Package ‘mcka’
May 12, 2016

Type Package
Title Machine Coded Genetic Algorithms for Real-Valued Optimization Problems
Version 3.0.1
Date 2016-05-12
Author Mehmet Hakan Satman
Maintainer Mehmet Hakan Satman <mhsatman@istanbul.edu.tr>
Description Machine coded genetic algorithm (MCGA) is a fast tool for real-valued optimization problems. It uses the byte representation of variables rather than real-values. It performs the classical crossover operations (uniform) on these byte representations. Mutation operator is also similar to classical mutation operator, which is to say, it changes a randomly selected byte value of a chromosome by +1 or -1 with probability 1/2. In MCGAs there is no need for encoding-decoding process and the classical operators are directly applicable on real-values. It is fast and can handle a wide range of a search space with high precision. Using a 256-unary alphabet is the main disadvantage of this algorithm but a moderate size population is convenient for many problems. Package also includes multi_mcka function for multi objective optimization problems. This function sorts the chromosomes using their ranks calculated from the non-dominated sorting algorithm.
License GPL (>= 2)
Depends GA
Imports Rcpp (>= 0.11.4)
LinkingTo Rcpp
NeedsCompilation yes
LazyLoad yes
Repository CRAN
Date/Publication 2016-05-12 16:24:50
RoxygenNote 5.0.1
## R topics documented:

- **mcga-package** ................................................................. 2
- **arithmetic_crossover** .................................................... 4
- **blx_crossover** ............................................................. 5
- **ByteCodeMutation** ......................................................... 6
- **ByteCodeMutationUsingDoubles** .................................... 7
- **ByteCodeMutationUsingDoublesRandom** .......................... 8
- **BytesToDouble** ........................................................... 9
- **ByteVectorToDoubles** ..................................................... 9
- **byte_crossover** ........................................................... 10
- **byte_crossover_1p** ........................................................ 11
- **byte_crossover_2p** ........................................................ 12
- **byte_mutation** ............................................................. 13
- **byte_mutation_dynamic** .................................................. 14
- **byte_mutation_random** ................................................... 15
- **byte_mutation_random_dynamic** ..................................... 16
- **DoubleToBytes** ........................................................... 17
- **DoubleVectorToBytes** .................................................... 18
- **EnsureBounds** ............................................................. 18
- **flat_crossover** ............................................................ 19
- **linear_crossover** .......................................................... 20
- **MaxDouble** ................................................................. 21
- **mcga** ........................................................................... 22
- **mcga2** ........................................................................... 23
- **multi_mcga** .................................................................... 25
- **OnePointCrossOver** ........................................................ 27
- **OnePointCrossOverOnDoublesUsingBytes** .......................... 28
- **sbx_crossover** .............................................................. 29
- **SizeOfDouble** .............................................................. 30
- **SizeOfInt** ....................................................................... 30
- **SizeOfLong** .................................................................... 31
- **TwoPointCrossOver** ........................................................ 31
- **TwoPointCrossOverOnDoublesUsingBytes** .......................... 32
- **unfair_average_crossover** .............................................. 33
- **UniformCrossOver** .......................................................... 34
- **UniformCrossOverOnDoublesUsingBytes** .......................... 35

**Index** ............................................................................ 36

---

Description

Machine coded genetic algorithm (MCGA) is a fast tool for real-valued optimization problems. It uses the byte representation of variables rather than real-values. It performs the classical crossover operations (uniform) on these byte representations. Mutation operator is also similar to classical mutation operator, which is to say, it changes a randomly selected byte value of a chromosome by +1 or -1 with probability 1/2. In MCGAs there is no need for encoding-decoding process and the classical operators are directly applicable on real-values. It is fast and can handle a wide range of a search space with high precision. Using a 256-unary alphabet is the main disadvantage of this algorithm but a moderate size population is convenient for many problems.

Details

Package: mega
Type: Package
Version: 2.0.3
Date: 2012-01-06
License: GPL
LazyLoad: yes

Author(s)

Mehmet Hakan Satman
Maintainer: Mehmet Hakan Satman <mhsatman@istanbul.edu.tr>

Examples

```r
## Not run:
# A sample optimization problem
# Min f(x) = (x1-7)^2 + (x2-77)^2 + (x3-777)^2 + (x4-7777)^2 + (x5-77777)^2
# The range of x is unknown. The solution is
# x1 = 7
# x2 = 77
# x3 = 777
# x4 = 7777
# x5 = 77777
# Min f(x) = 0
require("mcga")
f <- function(x){
  return ((x[1]-7)^2 + (x[2]-77)^2 + (x[3]-777)^2 + (x[4]-7777)^2 + (x[5]-77777)^2)
}
m <- mcga(popsize=200,
  chsize=5,
  minval=0.0,
  maxval=999999999.9,
  maxiter=2500,
  crossprob=1.0,
```
arithmetic_crossover

Performs arithmetic crossover operation on a pair of two selected parent candidate solutions

Description

This function is not called directly but is given as a parameter in GA::ga function. In GA::ga, if the parameter crossover is set to arithmetic_crossover than the arithmetic crossover operator is applied in the genetic search. arithmetic_crossover generates offspring using the weighted mean of parents' genes. Weights are drawn randomly.

Usage

arithmetic_crossover(object, parents, ...)

Arguments

object A GA::ga object
parents Indices of the selected parents
... Additional arguments to be passed to the function

Value

List of two generated offspring

Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

Examples

f <- function(x){
  return(-sum( (x-5)^2 ) )
}
myga <- ga(type="real-valued", fitness = f, popSize = 100, maxiter = 100,
          min = rep(-50,5), max = rep(50,5), crossover = arithmetic_crossover)
print(myga@solution)
**blx_crossover**

**Performs blx (blend) crossover operation on a pair of two selected parent candidate solutions**

**Description**

This function is not called directly but is given as a parameter in `GA::ga` function. In `GA::ga`, if the parameter `crossover` is set to `blx_crossover` than the blx crossover operator is applied in the genetic search.

**Usage**

`blx_crossover(object, parents, ...)`

**Arguments**

- **object**
  - A `GA::ga` object
- **parents**
  - Indices of the selected parents
- **...**
  - Additional arguments to be passed to the function

**Value**

List of two generated offspring

**Author(s)**

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

**Examples**

```r
f <- function(x)
  return(-sum((x-5)^2))
myga <- ga(type="real-valued", fitness = f, popSize = 100, maxiter = 100,
  min = rep(-50,5), max = rep(50,5), crossover = blx_crossover)
print(myga@solution)
```
**ByteCodeMutation**

*Mutation operator for byte representation of double values*

**Description**

This function is a C++ wrapper for mutating byte representation of a given candidate solution.

**Usage**

```
ByteCodeMutation(bytes1L pmutation)
```

**Arguments**

- `bytes1`: A vector of bytes of a candidate solution
- `pmutation`: Probability of mutation

**Value**

Byte vector of mutated solution

**Author(s)**

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

**See Also**

ByteCodeMutationUsingDoubles

**Examples**

```
set.seed(1246)
print(pi)
bytes <- DoubleToBytes(pi)
mutated.bytes <- ByteCodeMutation(bytes, 0.10)
new.var <- BytesToDouble(mutated.bytes)
print(new.var)
```
ByteCodeMutationUsingDoubles

Mutation operator for byte representation of double values

Description

This function is a C++ wrapper for mutating byte representation of a given candidate solution.

Usage

ByteCodeMutationUsingDoubles(d, pmutation)

Arguments

d A vector of doubles
pmutation Probability of mutation

Value

Double vector of mutated solution

Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

See Also

ByteCodeMutation

Examples

set.seed(1246)
print(pi)
print(exp(1))
new.var <- ByteCodeMutationUsingDoubles(c(pi, exp(1)), 0.10)
print(new.var)
ByteCodeMutationUsingDoublesRandom

*Mutation operator for byte representation of double values*

**Description**

This function is a C++ wrapper for mutating byte representation of a given candidate solution. This mutation operator randomly changes a byte in the range of [0,255].

**Usage**

```
ByteCodeMutationUsingDoublesRandom(d, pmutation)
```

**Arguments**

- `d`: A vector of doubles
- `pmutation`: Probability of mutation

**Value**

Double vector of mutated solution

**Author(s)**

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

**See Also**

ByteCodeMutation

**Examples**

```
set.seed(1246)
print(pi)
print(exp(1))
new.var <- ByteCodeMutationUsingDoublesRandom(c(pi, exp(1)), 0.10)
print(new.var)
```
BytesToDouble  

Converting sizeof(double) bytes to a double value

**Description**

This function converts sizeof(double) bytes to a double typed value.

**Usage**

BytesToDouble(x)

**Arguments**

x  
A vector of bytes (unsigned chars in C++)

**Value**

Corresponding double typed value for a given vector of bytes

**Author(s)**

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

**See Also**

DoubleClickVector
DoubleBytes
ByteVectorToDoubles

**Examples**

print(BytesToDouble(DoubleToBytes(56.43)))

---

ByteVectorToDoubles  

Converting p * sizeof(double) bytes to a vector of p double values

**Description**

This function converts a byte vector to a vector of doubles

**Usage**

ByteVectorToDoubles(b)
Arguments

\( b \)

A vector of bytes (unsigned chars in C++)

Value

Corresponding vector of double typed values for a given vector of bytes

Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

See Also

DoubleVectorToBytes

BytesToDouble

ByteVectorToDoubles

Examples

```r
a <- DoubleVectorToBytes(c(56.54, 89.7666, 98.565))
b <- ByteVectorToDoubles(a)
print(b)
```

---

**byte_crossover**

*Performs crossover operation on a pair of two selected parent candidate solutions*

Description

This function is not called directly but is given as a parameter in `GA::ga` function. In `GA::ga`, if the parameter `crossover` is set to `byte_crossover` than the byte-coded crossover operator is applied in the genetic search. In `mcga2` function, the hard-coded crossover parameter is set to `byte_crossover` by definition. `byte_crossover` function simply takes two double vectors (parents) and combines the bytes of doubles using a Uniform distribution with parameters 0 and 1.

Usage

`byte_crossover(object, parents, ...)`

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>A <code>GA::ga</code> object</td>
</tr>
<tr>
<td><code>parents</code></td>
<td>Indices of the selected parents</td>
</tr>
<tr>
<td><code>...</code></td>
<td>Additional arguments to be passed to the function</td>
</tr>
</tbody>
</table>
Value

List of two generated offspring

Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

References


See Also

mcga2

Examples

```r
f <- function(x){
  return(-sum((x-5)^2))
}
myga <- GA::ga(type="real-valued", fitness = f, popSize = 100, maxiter = 200,
               min = rep(-50,5), max = rep(50,5), crossover = byte_crossover,
               mutation = byte_mutation)
print(myga@solution)
```

---

`byte_crossover_1p` performs one-point crossover operation on a pair of two selected parent candidate solutions

Description

This function is not called directly but is given as a parameter in GA::ga function. In GA::ga, if the parameter crossover is set to `byte_crossover_1p` than the byte-coded one-point crossover operator is applied in the genetic search. In mcga2 function, the hard-coded crossover parameter is set to `byte_crossover` by definition. `byte_crossover_1p` function simply takes two double vectors (parents) and combines the bytes of doubles using given cut-point.

Usage

`byte_crossover_1p(object, parents, ...)`

Arguments

- `object`: A GA::ga object
- `parents`: Indices of the selected parents
- `...`: Additional arguments to be passed to the function
Value
List of two generated offspring

Author(s)
Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

References

See Also
mcga2

Examples
```r
f <- function(x){
  return(-sum((x-5)^2))
}
myga <- GA::ga(type="real-valued", fitness = f, popSize = 100, maxiter = 200,
               min = rep(-50,5), max = rep(50,5), crossover = byte_crossover_1p,
               mutation = byte_mutation)
print(myga@solution)
```

---

`byte_crossover_2p` **Performs two-point crossover operation on a pair of two selected parent candidate solutions**

Description
This function is not called directly but is given as a parameter in `GA::ga` function. In `GA::ga`, if the parameter `crossover` is set to `byte_crossover_2p` than the byte-coded two-point crossover operator is applied in the genetic search. In `mcga` function, the hard-coded crossover parameter is set to `byte_crossover` by definition. `byte_crossover_2p` function simply takes two double vectors (parents) and combines the bytes of doubles using given `cutpoint1` and `cutpoint2`.

Usage
```r
byte_crossover_2p(object, parents, ...)
```

Arguments
- **object**: A `GA::ga` object
- **parents**: Indices of the selected parents
- **...**: Additional arguments to be passed to the function
**Value**

List of two generated offspring

**Author(s)**

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

**References**


**See Also**

mcga2

**Examples**

```r
f <- function(x){
  return(-sum((x-5)^2))
}
myga <- GA::ga(type="real-valued", fitness = f, popSize = 100, maxiter = 200,
  min = rep(-50,5), max = rep(50,5), crossover = byte_crossover_2p,
  mutation = byte_mutation)
print(myga@solution)
```

---

**byte_mutation**

*Performs mutation operation on a given double vector*

**Description**

This function is not called directly but is given as a parameter in GA::ga function. In GA::ga, if the parameter `mutation` is set to `byte_mutation` than the byte-coded mutation operator is applied in the genetic search. In mcga2 function, the hard-coded mutation parameter is set to `byte_mutation` by definition. Byte-mutation function simply takes an double vector and changes bytes of this values by +1 or -1 using the pre-determined mutation probability.

**Usage**

```r
byte_mutation(object, parent, ...)
```

**Arguments**

- `object` A GA::ga object
- `parent` Index of the candidate solution of the current population
- `...` Additional arguments to be passed to the function
**Value**

Mutated double vector

**Author(s)**

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

**References**


**Examples**

```r
f <- function(x){
  return(-sum( (x-5)^2 ) )
}
myga <- GA::ga(type="real-valued", fitness = f, popSize = 100, maxiter = 200,
  min = rep(-50,5), max = rep(50,5), crossover = byte_crossover,
  mutation = byte_mutation)
print(myga@solution)
```

---

**byte_mutation_dynamic**

*Performs mutation operation on a given double vector using dynamic mutation probabilities*

**Description**

This function is not called directly but is given as a parameter in GA::ga function. In GA::ga, if the parameter mutation= is set to byte_mutation_dynamic then the byte-coded mutation operator is applied in the genetic search. In mcga2 function, the hard-coded mutation parameter is set to byte_mutation by definition. Byte-mutation function simply takes an double vector and changes bytes of this values by +1 or -1 using the dynamically decreased and pre-determined mutation probability.

**Usage**

`byte_mutation_dynamic(object, parent, ...)`

**Arguments**

- `object` A GA::ga object
- `parent` Index of the candidate solution of the current population
- `...` Additional arguments to be passed to the function

**Value**

Mutated double vector
Author(s)
Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

References

Examples

```r
f <- function(x){
  return(-sum((x-5)^2 ))
}
myga <- GA::ga(type="real-valued", fitness = f, popSize = 100, maxiter = 200,
              min = rep(-50,5), max = rep(50,5), crossover = byte_crossover,
              mutation = byte_mutation_dynamic, pmutation = 0.10)
print(myga@solution)
```

Description
This function is not called directly but is given as a parameter in `GA::ga` function. In `GA::ga`, if the parameter `mutation` is set to `byte_mutation_random` than the byte-coded mutation operator is applied in the genetic search. In `mcga2` function, the hard-coded mutation parameter is set to `byte_mutation` by definition. This function simply takes an double vector and changes bytes randomly in the range of [0,255] using the pre-determined mutation probability.

Usage

`byte_mutation_random(object, parent, ...)`

Arguments

- `object` A `GA::ga` object
- `parent` Index of the candidate solution of the current population
- `...` Additional arguments to be passed to the function

Value

Mutated double vector

Author(s)
Mehmet Hakan Satman - mhsatman@istanbul.edu.tr
References


Examples

```r
f <- function(x){
  return(-sum((x-5)^2))
}
myga <- GA::ga(type="real-valued", fitness = f, popSize = 100, maxiter = 200,
              min = rep(-50,5), max = rep(50,5), crossover = byte_crossover,
              mutation = byte_mutation_random, pmutation = 0.20)
print(myga$solution)
```

---

`byte_mutation_random_dynamic`

*Performs mutation operation on a given double vector with dynamic mutation probabilities*

Description

This function is not called directly but is given as a parameter in `GA::ga` function. In `GA::ga`, if the parameter `mutation` is set to `byte_mutation_random_dynamic` than the byte-coded mutation operator with dynamic probabilities is applied in the genetic search. In `mcga` function, the hard-coded mutation parameter is set to `byte_mutation` by definition. This function simply takes an double vector and changes bytes randomly in the range of [0,255] using the decreasing values of pre-determined mutation probability by generations.

Usage

`byte_mutation_random_dynamic(object, parent, ...)

Arguments

- `object` A `GA::ga` object
- `parent` Index of the candidate solution of the current population
- `...` Additional arguments to be passed to the function

Value

Mutated double vector

Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr
DoubleToBytes

References


Examples

f <- function(x){
  return(-sum( (x-5)^2 )
}

# Increase popSize and maxiter for more precise solutions
myga <- GA::ga(type="real-valued", fitness = f, popSize = 100, maxiter = 200,
  min = rep(-50,5), max = rep(50,5), crossover = byte_crossover,
  mutation = byte_mutation_random_dynamic, pmutation = 0.20)
print(myga@solution)

print(DoubleToBytes(56.43))

DoubleToBytes

Description

This function returns a vector of byte values with the length of sizeof(double) for a given double typed value

Usage

DoubleToBytes(x)

Arguments

x

A double typed value

Value

A vector of byte values with the length of sizeof(double) for a given double typed value

Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

See Also

DoubleVectorToBytes
BytesToDouble
ByteVectorToDoubles

Examples

print(DoubleToBytes(56.43))
**DoubleVectorToBytes**

*Byte representation of a vector of double typed variables*

**Description**

This function returns a vector of byte values for a given vector of double typed values.

**Usage**

`DoubleVectorToBytes(d)`

**Arguments**

- `d` A vector of double typed values

**Value**

returns a vector of byte values for a given vector of double typed values

**Author(s)**

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

**See Also**

- `DoubleToBytes`
- `BytesToDouble`
- `ByteVectorToDoubles`

**Examples**

```
print(DoubleVectorToBytes(c(56.54, 89.7666, 98.565)))
```

---

**EnsureBounds**

*Altering vector of doubles to satisfy boundary constraints*

**Description**

Byte based crossover and mutation operators can generate variables out of bounds of the decision variables. This function controls if variables are between their lower and upper bounds and if not, draws random numbers between these ranges. This function directly modifies the argument doubles and does not return a value.

**Usage**

`EnsureBounds(doubles, mins, maxs)`
**flat_crossover**

### Arguments

- **doubles**: A vector of doubles
- **mins**: A vector of lower bounds of decision variables
- **maxs**: A vector of upper bounds of decision variables

### Value

Function directly modifies the argument doubles and does not return a result.

### Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

### See Also

- `byte_crossover`
- `byte_mutation`
- `mcga2`

### Examples

```r
set.seed(1234)
x <- runif(10)
print(x)
# [1] 0.113703411 0.622299405 0.609274733 0.623379442 0.860915384 0.640310605
# [7] 0.009495756 0.232550506 0.666083758 0.514251141
EnsureBounds(x, mins=rep(0,10), maxs=rep(0.2,10))
print(x)
# [1] 0.113703411 0.138718258 0.108994967 0.056546717 0.184686697 0.058463168
# [7] 0.009495756 0.167459126 0.057244657 0.053364156
```

---

**flat_crossover**  
Performs flat crossover operation on a pair of two selected parent candidate solutions

### Description

This function is not called directly but is given as a parameter in `GA::ga` function. In `GA::ga`, if the parameter `crossover` is set to `flat_crossover` than the flat crossover operator is applied in the genetic search. `flat_crossover` draws a random number between parents’ genes and returns a pair of generated offspring.

### Usage

```r
flat_crossover(object, parents, ...)```
linear_crossover

Arguments

object A GA::ga object
parents Indices of the selected parents
... Additional arguments to be passed to the function

Value

List of two generated offspring

Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

Examples

```r
f <- function(x){
  return(-sum((x-5)^2))
}
myga <- ga(type="real-valued", fitness = f, popSize = 100, maxiter = 100,
  min = rep(-50, 5), max = rep(50, 5), crossover = flat_crossover)
print(myga@solution)
```

linear_crossover Performs linear crossover operation on a pair of two selected parent candidate solutions

Description

This function is not called directly but is given as a parameter in GA::ga function. In GA::ga, if the parameter crossover= is set to linear_crossover than the linear crossover operator is applied in the genetic search. linear_crossover generates three offspring and performs a selection mechanism to determine best two of them.

Usage

```r
linear_crossover(object, parents, ...)
```

Arguments

object A GA::ga object
parents Indices of the selected parents
... Additional arguments to be passed to the function

Value

List of two generated offspring
MaxDouble

Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

Examples

```
f <- function(x){
  return(-sum( (x-5)^2 ) )
}
myga <- ga(type="real-valued", fitness = f, popSize = 100, maxiter = 100,
          min = rep(-50,5), max = rep(50,5), crossover = linear_crossover)
print(myga@solution)
```

MaxDouble

Maximum value of a double typed variable

Description

Maximum value of a double typed variable

Usage

MaxDouble()

Value

Returns maximum value of a double typed variable in C++ compiler

Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

Examples

```
print(MaxDouble())
```
**mcka**

Performs machine coded genetic algorithms on a function subject to be minimized.

**Description**

Machine coded genetic algorithm (MCGA) is a fast tool for real-valued optimization problems. It uses the byte representation of variables rather than real-values. It performs the classical crossover operations (uniform) on these byte representations. Mutation operator is also similar to classical mutation operator, which is to say, it changes a randomly selected byte value of a chromosome by +1 or -1 with probability 1/2. In MCGAs there is no need for encoding-decoding process and the classical operators are directly applicable on real-values. It is fast and can handle a wide range of a search space with high precision. Using a 256-unary alphabet is the main disadvantage of this algorithm but a moderate size population is convenient for many problems.

**Usage**

`
mcka(popsize, chsize, crossprob = 1.0, mutateprob = 0.01,
crossprob = 1, minval, maxval, maxiter = 10, evalFunc)
``

**Arguments**

- **popsize**
  Number of chromosomes.
- **chsize**
  Number of parameters.
- **crossprob**
  Crossover probability. By default it is 1.0
- **mutateprob**
  Mutation probability. By default it is 0.01
- **elitism**
  Number of best chromosomes to be copied directly into next generation. By default it is 1
- **minval**
  The lower bound of the randomized initial population. This is not a constraint for parameters.
- **maxval**
  The upper bound of the randomized initial population. This is not a constraint for parameters.
- **maxiter**
  The maximum number of generations. By default it is 10
- **evalFunc**
  An R function. By default, each problem is a minimization.

**Value**

- **population**
  Sorted population resulted after generations
- **costs**
  Cost values for each chromosomes in the resulted population

**Author(s)**

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr
References


Examples

# A sample optimization problem
# Min f(x) = (x1-7)^2 + (x2-77)^2 + (x3-777)^2 + (x4-7777)^2 + (x5-77777)^2
# The range of x_i is unknown. The solution is
# x1 = 7
# x2 = 77
# x3 = 777
# x4 = 7777
# x5 = 77777
# Min f(x) = 0
require("mcga")
f <- function(x){
  return ((x[1]-7)^2 + (x[2]-77)^2 + (x[3]-777)^2 + (x[4]-7777)^2 + (x[5]-77777)^2)
}
m <- mcga(fitness = f, 
          min = rep(0, 5), 
          max = rep(999999999, 5), 
          popsize = 200, 
          chsize = 5, 
          elitism = base::max(1, round(popsize/10)), 
          pcrossover = 1, 
          pmutation = 0.01, 
          evalFunc = f)

cat("Best chromosome:

print(m$population[1,])
cat("Cost: ", m$costs[1], "\n")

mcga2

Perform a machine-coded genetic algorithm search for a given optimization problem

Description

mcga2 is the improvement version of the standard mcga function as it is based on the GA::ga function. The byte_crossover and the byte_mutation operators are the main reproduction operators and these operators uses the byte representations of parents in the computer memory.

Usage

mcga2(fitness, ..., min, max, 
      population = gaControl("real-valued")$population, 
      selection = gaControl("real-valued")$selection, 
      crossover = byte_crossover, mutation = byte_mutation, popSize = 50, 
      pcrossover = 0.8, pmutation = 0.1, elitism = base::max(1, round(popSize/10)), 
      ...)

mcga
* 0.05), maxiter = 100, run = maxiter, maxFitness = Inf, names = NULL, parallel = FALSE, monitor = gaMonitor, seed = NULL)

Arguments

- **fitness**: The goal function to be maximized
- **min**: Vector of lower bounds of variables
- **max**: Vector of upper bounds of variables
- **population**: Initial population. It is gaControl("real-valued")$population by default.
- **selection**: Selection operator. It is gaControl("real-valued")$selection by default.
- **crossover**: Crossover operator. It is byte_crossover by default.
- **mutation**: Mutation operator. It is byte_mutation by default. Other values can be given including byte_mutation_random, byte_mutation_dynamic and byte_mutation_random_dynamic
- **popSize**: Population size. It is 50 by default
- **pcrossover**: Probability of crossover. It is 0.8 by default
- **pmutation**: Probability of mutation. It is 0.1 by default
- **elitism**: Number of elitist solutions. It is base::max(1, round(popSize*0.05)) by default
- **maxiter**: Maximum number of generations. It is 100 by default
- **run**: The genetic search is stopped if the best solution has not any improvements in last run generations. By default it is maxiter
- **maxFitness**: Upper bound of the fitness function. By default it is Inf
- **names**: Vector of names of the variables. By default it is NULL
- **parallel**: If TRUE, fitness calculations are performed parallel. It is FALSE by default
- **monitor**: The monitoring function for printing some information about the current state of the genetic search. It is gaMonitor by default
- **seed**: The seed for random number generating. It is NULL by default

Value

Returns an object of class ga-class

Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

References

**multi_mcga**

**See Also**

GA::ga

**Examples**

```r
f <- function(x){
  return(-sum( (x-M)^2 ) )
}
myga <- mcga2(fitness = f, popSize = 100, maxIter = 300, 
               min = rep(-50,5), max = rep(50,5))
print(myga@solution)
```

---

**multi_mcga**  
*Performs multi objective machine coded genetic algorithms.*

**Description**

Machine coded genetic algorithm (MCGA) is a fast tool for real-valued optimization problems. It uses the byte representation of variables rather than real-values. It performs the classical crossover operations (uniform) on these byte representations. Mutation operator is also similar to classical mutation operator, which is to say, it changes a randomly selected byte value of a chromosome by +1 or -1 with probability 1/2. In MCGAs there is no need for encoding-decoding process and the classical operators are directly applicable on real-values. It is fast and can handle a wide range of a search space with high precision. Using a 256-unary alphabet is the main disadvantage of this algorithm but a moderate size population is convenient for many problems.

This function performs multi objective optimization using the same logic underlying the mcga. Chromosomes are sorted by their objective values using a non-dominated sorting algorithm.

**Usage**

```r
multi_mcga(popsize, chsize, crossprob = 1.0, mutateprob = 0.01, 
elitism = 1, minval, maxval, maxiter = 10, numfunc, evalFunc)
```

**Arguments**

- `popsize`  
  Number of chromosomes.

- `chsize`  
  Number of parameters.

- `crossprob`  
  Crossover probability. By default it is 1.0

- `mutateprob`  
  Mutation probability. By default it is 0.01

- `elitism`  
  Number of best chromosomes to be copied directly into next generation. By default it is 1

- `minval`  
  The lower bound of the randomized initial population. This is not a constraint for parameters.

- `maxval`  
  The upper bound of the randomized initial population. This is not a constraint for parameters.
multi_mcga

maxiter  The maximum number of generations. By default it is 10.
numfunc  Number of objective functions.
evalFunc  An R function. By default, each problem is a minimization. This function must
          return a cost vector with dimension of numfunc. Each element of this vector
          points to the corresponding function to optimize.

Value

population  Sorted population resulted after generations
costs  Cost values for each chromosomes in the resulted population
ranks  Calculated ranks using a non-dominated sorting for each chromosome

Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

References

in applied mechanics and engineering, 186(2), 311-338.

Examples

```r
# Not run:
# We have two objective functions.
f1<-function(x){
  return(sin(x))
}

f2<-function(x){
  return(sin(2*x))
}

# This function returns a vector of cost functions for a given x sent from mcga
f<-function(x){
  return ( c( f1(x), f2(x) ) )
}

# main loop
m<-multi_mcga(popsize=200, chsize=1, minval= 0, elitism=2,
               maxval= 2.0 * pi, maxiter=1000, crossprob=1.0,
               mutateprob=0.01, evalFunc=f, numfunc=2)

# Points show best five solutions.
curve(f1, 0, 2*pi)
curve(f2, 0, 2*pi, add=TRUE)

p <- m$population[1:5,]
points(p, f1(p))
points(p, f2(p))
```
OnePointCrossOver

## Description

This function is a C++ wrapper for crossing-over of two byte vectors of candidate solutions.

## Usage

```r
OnePointCrossOver(bytes1, bytes2, cutpoint)
```

## Arguments

- `bytes1`: A vector of bytes of the first parent.
- `bytes2`: A vector of bytes of the second parent.
- `cutpoint`: Cut-point for the single point crossing-over.

## Value

- List of two byte vectors of offspring.

## Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

## See Also

- UniformCrossOver
- UniformCrossOverOnDoublesUsingBytes

## Examples

```r
b1 <- DoubleVectorToBytes(c(56.54, 89.7666, 98.565))
b2 <- DoubleVectorToBytes(c(79.76, 56.4443, 34.22121))
result <- OnePointCrossOver(b1, b2, round(runif(1,1,SizeOfDouble() * 3)))
print(ByteVectorToDoubles(result[[1]]))
print(ByteVectorToDoubles(result[[2]]))
```
OnePointCrossOverOnDoublesUsingBytes

One-point Crossover operation on the two vectors of doubles using their byte representations

Description

This function is a C++ wrapper for crossing-over of two double vectors of candidate solutions using their byte representations.

Usage

OnePointCrossOverOnDoublesUsingBytes(d1, d2, cutpoint)

Arguments

d1        A vector of doubles of the first parent

d2        A vector of doubles of the second parent

| cutpoint | An integer between 1 and chromosome length for crossover cutting |

Value

List of two double vectors of offspring

Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

See Also

OnePointCrossOver
UniformCrossOverOnDoublesUsingBytes

Examples

d1 <- runif(3)
d2 <- runif(3)
cutp <- sample(1:(length(d1)*SizeOfDouble() - 1)[1]
offspring <- OnePointCrossOverOnDoublesUsingBytes(d1, d2, cutp)
print("Parents:")
print(d1)
print(d2)
print("Offspring:")
print(offspring[[1]])
print(offspring[[2]])
sbx_crossover

Performs sbx (simulated binary) crossover operation on a pair of two selected parent candidate solutions

Description

This function is not called directly but is given as a parameter in GA::ga function. In GA::ga, if the parameter crossover is set to sbx_crossover than the sbx crossover operator is applied in the genetic search. sbx_crossover mimics the classical single-point crossover operator in binary genetic algorithms.

Usage

sbx_crossover(object, parents, ...)

Arguments

object A GA::ga object
parents Indices of the selected parents
... Additional arguments to be passed to the function

Value

List of two generated offspring

Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

References


Examples

f <- function(x){
  return(-sum( (x-5)^2 )
}
myga <- ga(type="real-valued", fitness = f, popSize = 100, maxiter = 100,
  min = rep(-50,5), max = rep(50,5), crossover = sbx_crossover)
print(myga@solution)
SizeOfDouble  

*Byte-length of a double typed variable*

**Description**

Byte-length of a double typed variable in computer memory

**Usage**

```c
SizeOfDouble()
```

**Value**

Returns the byte-length of a double typed variable in computer memory

**Author(s)**

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

**Examples**

```c
print(SizeOfDouble())
```

SizeOfInt  

*Byte-length of a int typed variable*

**Description**

Byte-length of a int typed variable in computer memory

**Usage**

```c
SizeOfInt()
```

**Value**

Returns the byte-length of a int typed variable in computer memory

**Author(s)**

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

**Examples**

```c
print(SizeOfInt())
```
**SizeOfLong**  
*Byte-length of a long typed variable*

**Description**

Byte-length of a long typed variable in computer memory

**Usage**

SizeOfLong()

**Value**

Returns the byte-length of a long typed variable in computer memory

**Author(s)**

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

**Examples**

print(SizeOfLong())

---

**TwoPointCrossOver**  
*Two Point Crossover operation on the two vectors of bytes*

**Description**

This function is a C++ wrapper for crossing-over of two byte vectors of candidate solutions

**Usage**

TwoPointCrossOver(bytes1, bytes2, cutpoint1, cutpoint2)

**Arguments**

- **bytes1**  
  A vector of bytes of the first parent
- **bytes2**  
  A vector of bytes of the second parent
- **cutpoint1**  
  First cut-point for the single point crossing-over
- **cutpoint2**  
  Second cut-point for the single point crossing-over

**Value**

List of two byte vectors of offspring
TwoPointCrossOverOnDoublesUsingBytes

Description

This function is a C++ wrapper for crossing-over of two double vectors of candidate solutions using their byte representations.

Usage

TwoPointCrossOverOnDoublesUsingBytes(d1, d2, cutpoint1, cutpoint2)

Arguments

d1 A vector of doubles of the first parent
d2 A vector of doubles of the second parent
cutpoint1 An integer between 1 and chromosome length for crossover cutting
cutpoint2 An integer between cutpoint1 and chromosome length for crossover cutting

Value

List of two double vectors of offspring

Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr
**unfair_average_crossover**

Performs unfair average crossover operation on a pair of two selected parent candidate solutions

**Description**

This function is not called directly but is given as a parameter in `GA::ga` function. In `GA::ga`, if the parameter `crossover` is set to `unfair_average_crossover` than the unfair average crossover operator is applied in the genetic search.

**Usage**

```
unfair_average_crossover(object, parents, ...)
```

**Arguments**

- **object**
  
  A `GA::ga` object

- **parents**
  
  Indices of the selected parents

- **...**
  
  Additional arguments to be passed to the function

**Value**

List of two generated offspring

**Author(s)**

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

---

**Examples**

```r
d1 <- runif(3)
d2 <- runif(3)
cutpoints <- sort(sample(1:(length(d1)*sizeofDouble()), 2, replace = FALSE))
offspring <- TwoPointCrossOverOnDoublesUsingBytes(d1,d2,cutpoints[1], cutpoints[2])
print("Parents: ")
print(d1)
print(d2)
print("Offspring: ")
print(offspring[[1]])
print(offspring[[2]])
```
Examples

f <- function(x){
  return(-sum( (x-5)^2 ) )
}
myga <- ga(type="real-valued", fitness = f, popSize = 100, maxIter = 100,
           min = rep(-50,5), max = rep(50,5), crossover = unfair_average_crossover)
print(myga@solution)

---

UniformCrossOver

Uniform Crossover operation on the two vectors of bytes

Description

This function is a C++ wrapper for crossing-over of two byte vectors of candidate solutions.

Usage

UniformCrossOver(bytes1, bytes2)

Arguments

bytes1 A vector of bytes of the first parent
bytes2 A vector of bytes of the second parent

Value

List of two byte vectors of offspring

Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

See Also

OnePointCrossOver
UniformCrossOverOnDoublesUsingBytes

Examples

b1 <- DoubleVectorToBytes(c(56.54, 89.7666, 98.565))
b2 <- DoubleVectorToBytes(c(79.76, 56.4443, 34.22121))
result <- UniformCrossOver(b1,b2)
print(ByteVectorToDoubles(result[[1]]))
print(ByteVectorToDoubles(result[[2]]))
Uniform Crossover operation on the two vectors of doubles using their byte representations

Description

This function is a C++ wrapper for crossing-over of two double vectors of candidate solutions using their byte representations.

Usage

UniformCrossoverOnDoublesUsingBytes(d1, d2)

Arguments

d1 A vector of doubles of the first parent
d2 A vector of doubles of the second parent

Value

List of two double vectors of offspring

Author(s)

Mehmet Hakan Satman - mhsatman@istanbul.edu.tr

See Also

OnePointCrossOver
OnePointCrossOverOnDoublesUsingBytes

Examples

d1 <- runif(3)
d2 <- runif(3)
offspring <- UniformCrossoverOnDoublesUsingBytes(d1, d2)
print("Parents:")
print(d1)
print(d2)
print("Offspring:")
print(offspring[[1]])
print(offspring[[2]])
<table>
<thead>
<tr>
<th>Index</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic_crossover, 4</td>
<td></td>
</tr>
<tr>
<td>blx_crossover, 5</td>
<td></td>
</tr>
<tr>
<td>byte_crossover, 10</td>
<td></td>
</tr>
<tr>
<td>byte_crossover_1p, 11</td>
<td></td>
</tr>
<tr>
<td>byte_crossover_2p, 12</td>
<td></td>
</tr>
<tr>
<td>byte_mutation, 13</td>
<td></td>
</tr>
<tr>
<td>byte_mutation_dynamic, 14</td>
<td></td>
</tr>
<tr>
<td>byte_mutation_random, 15</td>
<td></td>
</tr>
<tr>
<td>byte_mutation_random_dynamic, 16</td>
<td></td>
</tr>
<tr>
<td>ByteCodeMutation, 6</td>
<td></td>
</tr>
<tr>
<td>ByteCodeMutationUsingDoubles, 7</td>
<td></td>
</tr>
<tr>
<td>ByteCodeMutationUsingDoublesRandom, 8</td>
<td></td>
</tr>
<tr>
<td>BytesToDouble, 9</td>
<td></td>
</tr>
<tr>
<td>ByteVectorToDoubles, 9</td>
<td></td>
</tr>
<tr>
<td>DoubleToBytes, 17</td>
<td></td>
</tr>
<tr>
<td>DoubleVectorToBytes, 18</td>
<td></td>
</tr>
<tr>
<td>EnsureBounds, 18</td>
<td></td>
</tr>
<tr>
<td>flat_crossover, 19</td>
<td></td>
</tr>
<tr>
<td>linear_crossover, 20</td>
<td></td>
</tr>
<tr>
<td>MaxDouble, 21</td>
<td></td>
</tr>
<tr>
<td>mcga, 22</td>
<td></td>
</tr>
<tr>
<td>mcga-package, 2</td>
<td></td>
</tr>
<tr>
<td>mcga2, 23</td>
<td></td>
</tr>
<tr>
<td>multi.mcga, 25</td>
<td></td>
</tr>
<tr>
<td>OnePointCrossOver, 27</td>
<td></td>
</tr>
<tr>
<td>OnePointCrossOverOnDoublesUsingBytes, 28</td>
<td></td>
</tr>
<tr>
<td>sbx_crossover, 29</td>
<td></td>
</tr>
<tr>
<td>SizeOfDouble, 30</td>
<td></td>
</tr>
<tr>
<td>SizeOfInt, 30</td>
<td></td>
</tr>
<tr>
<td>SizeOfLong, 31</td>
<td></td>
</tr>
<tr>
<td>TwoPointCrossOver, 31</td>
<td></td>
</tr>
</tbody>
</table>