, y)
```

**Arguments**

- \(x\) first permutation (integer vector)
- \(y\) second permutation (integer vector)

**Value**

numeric distance value \(d(x, y)\), scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

**References**

distancePermutationLevenshtein

Levenshtein Distance for Permutations

Description

Levenshtein Distance, often just called "Edit Distance". The number of insertions, substitutions or deletions to turn one permutation (or string of equal length) into another.

Usage

distancePermutationLevenshtein(x, y)

Arguments

x          first permutation (integer vector)
y          second permutation (integer vector)

Value

numeric distance value

\[ d(x, y) \]

, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

References


Examples

x <- 1:5  
y <- c(5,1,2,3,4)  
distancePermutationLevenshtein(x, y)  
p <- replicate(10, sample(1:5), simplify = FALSE)  
distanceMatrix(p, distancePermutationLevenshtein)
**distancePermutationLex**

*Lexicographic permutation distance*

**Description**

This function calculates the lexicographic permutation distance. That is the difference of positions that both positions would receive in a lexicographic ordering. Note, that this distance measure can quickly become inaccurate if the length of the permutations grows too large, due to being based on the factorial of the length. In general, permutations longer than 100 elements should be avoided.

**Usage**

\[
\text{distancePermutationLex}(x, y)
\]

**Arguments**

- \(x\) first permutation (integer vector)
- \(y\) second permutation (integer vector)

**Value**

numeric distance value

\[
d(x, y)
\]

, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

**See Also**

lexicographicPermutationOrderNumber

**Examples**

\[
x <- 1:5
y <- c(1,2,3,5,4)
distancePermutationLex(x, y)
p <- replicate(10, sample(1:5), simplify=FALSE)
distanceMatrix(p, distancePermutationLex)
\]
Manhattan Distance for Permutations

Description

Manhattan distance for permutations, scaled to values between 0 and 1:

\[ d(x, y) = \frac{1}{r} \sum_{i=1}^{n} |x_i - y_i| \]

where \( n \) is the length of the permutations \( x \) and \( y \), and scaling factor \( r = \frac{n^2 - 1}{2} \) (if \( n \) is odd) or \( r = \frac{(n^2)/2}{2} \) (if \( n \) is even).

Usage

\texttt{distancePermutationManhattan(x, y)}

Arguments

- \( x \) first permutation (integer vector)
- \( y \) second permutation (integer vector)

Value

numeric distance value

\( d(x, y) \)

, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

Examples

\begin{verbatim}
x <- 1:5
y <- c(5,1,2,3,4)
distancePermutationManhattan(x,y)
p <- replicate(10,sample(1:5),simplify=FALSE)
distanceMatrix(p,distancePermutationManhattan)
\end{verbatim}
**distancePermutationPosition**

*Position Distance for Permutations*

**Description**

Position distance (or Spearman's Correlation Coefficient), scaled to values between 0 and 1.

**Usage**

```r
distancePermutationPosition(x, y)
```

**Arguments**

- `x` first permutation (integer vector)
- `y` second permutation (integer vector)

**Value**

numeric distance value

\[ d(x, y) \]

, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

**References**


**Examples**

```r
x <- 1:5
y <- c(1, 3, 5, 4, 2)
distancePermutationPosition(x, y)
p <- replicate(10, sample(1:5, simplify=FALSE))
distanceMatrix(p, distancePermutationPosition)
```
distancePermutationPosition2

Squared Position Distance for Permutations

Description

Squared position distance (or Spearman’s Footrule), scaled to values between 0 and 1.

Usage

distancePermutationPosition2(x, y)

Arguments

- x: first permutation (integer vector)
- y: second permutation (integer vector)

Value

numeric distance value

\[ d(x, y) \]

, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

References


Examples

```r
x <- 1:5
y <- c(1, 3, 5, 4, 2)
distancePermutationPosition2(x, y)
p <- replicate(10, sample(1:5), simplify=FALSE)
distanceMatrix(p, distancePermutationPosition2)
```
**distancePermutationR**  \hspace{1cm} **R-Distance for Permutations**

---

**Description**

R distance or unidirectional adjacency distance. Based on count of number of times that a two element sequence in x also occurs in y, in the same order.

**Usage**

```r
distancePermutationR(x, y)
```

**Arguments**

- **x**
  - first permutation (integer vector)
- **y**
  - second permutation (integer vector)

**Value**

numeric distance value

\[ d(x, y) \]

, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

**References**


**Examples**

```r
x <- 1:5
y <- c(1,2,3,5,4)
distancePermutationR(x, y)
p <- replicate(10, sample(1:5), simplify=FALSE)
distanceMatrix(p, distancePermutationR)
```
**distancePermutationSwap**

*Swap-Distance for Permutations*

### Description

The swap distance is an edit-distance, counting how many edit operation (here: swaps, i.e., transposition of two adjacent elements) have to be performed to transform permutation \( x \) into permutation \( y \).

### Usage

\[
distancePermutationSwap(x, y)
\]

### Arguments

- **x**: first permutation (integer vector)
- **y**: second permutation (integer vector)

### Value

numeric distance value

\[
d(x, y)
\]

, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

### References


### Examples

```r
x <- 1:5
y <- c(1,2,3,5,4)
distancePermutationSwap(x,y)
p <- replicate(10,sample(1:5),simplify=FALSE)
distanceMatrix(p,distancePermutationSwap)
```
infillExpectedImprovement

Negative Logarithm of Expected Improvement

Description

This function calculates the Expected Improvement of candidate solutions, based on predicted means, standard deviations (uncertainty) and the best known objective function value so far.

Usage

infillExpectedImprovement(mean, sd, min)

Arguments

- mean: predicted mean values
- sd: predicted standard deviation
- min: minimum of all observations so far

Value

Returns the negative logarithm of the Expected Improvement.

landscapeGeneratorGaussian

Create Gaussian Landscape

Description

This function is loosely based on the Gaussian Landscape Generator by Bo Yuan and Marcus Gallagher. It creates a Gaussian Landscape every time it is called. This Landscape can be evaluated like a function. To adapt to combinatorial spaces, the Gaussians are here based on a user-specified distance measure. Due to the expected nature of combinatorial spaces and their lack of direction, the resulting Gaussians are much simplified in comparison to the continuous, vector-valued case (e.g., no rotation). Since the CEGO package is tailored to minimization, the landscape is inverted.

Usage

landscapeGeneratorGaussian(nGaussian = 10, upper = 1, ratio = 0.2, seed = 1, distanceFunction, creationFunction)
Arguments

- `nGaussian`: number of Gaussian components in the landscape. Default is 10.
- `upper`: upper boundary of the `distanceFunction`. Controls width of Gaussian components. Default is 1.
- `ratio`: minimal function value of the local minima. Default is 0.2. (Note: Global minimum will be at zero, local minimal will be in range `[ratio, 1]`)
- `seed`: seed for the random number generator used before creation of the landscape. Generator status will be saved and reset afterwards.

`distanceFunction`:
A function of type `f(x, y)`, to evaluate distance between two samples in their given representation.

`creationFunction`:
function to randomly generate the centers of the Gaussians, in form of their given representation.

Value
returns a function. The function takes a solution sample (e.g., a permutation, a tree) as input. The function returns a single scalar value, which is the objective value for the given sample.

References

Examples
```r
# rng seed
seed=101
# distance function
dF <- function(x,y)(sum((x-y)^2)) # sum of squares
# df <- function(x,y)sqrt(sum((x-y)^2)) # euclidean distance
# creation function
cF <- function()runif(1)
# plot pars
par(mfrow=c(3,1),mar=c(3.5,3.5,0.2,0.2),mgp=c(2,1,0))
## uni modal distance landscape
# set seed
set.seed(seed)
# landscape
lf <- Vectorize(landscapeGeneratorUNI(cF(),dF))
x <- seq(from=-0,by=0.001,to=1)
plot(x,lf(x),type="l")
## multi-modal distance landscape
# set seed
set.seed(seed)
# landscape
lf <- Vectorize(landscapeGeneratorMUL(replicate(5,cF(),FALSE),dF))
```
landscapeGeneratorMUL

Description

This function generates multi-modal fitness landscapes based on distance measures. The fitness is the minimal distance to several reference individuals. Hence, each reference individual is an optimum of the landscape.

Usage

landscapeGeneratorMUL(ref = list(1:10), distFun)

Arguments

ref list of reference individuals

distfun Distance function, used to evaluate d(x,ref[[n]]), where x is an arbitrary new individual

Value

returns a function. The function can take any type of individual as its input, depending on whether it is suitable for the distance function.

See Also

landscapeGeneratorUNI, landscapeGeneratorGaussian

fun <- landscapeGeneratorMUL(ref=list(1:7,c(2,4,1,5,3,7,6)),distancePermutationCos) x <- 1:7 fun(x) x <- c(2,4,1,5,3,7,6) fun(x) x <- 7:1 fun(x) x <- sample(7) fun(x)
Description

This function generates uni-modal fitness landscapes based on distance measures. The fitness is the
distance to a reference individual. Hence, the reference individual is the optimum of the landscape.

Usage

landscapeGeneratorUNI(ref = 1:10, distFun)

Arguments

ref Reference individual
distFun Distance function, used to evaluate d(x,ref), where x is an arbitrary new individ-
ual

Value

returns a function. The function can take any type of individual as its input, depending on whether
it is suitable for the distance function.

References

Moraglio, Alberto, Yong-Hyuk Kim, and Yourim Yoon. "Geometric surrogate-based optimisation
for permutation-based problems." Proceedings of the 13th annual conference companion on Genetic
and evolutionary computation. ACM, 2011.

See Also

landscapeGeneratorMUL, landscapeGeneratorGaussian

Examples

```r
fun <- landscapeGeneratorUNI(ref=1:7,distancePermutationCos)
x <- 1:7
fun(x)
x <- 7:1
fun(x)
x <- sample(7)
fun(x)
```
**Lexicographic Order Number**

*Lexicographic order number*

**Description**

This function returns the position-number that a permutation would receive in a lexicographic ordering. It is used in the lexicographic distance measure.

**Usage**

```r
lexicographicPermutationOrderNumber(x)
```

**Arguments**

- `x`  permutation (integer vector)

**Value**

numeric value giving position in lexicographic order.

**See Also**

`distancePermutationLex`

**Examples**

```r
lexicographicPermutationOrderNumber(1:5)
lexicographicPermutationOrderNumber(c(1,2,3,5,4))
lexicographicPermutationOrderNumber(c(1,2,4,3,5))
lexicographicPermutationOrderNumber(c(1,2,4,5,3))
lexicographicPermutationOrderNumber(c(1,2,5,3,4))
lexicographicPermutationOrderNumber(c(1,2,5,4,3))
lexicographicPermutationOrderNumber(c(1,3,2,4,5))
lexicographicPermutationOrderNumber(5:1)
lexicographicPermutationOrderNumber(1:7)
lexicographicPermutationOrderNumber(7:1)
```

---

**Longest Increasing Subsequence**

*Longest Increasing Subsequence*

**Description**

Calculate the longest increasing subsequence (LIS) in a vector (e.g., a permutation).
Usage

longestIncreasingSubsequence(x)

Arguments

x integer vector

References


Examples

longestIncreasingSubsequence(1:5)
longestIncreasingSubsequence(5:1)
longestIncreasingSubsequence(c(1,2,4,3,5))
longestIncreasingSubsequence(c(1,2,4,3,5,1,2,4,3,2,9))

---

**mutationBinary**

*Bit-flip Mutation for Bit-strings*

Description

Given a population of bit-strings, this function mutates all individuals by randomly inverting one or more bits in each individual.

Usage

mutationBinary(population, parameters)

Arguments

<table>
<thead>
<tr>
<th>argument</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>population</td>
<td>List of bit-strings</td>
</tr>
<tr>
<td>parameters</td>
<td>list of parameters: parameters$mutationRate =&gt; mutation rate, specifying number of bits flipped. Should be in range between zero and one</td>
</tr>
</tbody>
</table>

Value

mutated population
**mutationBinaryFast**

*Bit-flip Mutation for Bit-strings (Fast)*

**Description**

Given a population of bit-strings, this function mutates all individuals by randomly inverting one bit in each individual. Due to the fixed mutation rate, this is computationally faster.

**Usage**

```
mutationBinaryFast(population, parameters)
```

**Arguments**

- **population**: List of bit-strings
- **parameters**: not used

**Value**

mutated population

**mutationPermutationInterchange**

*Interchange Mutation for Permutations*

**Description**

Given a population of permutations, this function mutates all individuals by randomly interchanging two arbitrary elements of the permutation.

**Usage**

```
mutationPermutationInterchange(population, parameters)
```

**Arguments**

- **population**: List of permutations
- **parameters**: not used.

**Value**

mutated population
**mutationPermutationSwap**

*Swap Mutation for Permutations*

**Description**

Given a population of permutations, this function mutates all individuals by randomly interchanging two adjacent elements of the permutation.

**Usage**

```r
mutationPermutationSwap(population, parameters)
```

**Arguments**

- `population`: List of permutations
- `parameters`: not used

**Value**

mutated population

---

**optim2Opt**

*Two-Opt*

**Description**

Implementation of a Two-Opt local search.

**Usage**

```r
optim2Opt(par = NULL, fun, creationFunction, control = list())
```

**Arguments**

- `par`: start solution of the local search
- `fun`: function that determines cost or length of a route/permutation
- `creationFunction`: Function to create individuals/solutions in search space
- `control`: (list), with the options
  - `budget`: The limit on number of target function evaluations (stopping criterion) (default: 100)
  - `vectorized`: Boolean. Defines whether target function is vectorized (takes a list of solutions as argument) or not (takes single solution as argument). Default: FALSE
Value

a list:
xbest best solution found
ybest fitness of the best solution
count number of performed target function evaluations

References


Examples

```r
seed=0
glgseed=1
#distance
dF <- distancePermutationHamming
#mutation
mF <- mutationPermutationSwap
#recombination
rF <- recombinationPermutationCycleCrossover
#creation
cF <- function(sample(5)
#objective function
LF <- landscapeGeneratorUNI(ref=1:5,distFun=dF)
#start optimization
set.seed(seed)
res <- optim2Opt(LF,cF,list(budget=100))
res
```

optimCEGO Combinatorial Efficient Global Optimization

Description

Model-based optimization for combinatorial or mixed problems. Based on measures of distance or dissimilarity.

Usage

```r
optimCEGO(par = NULL, fun, creationFunction, distanceFunction, control = list())
```

Arguments

par Optional initial design as a list. If NULL (default), creationFunction is used to create initial design. If par has less individuals than specified by control$evalInit, creationFunction will fill up the design.
fun  target function to be minimized
creationFunction  Function to create individuals/solutions in search space
distanceFunction  a suitable distance function of type f(x1,x2), returning a scalar distance value,
  preferably between 0 and 1. Maximum distances larger 1 are no problem, but
  may yield scaling bias when different measures are compared. Should be non-
  negative and symmetric. With the setting control$model="K" this can also be
  a list of different fitness functions.
control  (list), with the options of optimization and model building approaches employed
  evalInit Number of initial evaluations (i.e., size of the initial design), integer,
    default is 2
  vectorized Boolean. Defines whether target function is vectorized (takes a list
    of solutions as argument) or not (takes single solution as argument). Default: FALSE
  verbosity Level of text output during run. Defaults to 0, no output.
  plotting Plot optimization progress during run (TRUE) or not (FALSE). De-
    fault is FALSE.
  targetY optimal value to be found, stopping criterion, default is \( -\infty \)
  evalBudget maximum number of target function evaluations, default is 100
  creationRetries When a model does not predict an actually improving solu-
    tion, a random exploration step is performed. creationRetries solutions are
    created randomly. For each, distance to all known solutions is calculated. The
    minimum distance is recorded for each random solution. The random solution
    with maximal minimum distance is chosen doe be evaluated in the next itera-
    tion.
  model Model to be used as a surrogate of the target function. Default is "K"
    (Kriging). Also available are: "LM" (linear, distance-based model), "RBFN"
    Radial Basis Function Network.
  modelSettings List of settings for model building, passed on as the control ar-
    gument to the model training functions combinatorialKriging, combinatorialLM,
    combinatorialRBFN. infill This parameter specifies a function to be used for
    the infill criterion (e.g., the default is expected improvement infillExpectedImprovement).
    To use no specific infill criterion this has to be set to NA. Infill criteria are only
    used with models that may provide some error estimate with predictions.
  optimizer Optimizer that finds the minimum of the surrogate model. De-
    fault is "EA" an Evolutionary Algorithm. No alternatives implemented yet.
  optimSettings List of settings for the method to optimize the model.

Value

  a list:
    xbest best solution found
    ybest fitness of the best solution
    x history of all evaluated solutions
    y corresponding target function values f(x)
    fit model-fit created in the last iteration
References


See Also

combinatorialkriging, combinatoriallm, combinatorialrbfn, buildmodel, optimEA

Examples

```r
seed=0
glgseed=1
#distance
dF <- distancePermutationHamming
#mutation
mF <- mutationPermutationSwap
#recombination
rF <- recombinationPermutationCycleCrossover
#creation
cF <- function(){sample(5)}
#objective function
1F <- landscapeGeneratorUNI(ref=1:5,distFun=dF)
#start optimization
set.seed(seed)
res1 <- optimCEGO(1F,cF,dF,list(optimSettings=list(
  mutationFunction=mF,recombinationFunction=rF),
evalInit=5,budget=15,targetY=0,verbosity=1,model="K"))
set.seed(seed)
res2 <- optimCEGO(1F,cF,dF,list(optimSettings=list(
  mutationFunction=mF,recombinationFunction=rF),
evalInit=5,budget=15,targetY=0,verbosity=1,model="RBFN"))
res1$xbest
res2$xbest
```

Description

A basic implementation of a simple Evolutionary Algorithm for Combinatorial Optimization. Default evolutionary operators aim at permutation optimization problems.
Usage

optimEA(par = NULL, fun, creationFunction, control = list())

Arguments

par          Optional start individual(s) as a list. If NULL (default), creationFunction is used to create initial population. If par has less individuals than the population size, creationFunction will fill up the rest.

fun          target function to be minimized

creationFunction

control

(list), with the options

budget       The limit on number of target function evaluations (stopping criterion) (default: 1000)
popsizesize  Population size (default: 100)
generations  Number of generations (stopping criterion) (default: Inf)
targetY      Target function value (stopping criterion) (default: -Inf)
vectorized   Boolean. Defines whether target function is vectorized (takes a list of solutions as argument) or not (takes single solution as argument). Default: FALSE

verbosity    Level of text output during run. Defaults to 0, no output. plotting       Plot optimization progress during run (TRUE) or not (FALSE). Default is FALSE.

recombinationFunction Function that performs recombination, default: recombinationPermutationCrossover, which is cycle crossover for permutations.

recombinationParameters Parameter list for recombination (e.g., recombinationParameters$recrate => recombination rate, defaults to 0.5). List is passed to recombinationFunction.

mutationFunction Function that performs mutation, default: mutationPermutationSwap, which is swap mutation for permutations.

mutationParameters Parameter list for mutation (e.g., mutationParameters$mutationRate => mutation rate, defaults to 0.5). List is passed to mutationFunction.

selection Selection process: "tournament" (default) or "truncation"
tournamentSize Tournament size (default: 2)
tournamentProbability Tournament probability (default: 0.9)

localSearchFunction If specified, this function is used for a local search step. Default is NULL.

localSearchRate Specifies on what fraction of the population local search is applied. Default is zero. Maximum is 1 (100 percent).

localSearchSettings List of settings passed to the local search function control parameter. stoppingCriterionFunction

Value

a list:

xbest best solution found
ybest fitness of the best solution
x history of all evaluated solutions
y corresponding target function values f(x)
count number of performed target function evaluations message Termination message: Which stopping criterion was reached. population Last population fitness Fitness of last population

See Also
optimCEGO, optimRS, optim2Opt

Examples

```r
seed=0
glgseed=1
#distance
dF <- distancePermutationHamming
#mutation
mF <- mutationPermutationSwap
#recombination
rF <- recombinationPermutationCycleCrossover
#creation
cF <- function()sample(5)
#objective function
lf <- landscapeGeneratorUNI(ref=1:5,distFun=dF)
#start optimization
set.seed(seed)
res <- optimEA(lF,cF,list(mutationFunction=mF,recombinationFunction=rF,
                        popsize=15,budget=100,targetY=0,verbosity=1))
res$x$best
```

Description

This function is an interface fashioned like the optim function. Unlike optim, it collects a set of bound-constrained optimization algorithms with local as well as global approaches. It is used in the CEGO package to solve the optimization problem that occurs during parameter estimation in the Kriging model (based on Maximum Likelihood Estimation). Note that this function is NOT for combinatorial optimization problems.

Usage

```r
optimInterface(par, fn, lower = -Inf, upper = Inf,
               method = "optim-L-BFGS-B", control = list(), ...)
```
Arguments

par is a point (vector) in the decision space of fn
fn is the target function of type \( y = f(x, \ldots) \)
lower is a vector that defines the lower boundary of search space
upper is a vector that defines the upper boundary of search space
method is a string that describes which method is to be used, as implemented in this function. Else it can be a function, which is a custom optimization function created by the user. See details.
control is a list of additional settings. See details.

Details

The control list contains:
- fevals stopping criterion, number of evaluations allowed for fn (defaults to 100)
- reltol stopping criterion, relative tolerance (default: 1e-6)
- popsize population size or number of particles (default: \( 10 \times \text{dimension} \), where \text{dimension} is derived from the length of the vector \( \text{lower} \)).
- restarts the number of restarts to perform (Default: 0). Function evaluation budget will be split accordingly. Early convergence of optimization runs may lead to additional restarts. Violations of the provided budget may decrease the number of restarts.

Also note that the parameter method will be used to choose the optimization method from the following list:
- "L-BFGS-B" - BFGS quasi-Newton: stats Package optim function
- "nlminb" - box-constrained optimization using PORT routines: stats Package nlminb function
- "DEoptim" - Differential Evolution implementation: DEoptim Package

Additionally to the above methods, several methods from the package nloptr can be chosen. The complete list of suitable nlopt methods (non-gradient, bound constraints) is:
- "NLOPT_GN_DIRECT" , "NLOPT_GN_DIRECT_L" , "NLOPT_GN_DIRECT_L_RAND" , "NLOPT_GN_DIRECT_NOSCAL" , "NLOPT_GN_DIRECT_L_NOSCAL" , "NLOPT_GN_DIRECT_L_RAND_NOSCAL",
- "NLOPT_LN_MOO" , "NLOPT_LN_NELDERMead" , "NLOPT_LN_SBPLX" , "NLOPT_LN_BOBYQA" , "NLOPT_GN_ISRES"

All of the above methods use bound constraints. For references and details on the specific methods, please check the documentation of the packages that provide them.

Furthermore, the user can choose to use a function instead of a string for the method. The used function should have the same parameters and arguments as documented for this very function, i.e. optimInterface. The method parameters for the call to the supplied function will be extracted from control$method.

Value

This function returns a list with:
- par parameters of the found solution
- value target function value of the found solution
- counts number of evaluations of fn
Description

Random Search for mixed or combinatorial optimization.

Usage

```r
optimRS(par = NULL, fun, creationFunction, control = list())
```

Arguments

- `par` Optional set of solution(s) as a list. If NULL (default), creationFunction is used to create all solutions randomly. If par has less individuals than the population size, creationFunction will fill up the rest. par basically defines additional user specified solutions to be evaluated.
- `fun` target function to be minimized
- `creationFunction` Function to create individuals/solutions in search space
- `control` (list), with the options
  - `budget` The limit on number of target function evaluations (stopping criterion) (default: 100)
  - `vectorized` Boolean. Defines whether target function is vectorized (takes a list of solutions as argument) or not (takes single solution as argument). Default: FALSE

Value

a list:
- `xbest` best solution found
- `ybest` fitness of the best solution
- `x` history of all evaluated solutions
- `y` corresponding target function values f(x)
- `count` number of performed target function evaluations

See Also

`optimCEGO`, `optimEA`

Examples

```r
seed=0
gleakseed=1
#distance
dF <- distancePermutationHamming
#mutation
```
mF <- mutationPermutationSwap
# recombination
rF <- recombinationPermutationCycleCrossover
# creation
cF <- function(){sample(5)}
# objective function
lf <- landscapeGeneratorUNI(ref=1:5,distFun=dF)
# start optimization
set.seed(seed)
res <- optimRS(lF,cF,list(budget=100))
res$x$xbest

---

**predict.CKriging**  
*Predict: Combinatorial Kriging*

**Description**

Predict with a model fit resulting from `combinatorialKriging`.

**Usage**

```r
## S3 method for class 'CKriging'
predict(object, x, predAll = FALSE, ...)
```

**Arguments**

- `object`: fit of the Kriging model (settings and parameters), of class `CKriging`.
- `x`: list of samples to be predicted
- `predAll`: if TRUE return all (RMSE and prediction, in a dataframe), else return only prediction
- `...`: further arguments, not used

**Value**

Returned value depends on the setting of `predAll`
TRUE: data.frame with 2 columns for function value `f` and uncertainty estimate `s` (RMSE)
FALSE: data.frame with column `f` only

**See Also**

`combinatorialKriging`
**predict.CLM**

**Predict: Combinatorial Kriging**

**Description**

Predict with a combinatorialLM fit.

**Usage**

```r
## S3 method for class 'CLM'
predict(object, x, ...)```

**Arguments**

- `object`: fit of the Kriging model (settings and parameters), of class CLM.
- `x`: list of samples to be predicted
- `...`: further arguments, not used

**Value**

numeric vector of predictions

**See Also**

`combinatorialLM`

---

**predict.CRBFN**

**Predict: Combinatorial RBFN**

**Description**

Predict with a model fit resulting from combinatorialRBFN.

**Usage**

```r
## S3 method for class 'CRBFN'
predict(object, x, predAll = FALSE, ...)```

**Arguments**

- `object`: fit of the RBFN model (settings and parameters), of class CRBFN.
- `x`: list of samples to be predicted
- `predAll`: if TRUE return all (RMSE and prediction, in a dataframe), else return only prediction
- `...`: further arguments, not used
recombinationPermutationCycleCrossover

Value

Returned value depends on the setting of predAll
TRUE: data.frame with 2 columns for function value \( f \) and uncertainty estimate \( s \) (RMSE)
FALSE: data.frame with column \( f \) only

See Also

combinatorialRBFN

recombinationBinaryUniformCrossoverFast

*Uniform Crossover for Bit Strings*

Description

Given a population of bit-strings, this function recombines each individual with another individual by randomly picking bits from each parent. Note, that optima will not pass the whole population to recombination functions, but only the chosen parents.

Usage

`recombinationBinaryUniformCrossoverFast(population, parameters)`

Arguments

- `population`: List of bit-strings
- `parameters`: not used

Value

population of recombined offspring

recombinationPermutationCycleCrossover

*Cycle Crossover (CX) for Permutations*

Description

Given a population of permutations, this function recombines each individual with another random individual. Note, that optima will not pass the whole population to recombination functions, but only the chosen parents.

Usage

`recombinationPermutationCycleCrossover(population, parameters)`
solutionFunctionGeneratorBinary

Arguments
- population: List of permutations
- parameters: not used

Value
- population of recombined offspring

Description
Returns a function that generates random bit-strings of length N. Can be used to create individuals of NK-Landscapes or other problems with binary representation.

Usage
solutionFunctionGeneratorBinary(N)

Arguments
- N: length of the bit-strings

Value
- returns a function, without any arguments

solutionFunctionGeneratorPermutation

Permutation Generator Function

Description
Returns a function that generates random permutations of length N. Can be used to generate individual solutions for permutation problems, e.g., Travelling Salesperson Problem

Usage
solutionFunctionGeneratorPermutation(N)

Arguments
- N: length of the permutations returned
Value

returns a function, without any arguments
Index

+Topic package
  CEGO-package, 3
  benchmarkGeneratorFSP, 3, 5–7
  benchmarkGeneratorWKL, 4
  benchmarkGeneratorQAP, 4, 5, 6, 7
  benchmarkGeneratorTSP, 4, 5, 6, 7
  benchmarkGeneratorWT, 4–6, 7
  buildModel, 39
  CEGO (CEGO-package), 3
  CEGO-package, 3
  combinatorialKriging, 8, 38, 39, 44
  combinatorialLLM, 10, 38, 39, 45
  combinatorialRBFN, 12, 38, 39, 45, 46
  distancePermutationAdjacency, 13
  distancePermutationChebyshev, 14
  distancePermutationCos, 15
  distancePermutationEuclidean, 16
  distancePermutationHamming, 17
  distancePermutationInsert, 17, 19
  distancePermutationInterchange, 18
  distancePermutationLCSeq, 17, 19
  distancePermutationLCStr, 20
  distancePermutationLee, 21
  distancePermutationLevenshtein, 22
  distancePermutationLex, 23, 33
  distancePermutationManhattan, 24
  distancePermutationPosition, 25
  distancePermutationPosition2, 26
  distancePermutationR, 27
  distancePermutationSwap, 28
  infillExpectedImprovement, 29
  landscapeGeneratorGaussian, 29, 31, 32
  landscapeGeneratorMUL, 31, 32
  landscapeGeneratorUNI, 31, 32
  lexicographicPermutationOrderNumber, 23, 33
  longestIncreasingSubsequence, 33
  mutationBinary, 34
  mutationBinaryFast, 35
  mutationPermutationInterchange, 35
  mutationPermutationSwap, 36, 40
  optim, 41
  optim2Opt, 36, 41
  optimCEGO, 3, 37, 41, 43
  optimEA, 39, 39, 43, 46
  optimInterface, 8, 41
  optimRS, 41, 43
  predict.CKriging, 10, 44
  predict.CLM, 11, 45
  predict.CRBFN, 13, 45
  recombinationBinaryUniformCrossoverFast, 46
  recombinationPermutationCycleCrossover, 40, 46
  solutionFunctionGeneratorBinary, 47
  solutionFunctionGeneratorPermutation, 47

49