R documentation

of ‘Bag.Rd’ etc.

February 8, 2006

R topics documented:

Bag 4
print.Bag 5
summary.Bag 6
length.Bag 6
Gene 7
reInit.Gene 8
generateRandom.Gene 9
mutate.Gene 10
print.Gene 11
as.double.Gene 12
as.matrix.Gene 12
summary.Gene 13
newCollection.Gene 14
newRandomCollection.Gene 15
Chromosome 16
reInit.Chromosome 18
clone.Chromosome 18
genes.Chromosome 19
generateRandom.Chromosome 20
length.Chromosome 21
summary.Chromosome 22
print.Chromosome 22
as.double.Chromosome 23
mutate.Chromosome 24
decode.Chromosome 25
newCollection.Chromosome 26
newRandomCollection.Chromosome 27
Niche 28
reInit.Niche 31
clone.Niche 32
generateRandom.Niche 33
length.Niche 34
as.matrix.Niche 34
as.double.Niche 35
print.Niche 36

summary.Niche . . . 37
refreshStats.Niche . . . 37
plot.Niche . . . 38
newCollection.Niche . . . 40
newRandomCollection.Niche . . . 41
mutate.Niche . . . 42
best.Niche . . . 43
max.Niche . . . 44
bestFitness.Niche . . . 45
maxFitness.Niche . . . 46
getFitness.Niche . . . 47
crossover.Niche . . . 47
offspring.Niche . . . 48
progeny.Niche . . . 49
evaluate.Niche . . . 50
World . . . 51
reInit.World . . . 54
clone.World . . . 54
generateRandom.World . . . 55
length.World . . . 56
print.World . . . 57
summary.World . . . 58
refreshStats.World . . . 58
plot.World . . . 59
newCollection.World . . . 61
newRandomCollection.World . . . 62
best.World . . . 63
max.World . . . 64
bestFitness.World . . . 65
maxFitness.World . . . 66
progeny.World . . . 67
evaluate.World . . . 68
Galgo . . . 69
clone.Galgo . . . 72
reInit.Galgo . . . 73
generateRandom.Galgo . . . 74
length.Galgo . . . 75
print.Galgo . . . 75
summary.Galgo . . . 76
refreshStats.Galgo . . . 77
evaluate.Galgo . . . 78
best.Galgo . . . 79
max.Galgo . . . 80
bestFitness.Galgo . . . 81
maxFitness.Galgo . . . 82
evolve.Galgo . . . 83
plot.Galgo . . . 84
BigBang . . . 85
blast.BigBang . . . 90
saveObject.BigBang . . . 91
print.BigBang . . . 92
summary.BigBang . . . 93

getFrequencies.BigBang . . . 94
activeChromosomeSet.BigBang . . . 95
plot.BigBang . . . 96
meanGeneration.BigBang . . . 100
geneFrequency.BigBang . . . 101
geneRankStability.BigBang . . . 102
meanFitness.BigBang . . . 104
classPrediction . . . 105
classPredictionMatrix.BigBang . . . 106
confusionMatrix.BigBang . . . 107
sensitivityClass.BigBang . . . 109
specificityClass.BigBang . . . 110
fitnessSplits.BigBang . . . 111
geneCoverage.BigBang . . . 113
forwardSelectionModels.BigBang . . . 114
heatmapModels.BigBang . . . 116
pcaModels.BigBang . . . 118
mergeBangs.BigBang . . . 119
addRandomSolutions.BigBang . . . 120
geneImportanceNetwork.BigBang . . . 121
assignParallelFile.BigBang . . . 122
loadParallelFiles.BigBang . . . 124
filterSolution.BigBang . . . 125
geneBackwardElimination . . . 126
robustGeneBackwardElimination . . . 127
modelSelection . . . 128
as.list.Object . . . 129
unObject . . . 130
reObject . . . 131
loadObject . . . 132
configBB.VarSel . . . 134
configBB.VarSelMisc . . . 139
runifInt . . . 143
ALL . . . 144
ALL.classes . . . 144
galgo.dist . . . 145
knn.C.predict . . . 146
knn.R.predict . . . 147
mlhd.C.predict . . . 147
mlhd.R.predict . . . 148
nearcent.C.predict . . . 149
nearcent.R.predict . . . 149
nnet.R.predict . . . 150
rpart.R.predict . . . 150
svm.C.predict . . . 151
svm.R.predict . . . 152

Index **153**

`Bag` *A list-like Object*

Description

Create a list of values. Lists inside an `Object` behave as by value (if the list is modified in a method, the original list is not updated). Therefore, `Bag` replace this behaviour extending `Object` and allowing to save reference-lists inside objects.

Usage

```
Bag(...)
```

Arguments

`...`	Values to store in the `Bag` object.

Class

Package: galgo
Class Bag

```
Object
~~|
~~+--Bag
```

Directly known subclasses:

public static class **Bag**
extends Object

Fields and Methods

Methods:

`length`	Gets the length of the object as its list version.
`print`	Prints the representation of the Bag object.
`summary`	Prints the representation of the Bag object.

Methods inherited from Object:
as.list, unObject, ,<-, [[, [[<-, as.character, attach, clone, detach, equals, extend, finalize, getFields, getInstanciationTime, getStaticInstance, hasField, hashCode, ll, load, objectSize, print, save

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

See Also

See also `list`().

Examples

```
    b <- Bag(a=1,b=2,c=3)
    b
    as.list(b)
    unObject(b)
```

`print.Bag` *Prints the representation of the Bag object*

Description

Prints the representation of the Bag object.

Usage

```
## S3 method for class 'Bag':
print(bag, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

See Also

For more information see `Bag`.

Examples

```
    b <- Bag(a=1,b=2,c=3)
    b
    summary(b)
    as.list(b)
    unObject(b)
```

`summary.Bag` *Prints the representation of the Bag object*

Description

Prints the representation of the Bag object.

Usage

```
## S3 method for class 'Bag':
summary(object, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

See Also

For more information see `Bag`. `*print()`.

Examples

```
  b <- Bag(a=1,b=2,c=3)
  b
  summary(b)
  as.list(b)
  unObject(b)
```

`length.Bag` *Gets the length of the object as its list version*

Description

Gets the length of the object as its list version.

Usage

```
## S3 method for class 'Bag':
length(bag, ...)
```

Value

Returns length of the object.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

See Also

For more information see `Bag`.

Examples

```
b <- Bag(a=1,b=2,c=3)
length(b)
```

`Gene` *The representation of a gene in a chromosome for genetic algorithms*

Description

Represents the behaviour of a gene in a chromosome for the genetic algorithm. The default properties are supposed to be used in the variable selection problem for microarray data. However, they can be used for any other problem. In addition, any other wanted variable can be added.

See references for Genetic Algorithms.

Usage

```
Gene(id=0, shape1=0, shape2=0, generateFunc=runifInt, ...)
```

Arguments

`id`	To identify the object.
`shape1`	Parameter for a distribution. Used to generate a random value for a gene (mean, minimum, alfa, etc).
`shape2`	Parameter for a distribution. Used to generate a random value for a gene (sd, maximum, beta, etc).
`generateFunc`	Function that generate a random value for a gene using the above shape parameters. This function would be used to get an initial value and to mutate a gene. The default is a random uniform integer with shape1 as minimum and shape2 as maximum (either inclusive). The parameters used in the call are object, n, shape1, and shape2. The random value generated is not saved. If future values depends on the previous, you must save it explicitly in the object.
`...`	Other user named values to include in the object.

Class

Package: galgo
Class Gene

```
Object
~~|
~~+--Gene
```

Directly known subclasses:

public static class **Gene**
extends Object

Fields and Methods

Methods:

`as.double`	Converts the gene parameters (shape1, shape2) to its numerical representation.
`as.matrix`	Converts the gene parameters (shape1, shape2) to matrix.
`generateRandom`	Generates a random value from the defined function.
`mutate`	Mutates a gene.
`newCollection`	Generates a list of cloned objects.
`newRandomCollection`	Generates a list of cloned objects and random values.
`print`	Prints the representation of a gene object.
`reInit`	Erases all internal values in order to re-use the object.
`summary`	Prints the representation of a gene object.

Methods inherited from Object:
as.list, unObject, ,<-, [[, [[<-, as.character, attach, clone, detach, equals, extend, finalize, getFields, getInstanciationTime, getStaticInstance, hasField, hashCode, ll, load, objectSize, print, save

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

`Chromosome`. `Niche`. `World`. `Galgo`. `BigBang`. `runIfInt`.

Examples

```
ge <- Gene(shape1=1, shape2=1000)
ge
```

`reInit.Gene` *Erases all internal values in order to re-use the object*

Description

Erases all internal values in order to re-use the object.

Usage

```
## S3 method for class 'Gene':
reInit(.O, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Gene`.

Examples

```
ge <- Gene(shape1=1, shape2=1000)
ge
reInit(ge)
```

`generateRandom.Gene` *Generates a random value from the defined function*

Description

Generates a random value from the defined function. The function used is stored in `generateFunc` value. The proper way to use this function is calling `*mutate()` method instead.

Usage

```
## S3 method for class 'Gene':
generateRandom(.O, n=1, ...)
```

Arguments

`n`	Number of random values.

Value

Returns random values.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Gene`. `*mutate()`.

Examples

```
ge <- Gene(shape1=1, shape2=1000)
ge
generateRandom(ge)
generateRandom(ge)
generateRandom(ge)

# generation that depends on initial random selection ==> "is it silly?"
ge$generateFunc = function(g, n=1, sh1, sh2) {
   if (is.null(g$value)) {
       g$value <- runif(n, sh1, sh2)
       g$value
   } else {
       g$value + runif(n, min=-10, max=10)
   }
}

generateRandom(ge)
generateRandom(ge)
generateRandom(ge)
```

`mutate.Gene` *Mutates a gene*

Description

Mutate a gene. This method is the proper way to call generateRandom method. For better description `*generateRandom()`.

Usage

```
## S3 method for class 'Gene':
mutate(.O, ...)
```

Arguments

`n`	Number of random values.

Value

Returns random values.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Gene`. `*generateRandom()`.

Examples

```
ge <- Gene(shape1=1, shape2=1000)
ge
mutate(ge)
mutate(ge)
mutate(ge)
```

`print.Gene` *Prints the representation of a gene object*

Description

Prints the representation of a gene object.

Usage

```
## S3 method for class 'Gene':
print(.O, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Gene`. `*summary()`

Examples

```
ge <- Gene(shape1=1, shape2=1000)
ge
print(ge)
```

as.double.Gene — *Converts the gene parameters (shape1, shape2) to its numerical representation*

Description

Converts the gene parameters (shape1, shape2) to its numerical representation.

Usage

```
## S3 method for class 'Gene':
as.double(x, ...)
```

Value

Returns a vector containig id, shape1, and shape2.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Gene`. `*genes()`.

Examples

```
ge <- Gene(shape1=1, shape2=1000)
ge
as.double(ge)
as.numeric(ge)
as.vector(ge) # returns NA
```

as.matrix.Gene — *Converts the gene parameters (shape1, shape2) to matrix*

Description

Converts the gene parameters (shape1, shape2) to matrix. It is suitable to bind rows for many genes (like in a chromosome).

Usage

```
## S3 method for class 'Gene':
as.matrix(x, ...)
```

Value

Returns a one-row matrix containig id, shape1, and shape2.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Gene`. `Chromosome`, `*genes()`, `*as.double()`.

Examples

```
ge <- Gene(shape1=1, shape2=1000)
ge
as.matrix(ge)
```

summary.Gene *Prints the representation of a gene object*

Description

Prints the representation of a gene object.

Usage

```
## S3 method for class 'Gene':
summary(object, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Gene`. `*print()`.

Examples

```
ge <- Gene(shape1=1, shape2=1000)
ge
print(ge)
summary(ge)
```

---

newCollection.Gene *Generates a list of cloned objects*

Description

Generates a list of cloned Gene Objects.

Usage

```
## S3 method for class 'Gene':
newCollection(.O, ...)
```

Arguments

n	Number of object clones.

Value

Returns a list with cloned objects. The names are build using class name and a consecutive number.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see Gene. *unObject(), *as.list(), *newRandomCollection(), Chromosome.

Examples

```
ge <- Gene(shape1=1, shape2=1000)
ge
print(ge)
# list of five new identical Gene objects (different id)
newCollection(ge, 5)
# list of two new identical Gene objects converted to a list using unObject
unObject(newCollection(ge,2))

# building chromosome from gene clones
# (perhaps for variable selection in microarray data)
cr <- Chromosome(genes=newCollection(ge, 5))
```

```
    cr
```

newRandomCollection.Gene

Generates a list of cloned objects and random values

Description

Creates a list of cloned objects with its internal values generated by random.

Usage

```
## S3 method for class 'Gene':
newRandomCollection(.O, ...)
```

Arguments

`n`	Number of object clones.

Details

For all cloned objects, `generateRandom` method is called. This has no effect for common `Gene` objects since the generated value is not stored there. However, this mechanism works equally well when it is needed to store values in `Gene`.

Value

Returns a list with cloned objects and random generated values.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Gene`. `*unObject()`, `*as.list()`, `*newCollection()`, `Chromosome`.

Examples

```
    ge <- Gene(shape1=1, shape2=1000)
    ge
    print(ge)
    # list of five new different Gene objects
    newRandomCollection(ge, 5)
    # list of two new different Gene objects converted to a list using unObject
    unObject(newRandomCollection(ge,2))
```

```
# building chromosome from gene clones
# (perhaps for variable selection in microarray data)
cr <- Chromosome(genes=newRandomCollection(ge, 5))
cr
```

Chromosome — *The representation of a set of genes for genetic algorithms*

Description

Represents a set of genes for the genetic algorithm. The chromosome contains all current values of each gene and will be evaluated using a "fitness" function similar to those defined by Goldberg. The fitness function normally depends on the `Galgo` object.

See references for Genetic Algorithms.

Usage

```
Chromosome(id=0,
        genes=list(),
        getValues=function(x, ...) unlist(lapply(x, ...)),
        decode=function(x) genes(x),
        values=list(),
        ...)
```

Arguments

`id`	A way to identify the object.
`genes`	A list of defined Gene objects composing the chromosome.
`getValues`	A function to be evaluated for every gene to obtain a value. In general, the result could be any object in a list. In particular, the default is a vector of current gene values.
`decode`	A function that converts the chromosome representation in real values. It is used mainly for output purposes and for frequency counting. It has no effect for variable selection in microarray data since the default `decode` is directly the gene value.
`values`	The specific initial values. If `value` is not specified, `getValues` function is ran to obtain initial values.
`...`	Other user named values to include in the object.

Class

Package: galgo
Class Chromosome

```
Object
~~|
~~+--Chromosome
```

Directly known subclasses:

public static class **Chromosome**
extends Object

Fields and Methods

Methods:

`as.double`	Converts the chromosome values (genes) to its numerical representation.
`clone`	Clones itself and its genes.
`decode`	Converts the gene values to user-readable values.
`generateRandom`	Generates random values for all genes in the chromosome.
`genes`	Converts the genes values to a numeric vector.
`length`	Gets the number of genes defined in the chromosome.
`mutate`	Mutates a chromosome in specific positions.
`newCollection`	Generates a list of chromosomes cloning the original chromosome object.
`newRandomCollection`	Creates a list of cloned chromosomes object with its internal values generated by randon
`print`	Prints the representation of the chromosome object.
`reInit`	Erases all internal values in order to re-use the object.
`summary`	Prints the representation of the chromosome object and all its genes.

Methods inherited from Object:
as.list, unObject, ,<-, [[, [[<-, as.character, attach, clone, detach, equals, extend, finalize, getFields, getInstanciationTime, getStaticInstance, hasField, hashCode, ll, load, objectSize, print, save

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

`Gene`. `Niche`. `World`. `Galgo`. `BigBang`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
```

`reInit.Chromosome` *Erases all internal values in order to re-use the object*

Description

Erases all internal values in order to re-use the object.

Usage

```
## S3 method for class 'Chromosome':
reInit(.O, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Chromosome`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
reInit(cr) # it does nothing in this case
cr
```

`clone.Chromosome` *Clones itself and its genes*

Description

Clones itself and its genes. Objects in S3 and this package are passed by reference and any “pointer” to it will affect the original object. Therefore, you must clone an object first in order to preserve the original values.

Usage

```
## S3 method for class 'Chromosome':
clone(.O, ...)
```

Value

Returns a new cloned object.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Chromosome`. `Object`

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
cr2 <- cr
generateRandom(cr2)
cr2
cr                    # cr and cr2 are the very same object
cr3 <- clone(cr2)
generateRandom(cr3)
cr3
cr2                   # now cr2 is different to cr3
cr                    # but cr2 is still the same than cr
```

`genes.Chromosome` *Converts the genes values to a numeric vector*

Description

Converts the genes values to a numeric vector.

Usage

```
## S3 method for class 'Chromosome':
genes(.O, ...)
```

Value

Returns a vector of gene values.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Chromosome`. `Gene`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
genes(cr)
# the output is the same, the print method uses genes method.
```

`generateRandom.Chromosome` *Generates random values for all genes in the chromosome*

Description

Updates gene values calling the method `generateRandom` to all its genes.

Usage

```
## S3 method for class 'Chromosome':
generateRandom(.O, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Chromosome`. `*unObject()`, `*as.list()`, `*newCollection()`, `*newRandomCollection()`

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
#
# from clone example
cr2 <- cr
generateRandom(cr2)
cr2
cr                     # cr and cr2 are the very same object
cr3 <- clone(cr2)
generateRandom(cr3)
cr3
cr2                    # now cr2 is different to cr3
cr                     # but cr2 is still the same than cr
```

`length.Chromosome` *Gets the number of genes defined in the chromosome*

Description

Gets the number of genes defined in the chromosome.

Usage

```
## S3 method for class 'Chromosome':
length(x, ...)
```

Value

A numeric value representing the number of genes in the chromosome.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Chromosome`. `*genes()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
length(cr)
```

`summary.Chromosome` *Prints the representation of the chromosome object and all its genes*

Description

Prints the representation of the chromosome object and all its genes.

Usage

```
## S3 method for class 'Chromosome':
summary(object, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Chromosome`. `*print()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
print(cr) # the same
summary(cr) # expanded view
```

`print.Chromosome` *Prints the representation of the chromosome object*

Description

Prints the representation of the chromosome object.

Usage

```
## S3 method for class 'Chromosome':
print(object, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Chromosome`. `*summary()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
print(cr) # the same
```

`as.double.Chromosome` *Converts the chromosome values (genes) to its numerical representation*

Description

Converts the chromosome values (genes) to its numerical representation.

Usage

```
## S3 method for class 'Chromosome':
as.double(x, ...)
```

Details

This function really calls `genes` method.

Value

Returns a vector containig the values for all genes in the chromosome.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Chromosome`. `*genes()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
as.double(cr) # the same
as.numeric(cr) # the same
as.vector(cr) # NA's is not the same
```

`mutate.Chromosome` *Mutates a chromosome in specific positions*

Description

Mutates a chromosome in specific positions.

Usage

```
## S3 method for class 'Chromosome':
mutate(ch, positions=sample(length(ch), 1), ...)
```

Arguments

`positions`	Vector of gene positions to be mutated. If `positions` is a vector of length 1 and the value is less than 1, it is considered as a probability; thus a `positions` vector is computed using the probability and the chromsome length.

Details

This method updates the gene values in the chromsome calling the method `mutate` for all genes indexed by `positions` vector.

Value

Returns the positions mutated.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Chromosome`. `*mutate()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
mutate(cr) # mutate 1 gene randomly
cr
mutate(cr,1:3) # mutate genes 1, 2, and 3
cr
```

`decode.Chromosome` *Converts the gene values to user-readable values*

Description

Converts the gene values to user-readable values.

Usage

```
## S3 method for class 'Chromosome':
decode(x, ...)
```

Details

This function really calls the function defined in the `$decode` variable in the `Chromosome` object.

Value

Returns the representation of the chromosome.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Chromosome`. `*genes()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
as.double(cr) # the same
as.numeric(cr) # the same
decode(cr) # the same
as.vector(cr) # NA's is not the same
```

newCollection.Chromosome

Generates a list of chromosomes cloning the original chromosome object

Description

Generates a list of chromosomes cloning the original chromosome object. It only use the generic newCollection method.

Usage

```
## S3 method for class 'Chromosome':
newCollection(.O, ...)
```

Arguments

`n`	Number of object clones.

Value

Returns a list with cloned objects. The names are build with the class and a consecutive number.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Chromosome`. `*unObject()`, `*as.list()`, `*newCollection()`, `*newRandomCollection()`, `Niche`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
# list of two new identical Chromosome objects (different id)
newCollection(cr, 2)
ni <- Niche(chromosomes = newCollection(cr, 2))
ni # same genes values, different objects
generateRandom(ni)
ni # different genes values

# creation and random generation at the same time
ni <- Niche(chromosomes = newRandomCollection(cr, 2))
ni
```

`newRandomCollection.Chromosome`

Creates a list of cloned chromosomes object with its internal values generated by random

Description

Creates a list of cloned chromosomes object with its internal values generated by random.

Usage

```
## S3 method for class 'Chromosome':
newRandomCollection(.O, ...)
```

Arguments

`n`	Number of object clones.

Details

For all cloned objects, `generateRandom` method is called to replace internal values.

Value

Returns a list with cloned objects and random generated values.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Chromosome`. `*unObject()`, `*as.list()`, `*newCollection()`, `Chromosome`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr

# creation and random generation at the same time
ni <- Niche(chromosomes = newRandomCollection(cr, 2))
ni
```

`Niche` *The representation of a set of chromosomes for genetic algorithms*

Description

Niche represents a set of chromosomes for the genetic algorithm. The niche can generate a progeny that may be more adapted to certains tasks (or enviroment, see Goldberg). To decide which chromosomes are more suitable to be chosen as “parents”, every chromosome in the niche is evaluated using a “fitness” function. The selected chromosomes are mated using crossover to produce diversity. Finally the chromosomes are mutated and the new progeny is ready for next generation.

The basic idea to generate a progeny is a random selection biased toward the best chromosomes (see Goldberg). We implented this idea as a weighted probability for a chromosome to be selected using the formula:

p = scale * max(0,fitness - mean * mean(fitness))ˆ power

where scale, mean and power are the properties of the niche (`offspringScaleFactor, offspringMeanFacto` `and offspringPowerFactor` respectively). The default values were selected to be reasonably bias when the variance in the fitness are both high (at early generations) and low (in late generatios).

The crossover mechanism needs to know the positions whose chromosomes can actually mate (`crossoverPoints`). The number of crossovers can be customized with `crossoverFunc` (`*crossover()`).

The elitism mechanism (`elitism` variable) are implemented replacing a random chromosome from the niche at the end of the progeny process (`*progeny()`).

The Niche object keeps a record of the number of generations, the maximum chromosome in the niche, and the best chromosome ever known (see `*best()` for an example).

The length of the niche is static. Nevertheless this behaviour (and any other) can be customised overwriting original methods (like progeny or crossover) methods. However, this is intend to be used only for experienced users.

The niche is considered a “closed population”, this means mating with chromosomes within the same niche. Migration mechanism uses niches to exchange chromosomes between them, which is implemented in `World` object (see `World`).

Usage

```
Niche(id=0,
        chromosomes=list(),
        offspringScaleFactor=1,
        offspringMeanFactor=0.85,
        offspringPowerFactor=2,
        crossoverPoints=0,
        mutationsFunc=function(.O) length(.O),
        crossoverFunc=function(.O) length(.O)/2,
        elitism=1,
        generation=0,
        fitness=0,
        maxFitness=0,
        bestFitness=0,
        maxChromosome=NULL,
        bestChromosome=NULL,
        ...)
```

Arguments

`id`	A way to identify the object.
`chromosomes`	A list of defined chromosomes composing the niche.
`offspringScaleFactor`	The `offspringScaleFactor` parameter. See description.
`offspringMeanFactor`	The `offspringMeanFactor` parameter. See description.
`offspringPowerFactor`	The `offspringPowerFactor` parameter. See description.
`crossoverPoints`	Specific positions at which the chromosomes can be mated. Should be from 2 to *minimum* possible length of any chromosome in the niche.
`mutationsFunc`	A function returning the final number of mutations in the niche. It receives the `Niche` object as parameter. To implement "probability of mutation" instead, add a variable like `pMutation` in the constructor and multiply by the length of the niche and the length of the chromosome in the function (`function(niche) niche$pMutation * length(niche) * length(niche$chromosomes[[1]])`).
`crossoverFunc`	A function returning the final number of crossovers in the niche. It receives the `Niche` object as parameter. To implement "probability of crossover" instead, add a variable like `pCrossOver` in the constructor and multiply by the length of the niche in the function. (`function(niche) niche$pCrossOver * length(niche)`).
`elitism`	Controls the elitism mechanism. Elitism is desired to find solutions quicker, but it may be a nuisance when it is trapped in strong attractors. Therefore, in general, it may be a probability. Furthermore, it can be a vector of probabilities where the index is controlled by generation. If the current generation is greather than the length of this vector, a cycled version is used (starting from the first value).
`fitness`	The current fitness. It should be 0 initially, but it is included for generalization.
`bestFitness`	The best fitness ever visited. It should be 0 initially. Included for generalization.
`maxFitness`	The maximum fitness from the current chromosomes. It should be 0 initially, but it is included for generalization.
`maxChromosome`	The chromosome whose fitness is maximum from the current chromosomes. It should be NULL initially, but it is included for generalization.
`bestChromosome`	The chromosome whose fitness is maximum visited ever. It should be NULL initially, but it is included for generalization.
`...`	Other user named values to include in the object (like pMutation, pCrossover or any other).

Class

Package: galgo
Class Niche

```
Object
~~|
~~+--Niche
```

Directly known subclasses:

public static class **Niche**
extends Object

Fields and Methods

Methods:

`as.double`	Converts the chromosome values (genes) to a vector.
`as.matrix`	Converts the chromosome values (genes) to a matrix.
`best`	Returns the best chromosome of the niche.
`bestFitness`	Returns the fitness of the best chromosome in the niche.
`clone`	Clones itself and its chromosomes.
`crossover`	Performs crossover between chromosomes of the niche.
`evaluate`	Evaluates the chromosome using a fitness function.
`generateRandom`	Generates random values for all genes contained in all chromosomes in the niche.
`getFitness`	Returns the fitness vector related to chromosomes.
`length`	Gets the number of chromosomes defined in the niche.
`max`	Returns the chromosome in the niche whose current fitness is maximum.
`maxFitness`	Returns the fitness of the maximum chromosome in the niche.
`mutate`	Mutates a niche calling mutate method for all chromosomes.
`newCollection`	Generates a list of cloned niches.
`newRandomCollection`	Creates a list of cloned niches with its internal values generated by random.
`offspring`	Overwrites the new niche selecting a new population from the best chromosomes.
`plot`	Plots information about niche object.
`print`	Prints the representation of a niche object.
`progeny`	Performs offspring, crossover, mutation, and elitism mechanism to generate the “evolved
`refreshStats`	Updates the internal values from the current population.
`reInit`	Erases all internal values in order to re-use the object.
`scaling`	Assigns a weight for every chromosome to be selected for the next generation.
`summary`	Prints the representation and statistics of the niche object.

Methods inherited from Object:
as.list, unObject, ,<-, [[, [[<-, as.character, attach, clone, detach, equals, extend, finalize, getFields, getInstanciationTime, getStaticInstance, hasField, hashCode, ll, load, objectSize, print, save

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

`Gene`, `Chromosome`, `World`, `Galgo`, `BigBang`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
ni

## in average, one of 10 genes can be mutated
mf <- function(niche) niche$pMutations * length(niche) * length(niche$chromosomes[[1]]
ni2 <- Niche(chromosomes=newRandomCollection(cr, 10),
       mutationsFunc=mf,
                pMutations=1/10)
ni2    # random initial niche
mutate(ni2) # returns the chromosomes indexes that were mutated
ni2    # mutated niche
```

`reInit.Niche` *Erases all internal values in order to re-use the object*

Description

Erases all internal values in order to re-use the object.

Usage

```
## S3 method for class 'Niche':
reInit(.O, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
ni
reInit(ni) # it does nothing in this case
```

---

clone.Niche *Clones itself and its chromosomes*

Description

Clones itself and its chromosomes. Objects in S3 and this package are passed by reference and any “pointer” to it will affect the original object. You must clone an object in order to conserve the original values.

Usage

```
## S3 method for class 'Niche':
clone(.O, ...)
```

Value

Returns a new cloned object.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`., `Object`

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
ni
ni2 <- ni
generateRandom(ni2)
ni2
ni                    # ni and ni2 are the very same object
ni3 <- clone(ni2)
generateRandom(ni3)
ni3
ni2                   # now ni2 is different to ni3
ni                    # but ni2 is still the same than ni
```

generateRandom.Niche

Generates random values for all genes contained in all chromosomes in the niche

Description

It only pass the message `generateRandom` to all its chromosomes.

Usage

```
## S3 method for class 'Niche':
generateRandom(.O, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`. `*unObject()`, `*as.list()`, `*newCollection()`, `*newRandomCollecti`

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
ni
ni2 <- ni
generateRandom(ni2)
ni2
ni                    # ni and ni2 are the very same object
ni3 <- clone(ni2)
generateRandom(ni3)
ni3
ni2                   # now cr2 is different to cr3
ni                    # but cr2 is still the same than cr
```

`length.Niche` *Gets the number of chromosomes defined in the niche*

Description

Gets the number of chromosomes defined in the niche.

Usage

```
## S3 method for class 'Niche':
length(x, ...)
```

Value

A numeric value representing the number of chromosomes in the niche.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`. `*length()`.

Examples

```
  cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
  ni <- Niche(chromosomes=newRandomCollection(cr, 10))
  ni
  length(ni)
```

`as.matrix.Niche` *Converts the chromosome values (genes) to a matrix*

Description

Converts the chromosome values (genes) to a matrix. Chromosomes are rows and genes are columns.

Usage

```
## S3 method for class 'Niche':
as.matrix(x, fitness=FALSE, ...)
```

Value

Returns a matrix containig the genes for all chromosomes. Chromosomes are rows and genes are columns.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`. `*print()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
ni
as.matrix(ni) # almost the same
as.matrix.Niche(ni, fitness=TRUE) # tricky undocumented version
```

`as.double.Niche` *Converts the chromosome values (genes) to a vector*

Description

Converts the chromosome values (genes) to a vector. It is a shortcut for `as.double(as.matrix(niche))`.

Usage

```
## S3 method for class 'Niche':
as.double(x, ...)
```

Value

Returns a vector containig the genes for all chromosomes in the niche. The order corresponds to the order inside chromosomes.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`.

Examples

```
cr
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
ni
as.double(ni)
as.double(as.matrix(ni)  # the same
as.numeric(ni) # the same
as.vector(ni) # NA is definitively NOT the same
```

`print.Niche` *Prints the representation of a niche object*

Description

Prints the representation of a niche object.

Usage

```
## S3 method for class 'Niche':
print(object, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`. `*summary()`.

Examples

```
cr
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
ni
print(ni) # the same
```

`summary.Niche` *Prints the representation and statistics of the niche object*

Description

Prints the representation and statistics of the niche object.

Usage

```
## S3 method for class 'Niche':
summary(object, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`. `*print()`.

Examples

```
  cr
  ni <- Niche(chromosomes=newRandomCollection(cr, 10))
  ni
  print(ni) # the same
  summary(ni) # expanded view
```

`refreshStats.Niche` *Updates the internal values from the current population*

Description

Updates the internal values from the current population. It updates maxFitness, maxChromosomes, bestFitness, and bestChromosomes.

Usage

```
## S3 method for class 'Niche':
refreshStats(.O, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`. `*summary()`.

Examples

```
cr
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
ni
summary(ni) # not so much
ni$fitness <- runif(10)  ## tricky fitness
refreshStats(ni)
summary(ni) # new updated values
ni$fitness <- runif(10)  ## new tricky fitness
refreshStats(ni)
summary(ni) # may be some new updated values
```

`plot.Niche` *Plots information about niche object*

Description

Plots information about niche object. See arguments for details.

Usage

```
## S3 method for class 'Niche':
plot(x,
        y,
        type=c("chromosomes", "fitness"),
        col=1,
        pch=19,
        horiz=TRUE,
        main="",
        xlab="",
        ylab="",
        chromosome=NULL,
        chromosome.chr="U",
        chromosomes=NULL,
        ...)
```

Arguments

`type`	The type of plot. `"chromosomes"` will plot the chromosomes in one axis and the genes in the other axis. The maximum chromosome is drawn with `"M"`, the best chromosome with `"B"` and the user chromosome with `"U"`. This plot give an overview of the population coverage. `"fitness"` plot the current fitness in vertical axis against chromosome index in horizontal.
`horiz`	Exchange the default choice of axis when `type="chromosomes"`.
`main,xlab, ylab,col,pch`	`Niche` defaults for common plot parameters. Their usage overwrite the default value. `col` controls the color for chromosomes
`chromosome`	An additional chromosome for comparison.
`chromosome.chr`	Explicit character for additional chromosome.
`chromosomes`	Use the specific list of chromosomes instead of the original `Niche` chromosomes.
`...`	Other user named values to include in the object.

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`.

Examples

```
cr
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
ni
plot(ni, main="My Niche")
plot(ni, type="fitness")
```

newCollection.Niche

Generates a list of cloned niches

Description

Generates a list of cloned niches. It only use the generic newCollection method.

Usage

```
## S3 method for class 'Niche':
newCollection(.O, ...)
```

Arguments

`n`	Number of object clones.

Value

Returns a list with cloned objects. The names are build with the class and a consecutive number.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`. `*unObject()`, `*as.list()`, `*newCollection()`, `*newRandomCollecti`

Examples

```
cr
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
ni
newCollection(ni, 2)                  # list of two new identical Niche objects
newRandomCollection(ni, 2)            # list of two new different Niche objects
```

`newRandomCollection.Niche` *Creates a list of cloned niches with its internal values generated by random*

Description

Creates a list of cloned niches with its internal values generated by random.

Usage

```
## S3 method for class 'Niche':
newRandomCollection(.O, ...)
```

Arguments

`n`	Number of object clones.

Details

Creates a list of cloned niches with its internal values generated by random. For all cloned objects, `generateRandom` method is called.

Value

Returns a list with cloned objects and random generated values.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`. `*unObject()`, `*as.list()`, `*newCollection()`, `Chromosome`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr

ni <- Niche(chromosomes = newRandomCollection(cr, 2))
ni

wo <- World(niches = newRandomCollection(ni, 2))
wo
```

mutate.Niche *Mutates a niche calling mutate method for all chromosomes*

Description

Mutates a niche calling mutate method for all chromosomes.

Usage

```
## S3 method for class 'Niche':
mutate(ni, n=(ni$mutationsFunc)(ni), ...)
```

Arguments

`n`	Number of chromosomes to mutate. The default is the result of calling `mutationsFunc`.

Details

This method update the gene values for random chromsomes. The number of chromosomes to mutate is normally obtained calling `mutationFunc`.

Value

This methods returns the chromosome indexes mutated.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`. `mutate.Chromosome`(), `mutate.Gene`().

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
ni
mutate(ni, 3)
ni
```

`best.Niche` *Returns the best chromosome of the niche*

Description

Returns the best chromosome of the niche.

Usage

```
## S3 method for class 'Niche':
best(ni, ...)
```

Value

Returns the best chromosome ever visited in the niche.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`. `*bestFitness`(), `*max`(), `*maxFitness`().

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
ni$fitness <- 1:10/10 # tricky fitness
refreshStats(ni)      # compute best and max chromosomes
summary(ni)
best(ni)
ni$bestChromosome     # the same
max(ni)               # the same in this case
bestFitness(ni)       # 1
maxtFitness(ni)       # 1
```

max.Niche *Returns the chromosome in the niche whose current fitness is maximum*

Description

Returns the chromosome in the niche whose current fitness is maximum.

Usage

```
## S3 method for class 'Niche':
max(ni, ...)
```

Value

Returns the chromosome in the niche whose current fitness is maximum.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`. `*best()` `*bestFitness()`, `*maxFitness()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
ni$fitness <- 1:10/10 # tricky fitness
refreshStats(ni)      # compute best and max chromosomes
summary(ni)
best(ni)
ni$bestChromosome     # the same
max(ni)               # the same in this case
bestFitness(ni)       # 1
maxtFitness(ni)       # 1
```

`bestFitness.Niche` *Returns the fitness of the best chromosome in the niche*

Description

Returns the fitness of the best chromosome in the niche.

Usage

```
## S3 method for class 'Niche':
bestFitness(ni, ...)
```

Value

Returns the fitness of the best chromosome.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`. `*best()`, `*max()`, `*maxFitness()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
ni$fitness <- 1:10/10 # tricky fitness
refreshStats(ni)      # compute best and max chromosomes
summary(ni)
best(ni)
ni$bestChromosome     # the same
max(ni)               # the same in this case
bestFitness(ni)       # 1
maxtFitness(ni)       # 1
```

maxFitness.Niche *Returns the fitness of the maximum chromosome in the niche*

Description

Returns the fitness of the maximum chromosome in the niche.

Usage

```
## S3 method for class 'Niche':
maxFitness(ni, ...)
```

Value

Returns the fitness of the maximum chromosome.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`. `*best()`, `*max()`, `*bestFitness()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
ni$fitness <- 1:10/10 # tricky fitness
refreshStats(ni)      # compute best and max chromosomes
summary(ni)
best(ni)
ni$bestChromosome     # the same
max(ni)               # the same in this case
bestFitness(ni)       # 1
maxtFitness(ni)       # 1
```

`getFitness.Niche` *Returns the fitness vector related to chromosomes*

Description

Returns the fitness vector related to chromosomes.

Usage

```
## S3 method for class 'Niche':
getFitness(ni, ...)
```

Value

Returns the fitness of each chromosome in the niche.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`. `*best()`, `*max()`, `*bestFitness()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
ni$fitness <- 1:10/10 # tricky fitness, instead of evaluating in a Galgo object
refreshStats(ni)      # compute best and max chromosomes
summary(ni)
getFitness(ni)
```

`crossover.Niche` *Performs crossover between chromosomes of the niche*

Description

Perform crossover between chromosomes of the niche. This method is called inside `progeny` method.

Usage

```
## S3 method for class 'Niche':
crossover(ni, n=(ni$crossoverFunc)(ni), ...)
```

Arguments

n	Number of crossover to perform. The default is obtained calling `crossoverFunc`.

Value

Returns the “males” and “females” chromosomes used to crossover.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`. `*progeny()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
ni
crossover(ni)
ni
```

`offspring.Niche` *Overwrites the new niche selecting a new population from the best chromosomes*

Description

Overwrites the new niche selecting from the best chromosomes. This method is called in the `progeny` method. For more information see `Niche`. for complete description.

Usage

```
## S3 method for class 'Niche':
offspring(ni, ...)
```

Details

The basic idea to generate a progeny is a selection biased toward the best chromosomes (see Goldberg). We implented this idea as a weighted probability (`*scaling()`).

`offspring` is part of `progeny` method.

For related details For more information see `Niche`.

Value

Returns the selected chromosomes.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`. `*progeny()`, `*scaling()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
ni
ni$fitness <- 1:10/10 # tricky fitness, only for showing purposes
scaling(ni)
offspring(ni)
ni
```

`progeny.Niche` *Performs offspring, crossover, mutation, and elitism mechanism to generate the “evolved” niche*

Description

Performs offspring, crossover, mutation, and elitism mechanism to generate the “evolved” niche.

Usage

```
## S3 method for class 'Niche':
progeny(ni, immigration=NULL, ...)
```

Arguments

`immigration`	Chromosomes wanted to immigrate (replacing) in the niche.

Details

The basic idea to generate a progeny is a random selection biased toward the best chromosomes (see Goldberg). We implented this idea as a weighted probability for a chromosome to be selected using the formula:

p = scale * max(0,fitness - mean * mean(fitness))ˆ power

where scale, mean and power are the properties of the niche (`offspringScaleFactor, offspringMeanFacto` `and offspringPowerFactor` respectively). The default values were selected to be reasonably bias when the variance in the fitness are both high (at early generations) and low (in late generatios).

`offspring` is part of `progeny` method.

For related details For more information see `Niche`.

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`. `*offspring()`, `*crossover()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
ni
ni$fitness <- 1:10/10 # tricky fitness, only for showing purposes
progeny(ni)
ni
```

`evaluate.Niche` *Evaluates the chromosome using a fitness function*

Description

Evaluate the chromosome using a fitness function. The result of this evaluation is treated as the “fitness” value as defined by Goldberg (see references). The `Galgo` object call this method and store the resulted value in order to decide which chromosomes are better choices to be part of the next generation. The “fitness function” should returns a numeric value scaled from 0 to 1. As close to 1 as better chance it have to be part of the next generation.

Usage

```
## S3 method for class 'Niche':
evaluate(.O, fn, parent, ...)
```

Arguments

`fn`	The "fitness" function to be called to evaluate all chromosomes. It should follow the format `function(obj, parent) { ... }`.
`parent`	The original object calling for the evaluation. This is passed when the function is sensitive to data stored in parent object. Commonly it is a `BigBang` object (perhaps `Galgo` instead).

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Niche`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
ni
fn <- function(chr, parent) { sd(as.double(chr))/mean(as.double(chr)) }
evaluate(ni, fn, parent)
getFitness(ni) ## see results
summary(ni)
```

`World` *The representation of a set of niches with migration for genetic algorithms*

Description

Represents a set of nices for the genetic algorithm. Because the niches are "closed populations", it is sometimes needed exchange information bewteen niches (or "islands"). The `World` object implements the exchange of chromosomes between niches, and to be compatible, it also implements the needed methods than an usual niche but considering the immigration property. Thus, the `Galgo` object can receive a list of Niches, a list of Worlds, or a list of any mixture of them.

Usage

```
World(id=0,
        niches=list(),
        immigration=0,
        maxFitness=0,
        bestFitness=0,
        maxChromosome=NULL,
        bestChromosome=NULL,
        generation=0,
        ...)
```

Arguments

`id`	A way to identify the object.
`niches`	A list of defined niches composing the world. However, it can be a list containing even `World` objects.
`immigration`	It can be `NULL`, a `function`, or a `vector`. When it is `NULL` immigration is disabled. When it is a `function` it is evaluated using the same `World` object as parameter, the result should be a numeric value. When the length of `immigration` is greather than 1 a cycled version is used depending on the `generation`. If the resulted or selected numeric value is greather than 1 it is interpreted as the number of chromosomes to migrate, otherwise it is assumed to be a probability to migrate one chromosome. The final `I` best chromosomes to migrate apply to all niches.
`bestFitness`	The best fitness ever visited.
`maxFitness`	The maximum fitness from the current chromosomes. It should be 0 initially, but it is included for generalization.
`maxChromosome`	The chromosome whose fitness is maximum from the current chromosomes. It should be NULL initially, but it is included for generalization.
`bestChromosome`	The chromosome whose fitness is maximum visited ever. It should be NULL initially, but it is included for generalization.
`...`	Other user named values to include in the object (like pMutation, pCrossover or any other).

Class

Package: galgo
Class World

```
Object
~~|
~~+--World
```

Directly known subclasses:

public static class **World**
extends Object

Fields and Methods

Methods:

`best`	Returns the best chromosome.
`bestFitness`	Returns the fitness of the best chromosome.
`clone`	Clones itself and its niches.
`evaluate`	Evaluate all niches with a fitness function.
`generateRandom`	Generates random values for all niches in the world.
`length`	Gets the number of niches defined in the world.
`max`	Returns the chromosome whose current fitness is maximum.
`maxFitness`	Returns the fitness of the maximum chromosome.
`newCollection`	Generates a list cloning an object.
`newRandomCollection`	Creates a list of cloned object with its internal values generated by random.
`plot`	Plots information about world object.
`print`	Prints the representation of a world object.
`progeny`	Calls progeny method to all niches in the world object.
`refreshStats`	Updates the internal statistics from the current population.
`reInit`	Erases all internal values in order to re-use the world object.
`summary`	Prints the representation and statistics of the world object.

Methods inherited from Object:
as.list, unObject, ,<-, [[, [[<-, as.character, attach, clone, detach, equals, extend, finalize, getFields, getInstanciationTime, getStaticInstance, hasField, hashCode, ll, load, objectSize, print, save

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

`Gene`, `Chromosome`, `Niche`, `Galgo`, `BigBang`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))
wo

progeny(wo) # returns the chromosomes indexes that were mutated
```

reInit.World *Erases all internal values in order to re-use the world object*

Description

Erase all internal values in order to re-use the world object.

Usage

```
## S3 method for class 'World':
reInit(.O, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see World.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))
wo
reInit(wo) # it does nothing in this case
```

clone.World *Clones itself and its niches*

Description

Clone itself and its niches. Objects in S3 and this package are passed by reference and any “pointer” to it will affect the original object. You must clone an object in order to conserve the original values.

Usage

```
## S3 method for class 'World':
clone(.O, ...)
```

Value

Returns a new cloned object.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `World`. `Object`

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))
wo
wo2 <- wo
generateRandom(wo2)
wo2
wo                      # wo and wo2 are the very same object
wo3 <- clone(wo2)
generateRandom(wo3)
wo3
wo2                     # now wo2 is different to wo3
wo                      # but wo2 is still the same than wo
```

`generateRandom.World` *Generates random values for all niches in the world*

Description

Generates random values for all niches in the world. It only pass the message `generateRandom` to all its niches.

Usage

```
## S3 method for class 'World':
generateRandom(.O, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `World`. `*unObject()`, `*as.list()`, `*newCollection()`, `*newRandomCollecti`

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))
wo
wo2 <- wo
generateRandom(wo2)
wo2
wo                     # wo and wo2 are the very same object
wo3 <- clone(wo2)
generateRandom(wo3)
wo3
wo2                    # now wo2 is different to wo3
wo                     # but wo2 is still the same than wo
```

`length.World` *Gets the number of niches defined in the world*

Description

Gets the number of niches defined in the world.

Usage

```
## S3 method for class 'World':
length(x, ...)
```

Value

A numeric value representing the number of niches in the world.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `World`. `*length.Niche()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))
wo
length(wo)
```

`print.World` *Prints the representation of a world object*

Description

Prints the representation of a world object.

Usage

```
## S3 method for class 'World':
print(object, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `World`. `*summary()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))
wo
print(wo) # the same
```

summary.World *Prints the representation and statistics of the world object*

Description

Prints the representation and statistics of the world object.

Usage

```
## S3 method for class 'World':
summary(object, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see World. *print().

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))
wo
summary(wo)
```

refreshStats.World *Updates the internal statistics from the current population*

Description

Update the internal statistics from the current population. It updates maxFitness, maxChromosomes, bestFitness, and bestChromosomes.

Usage

```
## S3 method for class 'World':
refreshStats(.O, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `World`. `*summary()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
ni$fitness <- runif(10)  ## tricky fitness
ni
wo <- World(niches=newRandomCollection(ni,2))
refreshStats(wo)
summary(wo)
```

`plot.World` *Plots information about world object*

Description

. See arguments for details.

Usage

```
## S3 method for class 'World':
plot(x,
        type=c("chromosomes", "fitness"),
        horiz=TRUE,
        pch=19,
        col=1,
        main="",
        xlab="",
        ylab="",
        chromosome=NULL,
        chromosome.chr="U",
        ...)
```

Arguments

`type`	The type of plot. `"chromosomes"` will plot the chromosomes in one axis and the genes in the other axis. The maximum chromosome is drawn with `"M"`, the best chromosome with `"B"` and the user chromosome with `"U"`. This plot give an overview of the population coverage. `"fitness"` plot the current fitness in vertical axis against chromosome index in horizontal.
`horiz`	Exchange the default choice of axis when `type="chromosomes"`.
`main,xlab, ylab,col,pch`	`World` defaults for common plot parameters. Their usage overwrite the default value. `col` controls the color for chromosomes
`chromosome`	An additional chromosome for comparison.
`chromosome.chr`	Explicit character for additional chromosome.
`chromosomes`	Use the specific list of chromosomes instead of the original `Niche` chromosomes.
`...`	Other user named values to include in the object.

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `World`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
ni$fitness <- runif(10)  ## tricky fitness
ni
wo <- World(niches=newRandomCollection(ni,2))
refreshStats(wo)
plot(wo, main="My Niche")
plot(wo, type="fitness")
```

newCollection.World

Generates a list cloning an object

Description

Generates a list cloning an object. It only use the generic newCollection method.

Usage

```
## S3 method for class 'World':
newCollection(.O, ...)
```

Arguments

n	Number of object clones.

Value

Returns a list with cloned objects. The names are build with the class and a consecutive number.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `World`. `*unObject()`, `*as.list()`, `*newCollection()`, `*newRandomCollecti`

Examples

```
  cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
  ni <- Niche(chromosomes=newRandomCollection(cr, 10))
  ni$fitness <- runif(10)  ## tricky fitness
  ni
  wo <- World(niches=newRandomCollection(ni,2))
  newCollection(wo, 2)                  # list of two new identical World objects
  newRandomCollection(wo, 2)            # list of two new different World objects
```

newRandomCollection.World

Creates a list of cloned object with its internal values generated by random

Description

Creates a list of cloned object with its internal values generated by random.

Usage

```
## S3 method for class 'World':
newRandomCollection(.O, ...)
```

Arguments

`n`	Number of object clones.

Details

. For all cloned objects, `generateRandom` method is called.

Value

Returns a list with cloned objects and random generated values.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `World`. `*unObject()`, `*as.list()`, `*newCollection()`, `Niche`.

Examples

```
    cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
    cr

    ni <- Niche(chromosomes = newRandomCollection(cr, 2))
    ni

    wo <- World(niches = newRandomCollection(ni, 2))
    wo
```

`best.World` *Returns the best chromosome*

Description

Returns the best chromosome.

Usage

```
## S3 method for class 'World':
best(.O, ...)
```

Value

Returns the best chromosome ever visited.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `World`. `*bestFitness()`. `*max()`. `*maxFitness()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
ni$fitness <- runif(10)  ## tricky fitness
ni
wo <- World(niches=newRandomCollection(ni,2))
refreshStats(wo)
best(wo)
max(wo)
bestFitness(wo)
maxFitness(wo)
```

`max.World` *Returns the chromosome whose current fitness is maximum*

Description

Returns the chromosome whose current fitness is maximum.

Usage

```
## S3 method for class 'World':
max(.O, ...)
```

Value

Returns the chromosome whose current fitness is maximum.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `World`. `*best()`. `*bestFitness()`. `*maxFitness()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
ni$fitness <- runif(10)  ## tricky fitness
ni
wo <- World(niches=newRandomCollection(ni,2))
refreshStats(wo)
best(wo)
max(wo)
bestFitness(wo)
maxFitness(wo)
```

`bestFitness.World` *Returns the fitness of the best chromosome*

Description

Returns the fitness of the best chromosome.

Usage

```
## S3 method for class 'World':
bestFitness(.O, ...)
```

Value

Returns the fitness of the best chromosome.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `World`. `*best()`, `*max()`, `*maxFitness()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
ni$fitness <- runif(10)  ## tricky fitness
ni
wo <- World(niches=newRandomCollection(ni,2))
refreshStats(wo)
best(wo)
max(wo)
bestFitness(wo)
maxFitness(wo)
```

maxFitness.World *Returns the fitness of the maximum chromosome*

Description

Returns the fitness of the maximum chromosome.

Usage

```
## S3 method for class 'World':
maxFitness(.O, ...)
```

Value

Returns the fitness of the maximum chromosome from the current population.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `World`. `*best()`, `*max()`, `*bestFitness()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
ni$fitness <- runif(10)  ## tricky fitness
ni
wo <- World(niches=newRandomCollection(ni,2))
refreshStats(wo)
best(wo)
max(wo)
bestFitness(wo)
maxFitness(wo)
```

`progeny.World` *Calls progeny method to all niches in the world object*

Description

Calls progeny method to all niches in the world object.

Usage

```
## S3 method for class 'World':
progeny(.O, immigration=NULL, ...)
```

Arguments

`immigration`	Chromosomes wanted to immigrate (replacing) in the niche.

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `World`. `Niche`, `progeny.Niche`(), `offspring.Niche`().

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
ni$fitness <- runif(10)  ## tricky fitness
ni
wo <- World(niches=newRandomCollection(ni,2))
progeny(wo)
progeny(wo,2)
```

`evaluate.World` *Evaluate all niches with a fitness function*

Description

Evaluate all niches with a fitness function. The result of this evaluation is treated as the “fitness” value as defined by Goldberg (see references). The `Galgo` object call this method and store the resulted value in order to decide which chromosomes are better choices to be part of the next generation. The “fitness function” should returns a numeric value scaled from 0 to 1. As close to 1 as better chance it have to be part of the next generation.

Usage

```
## S3 method for class 'World':
evaluate(.O, fn, parent, ...)
```

Arguments

`fn`	The “fitness” function to be called to evaluate all niches. It should follow the format `function(obj, parent) { ... }`
`parent`	The original object calling for the evaluation. This is passed when the function is sensitive to data stored in parent object. Commonly it is a `BigBang` object (perhaps `Galgo` instead).

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `World`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
cr
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
ni
fn <- function(chr, parent) { sd(as.double(chr))/mean(as.double(chr)) }
evaluate(ni, fn, parent)
getFitness(ni) ## see results
summary(ni)
wo <- World(niches=newRandomCollection(ni,2))
```

```
evaluate(wo, fn, parent)
```

`Galgo` *The representation of a Genetic Algorithm*

Description

Represents a genetic algorithm (GA) itself. The basic GA uses at least one population of chromosomes, a "fitness" function, and a stopping rule (see references).

The Galgo object is not limited to a single population, it implements a list of populations where any element in the list can be either a `Niche` object or a `World` object. Nervertheless, any user-defined object that implements `evolve, progeny, best, max, bestFitness, and maxFitness` methods can be part of the `populations` list.

The "fitness" function is by far the most important part of a GA, it evaluates a `Chromosome` to determine how good the chromosome is respect to a given goal. The function can be sensitive to data stored in `.GlobalEnv` or any other object (see `*evaluate()` for further details). For this package and in the case of the microarray, we have included several fitness functions to classify samples using different methods. However, it is not limited for a classification problem for microarray data, because you can create any fitness function in any given context.

The stopping rule has three options. First, it is simply a desired fitness value implemented as a numeric `fitnessGoal`, and If the maximum fitness value of a population is equal or higher than `fitnessGoal` the GA ends. Second, `maxGenerations` determine the maximum number of generations a GA can evolve. The current generation is increased after evaluating the fitness function to the entire population list. Thus, if the current generation reach `maxGenerations` the GA stops. Third, if the result of the user-defined `callBackFunc` is `NA` the GA stops. In addition, you can always break any R program using `Ctrl-C` (or `Esc` in Windows).

When the GA ends many values are used for futher analysis. Examples are the best chromosome (`best` method), its fitness (`bestFitness` method), the final generation (`generation` variable), the evolution of the maximum fitness (`maxFitnesses` list variable), the maximum chromosome in each generation (`maxChromosome` list variable), and the elapsed time (`elapsedTime` variable). Moreover, flags like `goalScored`, `userCancelled`, and `running` are available.

Usage

```
Galgo(id=0,
        populations=list(),
        fitnessFunc=function(...) 1,
        goalFitness=0.9,
        minGenerations=1,
        maxGenerations=100,
        addGenerations=0,
        verbose=20,
        callBackFunc=function(...) 1,
        data=NULL,
        gcCall=0,
        savePopulations=FALSE,
        maxFitnesses=c(),
        maxFitness=0,
        maxChromosomes=list(),
        maxChromosome=NULL,
        bestFitness=0,
        bestChromosome=NULL,
        savedPopulations=list(),
        generation=0,
        elapsedTime=0,
        initialTime=0,
        userCancelled=FALSE,
        goalScored=FALSE,
        running=FALSE,
        ...)
```

Arguments

`id`	A way to identify the object.
`populations`	A list of populations of any class `World`, `Niche`, or user-defined population.
`fitnessFunc`	The function that will be evaluate any chromosome in the populations. This function should receive two parameteres, the `Chromosome` object and the `parent` object (defined as a parameter as well). The `parent` object is commonly a object of class `BigBang` when used combined. Theoretically, the fitness function may return a numeric non-negative finite value, but commonly in practice these values are limited from `0` to `1`. The `offspring` factors in class `Niche` where established using the `0-1` range assumption.
`goalFitness`	The desired fitness. The GA will evolve until it reach this value or any other stopping rule is met. See description section.
`minGenerations`	The minimum number of generations. A GA evolution will not ends before this generation number even that `fitnessGoal` has been reach.
`maxGenerations`	The maximum number of generations that the GA could evolve.
`addGenerations`	The number of generations to over-evolve once that `goalFitness` has been met. Some solutions reach the goal from a large "jump" (or quasi-random mutation) and some other from "plateau". `addGenerations` helps to ensure the solutions has been "matured" at least that number of generations.
`verbose`	Instruct the GA to display the general information about the evolution. When `verbose==1` this information is printed every generation. In general every `verbose` number of generation would produce a line of output. Of course if `verbose==0` would not display a thing at all.
`callBackFunc`	A user-function to be called after every generation. It should receive the `Galgo` object itself. If the result is `NA` the GA ends. For instance, if `callBackFunc` is `plot` the trace of all generations is nicely viewed in a plot; however, in long runs it can consume time and memory.
`data`	Any user-data can be stored in this variable (but it is not limited to `data`, the user can insert any other like `myData`, `mama.mia` or `whatever` in the `...` argument).
`...`	Other user named values to include in the object (like pMutation, pCrossover or any other).

Class

Package: galgo
Class Galgo

```
Object
~~|
~~+--Galgo
```

Directly known subclasses:

public static class **Galgo**
extends Object

Fields and Methods

Methods:

`best`	Returns the best chromosome.
`bestFitness`	Returns the fitness of the best chromosome.
`clone`	Clones itself and all its objects.
`evaluate`	Evaluates all chromosomes with a fitness function.
`evolve`	Evolves the chromosomes populations of a Galgo (Genetic Algorithm).
`generateRandom`	Generates random values for all populations in the Galgo object.
`length`	Gets the number of populations defined in the Galgo object.
`max`	Returns the chromosome whose current fitness is maximum.
`maxFitness`	Returns the fitness of the maximum chromosome.
`plot`	Plots information about the Galgo object.
`print`	Prints the representation of a Galgo object.
`refreshStats`	Updates the internal values from the current populations.
`reInit`	Erases all internal values in order to re-use the object.
`summary`	Prints the representation and statistics of the galgo object.

Methods inherited from Object:
as.list, unObject, ,<-, [[, [[<-, as.character, attach, clone, detach, equals, extend, finalize, getFields, getInstanciationTime, getStaticInstance, hasField, hashCode, ll, load, objectSize, print, save

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

`Gene`, `Chromosome`, `Niche`, `World`, `BigBang`, `configBB.VarSel`(), `configBB.VarSelMisc`().

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))
ga <- Galgo(populations=list(wo), goalFitness = 0.75, callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))
ga
evolve(ga)

# missing a classification example
```

`clone.Galgo` *Clones itself and all its objects*

Description

Clone itself and all its objects.

Objects in S3 and this package are passed by reference and any “pointer” to it will affect the original object. You must clone an object in order to conserve the original values.

Usage

```
## S3 method for class 'Galgo':
clone(.O, ...)
```

Value

Returns a new cloned object.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Galgo`. `Object`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))

ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75, callBackFunc=plo
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))
ga
ga2 <- clone(ga)
```

```
evolve(ga)
evolve(ga2)
ga3 <- clone(ga)
evolve(ga3) # really nothing, everything already done
```

`reInit.Galgo` *Erases all internal values in order to re-use the object*

Description

Erases all internal values in order to re-use the object.

Usage

```
## S3 method for class 'Galgo':
reInit(.O, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Galgo`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))

ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
            callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))
ga
evolve(ga)
evolve(ga)  ## nothing
reInit(ga)
generateRandom(ga)
evolve(ga)  ## here we go again
```

generateRandom.Galgo

Generates random values for all populations in the Galgo object

Description

Generates random values for all populations in the Galgo object. It only pass the message `generateRandom` to all its populations.

Usage

```
## S3 method for class 'Galgo':
generateRandom(.O, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Galgo`. `*unObject()`, `*as.list()`, `*newCollection()`, `*newRandomCollecti`

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))

ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
            callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))
ga
evolve(ga)
evolve(ga)  ## nothing
reInit(ga)
generateRandom(ga)
evolve(ga)  ## here we go again
```

`length.Galgo` *Gets the number of populations defined in the Galgo object*

Description

Gets the number of populations defined in the Galgo object.

Usage

```
## S3 method for class 'Galgo':
length(x, ...)
```

Value

A numeric value representing the number of populations in the Galgo object.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Galgo`. `*length.Niche()`, `*length.World()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))

ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
            callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))
length(ga) ## 1
```

`print.Galgo` *Prints the representation of a Galgo object*

Description

Prints the representation of a Galgo object.

Usage

```
## S3 method for class 'Galgo':
print(object, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Galgo`. `*summary()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))

ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
            callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))
ga
evolve(ga)
ga
print(ga) # the same
```

`summary.Galgo` *Prints the representation and statistics of the galgo object*

Description

Prints the representation and statistics of the galgo object.

Usage

```
## S3 method for class 'Galgo':
summary(object, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Galgo`. `*print()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))

ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
            callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))
ga
summary(ga)
evolve(ga)
summary(ga)
```

`refreshStats.Galgo` *Updates the internal values from the current populations*

Description

Updates the internal values from the current populations. It updates maxFitness, maxChromosomes, bestFitness, and bestChromosomes. Called internally in `evolve` method.

Usage

```
## S3 method for class 'Galgo':
refreshStats(.O, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Galgo`. `*summary()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))

ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
            callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))
ga
summary(ga)
evaluate(ga) # manual evaluation
ga
refreshStats(ga)
ga           # updated values
summary(ga)  # but chromosomes have not been "evolved"
```

`evaluate.Galgo` *Evaluates all chromosomes with a fitness function*

Description

Evaluates all chromosomes with a fitness function. The result of this evaluation is treated as the “fitness” value (as defined by Goldberg, see references). The `Galgo` object call this method and store the returned value asociated with each chromosome in order to decide which chromosomes are the best choices to be part of the next generation. The “fitness function” commonly returns a numeric value scaled from 0 to 1 (but not always, For more information see `Galgo`.). As close to 1 as better chance it could be part of the next generation.

Usage

```
## S3 method for class 'Galgo':
evaluate(.O, fn=.O$fitnessFunc, parent=NULL, ...)
```

Arguments

`fn`	The “fitness” function to be called to evaluate all chromosomes. It should follow the format `function(obj, parent) { ... }`. The default is to use the function specified in the `Galgo` object.
`parent`	The original object calling for the evaluation. This is passed when the function is sensitive to data stored in parent object. Commonly it is a `BigBang` object (perhaps `Galgo` instead).

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Galgo`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))

ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
            callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))
ga
summary(ga)
evaluate(ga) # manual evaluation
ga
refreshStats(ga)
ga           # updated values
summary(ga)  # but chromosomes have not been "evolved"

evolve(ga)
# the usual evaluation of fitness function is inside evolve method
```

`best.Galgo` *Returns the best chromosome*

Description

Returns the best chromosome.

Usage

```
## S3 method for class 'Galgo':
best(.O, ...)
```

Value

Returns the best chromosome ever visited.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Galgo`. `*bestFitness()`, `*max()`, `*maxFitness()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))
ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
            callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))
evolve(ga)
best(ga)
max(ga)                        # the Maximum chromosome may be different to the best
bestFitness(ga)
maxFitness(ga)
```

`max.Galgo` *Returns the chromosome whose current fitness is maximum*

Description

Returns the chromosome whose current fitness is maximum.

Usage

```
## S3 method for class 'Galgo':
max(.O, ...)
```

Value

Returns the chromosome whose current fitness is maximum.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Galgo`. `*best()`, `*bestFitness()`, `*maxFitness()`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes = newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))
ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
            callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))
evolve(ga)
best(ga)
max(ga)                      # the Maximum chromosome may be different to the best
bestFitness(ga)
maxFitness(ga)
```

---

bestFitness.Galgo *Returns the fitness of the best chromosome*

Description

Returns the fitness of the best chromosome.

Usage

```
## S3 method for class 'Galgo':
bestFitness(.O, ...)
```

Value

Returns the fitness of the best chromosome.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see Galgo. *best(), *max(), *maxFitness().

Examples

```
wo <- World(niches=newRandomCollection(Niche(chromosomes=newRandomCollection(
      Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5)), 10),2),2))
ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
            callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))
evolve(ga)
best(ga)
max(ga)                      # the Maximum chromosome may be different to the best
bestFitness(ga)
maxFitness(ga)
```

`maxFitness.Galgo` *Returns the fitness of the maximum chromosome*

Description

Returns the fitness of the maximum chromosome.

Usage

```
## S3 method for class 'Galgo':
maxFitness(.O, ...)
```

Value

Returns the fitness of the maximum chromosome from the current population.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Galgo`. `*best()`, `*max()`, `*bestFitness()`.

Examples

```
wo <- World(niches=newRandomCollection(Niche(chromosomes=newRandomCollection(
      Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5)), 10),2),2))
ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
            callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))
evolve(ga)
best(ga)
max(ga)                       # the Maximum chromosome may be different to the best
bestFitness(ga)
maxFitness(ga)
```

`evolve.Galgo` *Evolves the chromosomes populations of a Galgo (Genetic Algorithm)*

Description

A generation consist of the evaluation of the fitness function to all chomosome populations and the determination of the maximum and best chromosomes. If a stoping rule has not been met, `progeny` is called to generate an "evolved" population and the process start again. The stoping rules are `maxGenerations` has been met, `goalFitness` has been reach or user-cancelled via `callBackFunc`. As any other program in R the process can be broken using `Ctrl-C` keys (`Esc` in Windows). Theoretically, if the process is cancelled via `Ctrl-C`, the process may be continued calling `evolve` method again; however it is never recommended.

Usage

```
## S3 method for class 'Galgo':
evolve(.O, parent=.O, ...)
```

Arguments

`parent`	The original object calling for the evaluation. This is passed to the fitness function in order to evaluate the function inside a context. Commonly it is a `BigBang` object.

Value

Returns nothing. The results are saved in the `Galgo` object.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Galgo`.

Examples

```
wo <- World(niches=newRandomCollection(Niche(chromosomes=newRandomCollection(
Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5)), 10),2),2))
ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
            callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))
evolve(ga)
best(ga)
bestFitness(ga)
```

`plot.Galgo` *Plots information about the Galgo object*

Description

. See arguments for details.

Usage

```
## S3 method for class 'Galgo':
plot(.O,
        type=c("all", "populations", "fitness", "maxchromosomes"),
        ...)
```

Arguments

`type`	The type of plot. `"populations"` will plots all chromosomes in one axis and the genes in the other axis. The maximum chromosome in each population is drawn with `"M"` whereas the best chromosome is drawn with `"B"`. The best chromosome from `Galgo` object is drawn with `"x"`. This plot give an overview of the population coverage. `"fitness"` plots the evolution of the maximum fitness in vertical axis against generation in horizontal. `maxChromosomes` plots the evolution of the maximum chromosomes in horizontal and the generation in vertical. `all` plots altogether.
`main,xlab, ylab,col,pch`	`World` defaults for common plot parameters. Their usage overwrite the default value. `col` controls the color for chromosomes
`...`	Other user named values to include in the object.

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `Galgo`.

Examples

```
wo <- World(niches=newRandomCollection(Niche(chromosomes=
                   newRandomCollection(Chromosome(genes=
                   newCollection(Gene(shape1=1, shape2=1000),5)),
                   10),2),2))
ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
            callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))
evolve(ga)
best(ga)
bestFitness(ga)
plot(ga)

reInit(ga)
generateRandom(ga)
evolve(ga)
best(ga)
bestFitness(ga)
plot(ga)
```

`BigBang` *Represents the ensemble of the results of evolving several Galgo objects*

Description

The `BigBang` object is an attempt to use more the information of a large collection of solutions instead of a unique solution. Perhaps we are studying the solution landscape or we would like to "ensemble" solutions from other "small" solutions. For complex problems (or even simple problems), the number of "solutions" may be very large and diverse. In the context of classification for microarray data, we have seen that models assembled from many solution could be used as "general models" and that the most frequent genes in solutions provide insights for biological phenomena.

Therefore, we designed the `BigBang` object, which implements methods to run a `Galgo` object several times recording relevant information from individual galgos for further analysis. Running a BigBang takes commonly several minutes, hours or perhaps days depending on the complexity of the fitness function, the data, the `goalFitness`, the stopping rules in `Galgo`, and the number of solutions to collect. Parallelism is not explicity implemented but some methods has been implemented to make this task easy and possible.

As in a `Galgo` object, there are three stopping methods: `maxBigBangs`, `maxSolutions` and `callBackFunc`. `maxBigBangs` controls the maximum number of galgo evolutions to run; when the current evolution-cycle reaches this value, the process ends. Sometimes evolutions do not end up with a `goalFitness` reached, this is not called a "solution". Therefore, `maxSolutions` controls the maximum number of solutions desired. If `onlySolutions==FALSE`, all galgo evolutions are saved and considered as "solution", nevertheless the `solution` variable save the real status in the `BigBang` object. `callBackFunc` may ends the process if it returns `NA`. It must be considered that any R-program can be broken typing `Ctrl-C` (`Esc` in Windows). If for some reason the process has been interrupt, the `BigBang` process can continue processing the same cycle just calling the method `blast` again. However the object integrity may be risked if the process is broken in critical parts (when the object is being updated at the end of each cycle). Thus, it is recommended to break the process in the galgo "evolution".

In the case of variable selection for microarray data, some methods has been proposed that use several independent solutions to design a final solution (or set of better solutions, see XXX references *** MISSING ***).

There is configBB.VarSel and configBB.VarSelMisc functions that configure a BigBang object together with all sub-objects for common variable selection problems (e.g. classification, regression, etc.)

Usage

```
BigBang(id=0,
        galgo=NULL,
        maxBigBangs=10,
        maxSolutions=1,
        collectMode=c("bigbang", "galgos", "chromosomes"),
        onlySolutions=TRUE,
        verbose=1,
        callPreFunc=function(bigbang, galgo) TRUE,
        callBackFunc=function(bigbang, galgo) TRUE,
        callEnhancerFunc=function(chr, parent) NULL,
        data=NULL,
        saveFile=NULL,
        saveFrequency=100,
        saveVariableName=collectMode,
        saveMode=c("unObject+compress", "unObject", "object", "object+compress")
        saveGeneBreaks=NULL,
        geneNames=NULL,
        sampleNames=NULL,
        classes=NULL,
        gcFrequency=123,
        gcCalls=5,
        call=NULL,
        ...)
```

Arguments

`id`	A way to identify the object.
`galgo`	The prototype `Galgo` object that will be used to run and collect solutions.
`maxBigBangs`	The maximum number of BigBangs. A bigbang is the evolution of a `Galgo` object using the method `evolve`. When the current number of bigbangs has reached `maxBigBangs` value, the process ends.
`maxSolutions`	The maximum number of solutions. If the total number of solutions collected achieve `maxSolutions` value the process ends. A solution is defined when the `goalFitness` has been reach. When the `Galgo` object ends and `goalFitness` has not been reached, The `best` chromosome is NOT saved unless `onlySolutions` is `FALSE`, in this case `maxSolutions` and `maxBigBangs` are equivalent.
`collectMode`	The type of result to collect for further analysis. `"galgos"` saves every evolved galgo object, thus it consumes a lot of memory; more than 100 is perhaps not recommendable. `"chromosomes"` and `"bigbangs"` save the best chromosome, its fitness, and fitness evolution in the `BigBang` object. `"bigbang"` saves the `BigBang` object to disk whereas `"chromosome"` saves only the list of chromosomes.

`onlySolutions`	If `TRUE` only solutions that has been reach the `goalFitness` are saved. Otherwise, all solutions are saved and counted as “solution” and `$solutions` variable contains the real status.
`verbose`	Instruct the BigBang to display the general information about the process. When `verbose==1` this information is printed every evolution. In general every `verbose` number of generation would produce a line of output. Of course if `verbose==0` would not display a thing at all.
`callPreFunc`	A user-function to be called before every evolution. It should receive the `BigBang` and `Galgo` objects. If the result is `NA`, the process ends.
`callBackFunc`	A user-function to be called after every evolution. It should receive the `BigBang` and `Galgo` objects. If the result is `NA`, the process ends. When `callBackFunc` is for instance `plot` the trace of the evolution is nicely viewed in a plot; however, in long runs it can consume time and memory.
`callEnhancerFunc`	A user-function to be called after every evolution to improve the solution. It should receive a `Chromosome` and the `BigBang` objects as parameters, and must return a new `Chromosome` object. If the result is `NULL` nothing is saved. The result replace the original evolved chromosomes, which is saved in evolvedChromosomes list variable in the `BigBang` object. For functional genomics data, we have included two general routines called `geneBackwardElimination` and `robustGeneBackwardElimination` to generate “enhanced” chromosomes.
`data`	Any user-data can be stored in this variable (but it is not limited to `data`, the user can insert any other like `myData`, `mama.mia` or `whatever` in the `...` argument).
`saveFile`	The file name where the objects would be saved (see `collectMode`).
`saveFrequency`	How often the operation of saving would occur. Saving is a time-consuming operation, low values may degradate the performance.
`saveVariableName`	The prefereable variable name used for saving (this will be needed when loading).
`saveMode`	Any combinations of the two options `compress` and `unObject`. It can be character vector length 1 or larger. For example, `saveMode=="compress+unObject"` would call `unObject` and save the file using `compress=TRUE`. The vector `c("object","compress")` (or shorter `c("compress")`) would save the `BigBang` object and compressed. It is not recommended to save the crude object because the functions varibles are stuck to environments and R will try to save those environments together, the result can be a waste of disk space and saving time. We strongly recommend `saveMode="unObject+compress"`.
`geneNames`	Gene names (if they are discrete and finite).
`sampleNames`	Sample names (if any).
`classes`	Class of the original samples (useful for classification problems only).
`saveGeneBreaks`	In the case of variable selection for microarray data (and other problems with discrete and finite genes), a summary on the genes selected is computed and saved in each evolution. It is used to facilitate the computation for some plots and others methods. For no-finite gene applications, it may be useful interpreting `saveGeneBreaks` as the breaks needed to create an histogram based on the genes included in the “best”.

`gcFrequency`	How often the garbage collector would be called. Useful if memory needs to be collected during the process.
`gcCalls`	How many calls to garbage collector (we have seen that many consecutive calls to `gc()` is better [R < 2.0]).
`...`	Other user named values to include in the object.

Class

Package: galgo
Class BigBang

```
Object
~~|
~~+--BigBang
```

Directly known subclasses:

public static class **BigBang**
extends Object

Fields and Methods

Methods:

`activeChromosomeSet`	Focus the analysis to different sets of chromosomes.
`addCount`	Add a chromosome to rank and frequency stability counting.
`addRandomSolutions`	Adds random pre-existed solutions.
`as.matrix`	Prints the representation of the BigBang object.
`assignParallelFile`	Assigns a different saveFile value for parallelization.
`blast`	Evolves Galgo objects saving the results for further analysis.
`buildCount`	Builds the rank and frequency stability counting.
`classPredictionMatrix`	Predicts class for samples from chromosomes.
`computeCount`	Compute the counts for every gene from a set of chromosomes..
`confusionMatrix`	Computes the class confusion matrix from a class prediction matrix.
`distanceImportanceNetwork`	Converts geneImportanceNetwork matrix to distance matrix.
`filterSolution`	Filters solutions.
`fitnessSplits`	Computes the fitness function from chromosomes for different splits.
`formatChromosome`	Converts chromosome for storage in BigBang object.
`forwardSelectionModels`	Gets the “best” models using top-ranked genes and a forward-selection strategy
`geneCoverage`	Computes the fraction of genes present in the top-rank from the total genes pres
`geneFrequency`	Computes the frequency of genes based on chromosomes.
`geneImportanceNetwork`	Computes the number of times a couple of top-ranked-genes are present in mod
`geneRankStability`	Computes the rank history for top-ranked genes.
`getFrequencies`	Computes gene freqencies.
`heatmapModels`	Plots models using heatmap plot.
`loadParallelFiles`	Load all files saved during the parallelization.
`meanFitness`	Computes the “mean” fitness from several solutions.
`meanGeneration`	Computes the mean number of generations requiered to reach a given fitness va
`mergeBangs`	Merges the information from other BigBang objects.

`pcaModels`	Plots models in principal components space.
`plot`	Plots about the collected information in a BigBang object.
`predict`	Predicts the class or fitting of new set of samples.
`print`	Prints the representation of a BigBang object.
`saveObject`	Saves the BigBang object into a file in a suitable format.
`sensitivityClass`	Computes the sensitivity of class prediction.
`specificityClass`	Computes the specificity of class prediction.
`summary`	Prints the representation of the BigBang object.

Methods inherited from Object:
as.list, unObject, ,<-, [[, [[<-, as.character, attach, clone, detach, equals, extend, finalize, getFields, getInstanciationTime, getStaticInstance, hasField, hashCode, ll, load, objectSize, print, save

Other Fields Created

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

`Gene`, `Chromosome`, `Niche`, `World`, `Galgo`, `configBB.VarSel`(), `configBB.VarSelMisc`().

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))
ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
                             callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))

#evolve(ga) ## not needed here

bb <- BigBang(galgo=ga, maxSolutions=10, maxBigBangs=10)
blast(bb)
## it performs 10 times evolve() onto ga object
## every time, it reinitilize and randomize
## finally, the results are saved.
plot(bb)

#it is missing a microarray classification example
```

`blast.BigBang` *Evolves Galgo objects saving the results for further analysis*

Description

The basic process is as follows.\n \tab1. Clone `Galgo` and generate random chromosomes\n \tab2. Call `evolve` method\n \tab3. Save results in `BigBang` object\n \tab4. Verify stop rules\n \tab5. Goto 1\n

Usage

```
## S3 method for class 'BigBang':
blast(.bb, add=0, ...)
```

Arguments

`add`	Force to add a number to maxBigBangs and maxSolutions in order to search for more solutions.

Value

Returns nothing. The results are saved in the the `BigBang` object.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`. `evolve.Galgo`().

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))
ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
                             callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))

#evolve(ga) ## not needed here

bb <- BigBang(galgo=ga, maxSolutions=10, maxBigBangs=10)
blast(bb)
plot(bb)
blast(bb, 3)
plot(bb)
```

`saveObject.BigBang` *Saves the BigBang object into a file in a suitable format*

Description

Saves the BigBang object into a file in a suitable format.

Usage

```
## S3 method for class 'BigBang':
saveObject(.bbO, file=.bbO$saveFile, mode=.bbO$saveMode, ...)
```

Arguments

`file`	The file name where the data will be saved. The default is taking the `$saveFile` variable form the `BigBang` object.
`saveMode`	Character vector specifying the saving mode. The default is taking the `$saveMode` variable from the `BigBang` object. Any combinations of the two options `compress` and `unObject`. It can be character vector length 1 or larger. For example, `saveMode=="compress+unObject"` would call `unObject` and save the file using `compress=TRUE`. The vector `c("object","compress")` (or shorter `c("compress")`) would save the `BigBang` object and compressed. It is not recommended to save the crude object because the functions varibles are stuck to environments and R will try to save those environments together, the result can be a waste of disk space and saving time. We strongly recommend `saveMode="unObject+compress"`.

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`.

Examples

```
   cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
   ni <- Niche(chromosomes=newRandomCollection(cr, 10))
   wo <- World(niches=newRandomCollection(ni,2))
   ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
               callBackFunc=plot,
               fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))
```

```
#evolve(ga) ## not needed here

bb <- BigBang(galgo=ga, maxSolutions=10, maxBigBangs=10)
blast(bb)
saveObject(bb, file="bb.Rdata", mode="unObject+compress")
```

print.BigBang *Prints the representation of a BigBang object*

Description

Prints the representation of a BigBang object.

Usage

```
## S3 method for class 'BigBang':
print(o, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`. `*summary()`.

Examples

```
wo <- World(niches=newRandomCollection(Niche(chromosomes=newRandomCollection(
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))
ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
            callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))

#evolve(ga) ## not needed here

bb <- BigBang(galgo=ga, maxSolutions=10, maxBigBangs=10)
blast(bb)
bb
print(bb)
summary(bb)
```

summary.BigBang *Prints the representation of the BigBang object*

Description

Prints the representation of the BigBang object.

Usage

```
## S3 method for class 'BigBang':
summary(o, ...)
```

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see BigBang. *print().

Examples

```
wo <- World(niches=newRandomCollection(Niche(chromosomes=newRandomCollection(
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))
ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
            callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))

#evolve(ga) ## not needed here

bb <- BigBang(galgo=ga, maxSolutions=10, maxBigBangs=10)
blast(bb)
bb
print(bb)
summary(bb)
```

getFrequencies.BigBang

Computes gene freqencies

Description

Computes gene freqencies.

Usage

```
## S3 method for class 'BigBang':
getFrequencies(o, filter="none", subset=TRUE, ...)
```

Arguments

`filter`	The `BigBang` object can save information about solutions that did not reach the `goalFitness`. `filter=="solutions"` ensures that only chromosomes that reach the `goalFitness` are considered. `fitlter=="none"` take all chromosomes. `filter=="nosolutions"` consider only no-solutions (for comparative purposes).
`subset`	Second level of filter. `subset` can be a vector specifying which filtered chromosomes are used. It can be a logical vector or a numeric vector (indexes in order given by `$bestChromosomes` in `BigBang` object variable).

Value

Returns a list of components.

`new`	Current gene count.
`n`	Number of genes in new.
`nChr`	Number of chromosomes counted.
`ord`	Decreasing order.
`rnk`	Rank (starting with maximum).

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`. `*geneFrequency()`.

Examples

```
    *** missing ***
```

`activeChromosomeSet.BigBang` *Focus the analysis to different sets of chromosomes*

Description

Swaps the "active" chromosomes for analysis. All the plots and methods compute the information from the variable `$bestChromosomes`, `$bestFitness` and `$count`. When `callEnhancerFunc` has been used it could be needed to use the same plots with different sets of chromosomes. `activeChromosomeSet` swaps the information between different chromosomes sets to concentrate the analysis on that set.

Usage

```
## S3 method for class 'BigBang':
activeChromosomeSet(.O,
        set=c("evolved", "default", "custom"),
        count=TRUE,
        chromosomes=NULL,
        fitness=NULL,
        ...)
```

Arguments

`set`	`"evolved"` specify to analyse original chromosomes that were evolved insted of the enhanced (see evolvedChromosomes and evolvedFitness parameters). `"default"` restore the original chromosomes. `"custom"` is for user-specified chromosomes and fitness.
`count`	Instruct to re-build the `count` matrix used for some plots. Recommended to be `TRUE` always.
`chromosomes`	The chromosome set to analyse. The default is to use the variable `$evolvedChromosomes` from the `BigBang` object.
`fitness`	The fitness of the chromosomes to analyse. The default is to use the variable `$evolvedFitness` from the `BigBang` object.

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`.

Examples

```
# bb was created
activeChromosomeSet(bb, set="evolved")
plot(bb)
activeChromosomeSet(bb, set="default")
plot(bb)
```

`plot.BigBang` *Plots about the collected information in a BigBang object*

Description

Plots about the collected information in a BigBang object. See arguments for details.

Usage

```
## S3 method for class 'BigBang':
plot(o,
        y=NULL,
        ...,
        type=c("genefrequency", "generank", "generankstability", "geneoverlap",
        filter=c("none", "solutions", "nosolutions"),
        subset=TRUE,
        mcol=8,
        mord=50,
        rcol=(if (mcol < 2) c(rep(1, mord), 0) else c(cut(1:mord, breaks = mcol,
        new.dev=FALSE,
        sort.chr=4,
        freq.col=rgb(0.4, 0.4, 0.4),
        freq.all.labels=FALSE,
        rank.lwd=5,
        rank.order=c("rank", "reverse", "random"),
        gene.names=TRUE,
        rankindex.log=NULL,
        coverage.log="x",
        classFunc=NULL,
        classes=NULL,
        confusion.all=TRUE,
        contrast=0.15,
        coverage=c(0.25, 0.5, 0.75, 1),
        samples=NULL,
        samples.cex=0.75,
        pch=20,
        main=o$main,
        nbf=1,
        net.method=c("isoMDS", "cmdscale", "sammon"),
        net.th=2,
        node.size=6,
        node.name=c("index", "rownames"),
        node.namecol=NULL,
        xlim=NULL,
        ylim=NULL,
        xlab="",
        ylab="",
        cex=1)
```

Arguments

`y`	Optional additional data relative to the plot type. Some types may benefit from this parameter.
`type`	Specify the types of plots.
"genefrequency"	Plot the frequency of genes computed from the chromosomes in the specified filter (see `filter` and `subset`). Peaks reveal high-frequent genes, thus potentially "important" genes. "Top-ranked" genes are colored respect to its rank (see `mord, mcol and rcol`). Labels are optional (see `freq.all.labels`).
"generank"	Similar to `"genefrequency"` but drawing only "top-ranked" genes and sorte by rank.
"generankstability"	Because of the stochasticity of the process, it is difficult to decide how many solutions are required to stabilize the gene ranks and thus avoiding random fluctuations. `"generankstability"` is designed to show visually how the rank of the current "top-ranked" genes has been changed in the course. Many changes of colours reveals rank instability whereas few or no-changes show stability. Commonly, the top (10 to 20) genes are the quickest genes to stabilize. One can decide to "stop" the process or "start" the analysis when at least 10 or 20 genes has been "stable" for 100 or 200 solutions.
"geneoverlap"	Overview of how the chromosomes are "overlapped" and "represented" by the top-ranked genes (see `sort.chr`).
"geneoverlaphor"	Horizontal version of `"geneoverlap"`.
"genesintop"	Shows the histogram of the number of top-genes included in models.
"fitness"	The evolution of the maximum fitness for each solution. It includes descriptive confidence intervals (average among all and average among the worst). The point where the highest interval intersects the `goalFitness` is the "average" number of generations needed to reach that fitness value. It could be useful for deciding the number of generations and the goal fitness value.
"fitnessboxes"	Similar to `"fitness"` but using boxplot. Useful for "statistical" intervals.
"generations"	Distribution of the final generation from each galgo. A large peak at `minGenerations` means "premature" convergence or "easy" codegoalFitness; perhaps increasing the `goalFitness` worth. A trend to "maxGenerations" may be indicative of very high `goalFitness` or low `maxGenerations`. (may be normal when `onlySolutions == FALSE`).
"rankindex"	Shows the rank versus index. A vertical line indicate many genes in the same rank, probably due by random, not stable or insuficent solutions.
"genefrequencydist"	Shows the distribution of the gene frequency.
"topgenenumber"	Shows the number of genes whose frecuency is higher that specific values. It try to answer questions like "how many genes appears in X chromosomes?". It is helpful to decide how many "top-genes" include in plots. Genes with low frequency may be asociated with random fluctations.

"confusion"	For classification problems, it shows the confusion matrix and the probability for all samples in each class. It needs a `classFunc` specification (unless `$data$classFunc` exists in the `BigBang` object) or `y=classPredictionMatrix`. An `NA` "class" has been add in the predicted class axis (vertical) for those classification methods that cannot produce a class prediction in all cases. The default is that the bar size is meant as "probability" of that sample to pertain in that class. The sensitivity and specificity for all classes are given in the horizontal axis (sensitivity=TP/TP+FN, specificity=TN/TN+FP, TP=True Positives, TN=True Negatives, FP=False Positives, FN=False Negatives).
"confusionbox"	Similar than "confusion" but showing distribution boxes for each class.
"confusionpamr"	Similar than "confusion" in style similar to pamr package.
"splits"	Gives an overview on how the splits were build. Perhaps useless.
"splitsmap"	Gives an clustering overview on how the splits were build (to detect biased splits). Perhaps useless.
"splitsfitness"	It plots the boxplot of the evaluation of chromosomes in different splits. Perhaps useless.
"fitnesssplits"	Plots the distribution of fitness evaluated in different splits. To check whether the chromosomes are "split-dependent".
"fitnesssplitsbox"	It plots the boxplot of the evaluation of chromosomes in different splits. Perhaps useless.
"genecoverage"	Plot the number of possible top-ranked genes in horizontal versus the percentage of total genes present in chromosomes. It tries to answer questions like "how many `N` top-genes are required to ensure that these `N` top-genes cover at least 50% of all genes in chromosomes?". Solution: Plot (`type="genecoverage"`) look for 0.5 (50%) in vertical axis (or use `coverage=0.5`) then project the point in the plot to horizontal axis.
"genenetwork"	Plot the "dependency" of genes to each other in a network format. The distance is a measure of how many chromosomes those two genes are together normalized to the total number of interactions. The thickness of the connection is relative to the relative strength of the shown connections.
`filter`	The `BigBang` object can save information about solutions that did not reach the `goalFitness`. `filter=="solutions"` ensures that only chromosomes that reach the `goalFitness` are considered. `fitlter=="none"` take all chromosomes. `filter=="nosolutions"` consider only no-solutions (for comparative purposes).
`subset`	Second level of filter. `subset` can be a vector specifying which filtered chromosomes are used. It can be a logical vector or a numeric vector (indexes in order given by `$bestChromosomes` in `BigBang` object variable). If it is a numeric vector length one, a positive value means take those top chromosomes sorted by fitness, a negative value take those at bottom.
`mord`	The number of "top-ranked-genes" to highlight.
`mcol`	The number of colours (or sections) to highlight ranked genes.
`rcol`	The specific colours for every "top-ranked-gene". If specified, its length should be `mord+1`.
`new.dev`	For `type` is a vector length greather than 1, `TRUE` create two new plot windows.
`sort.chr`	For `type=="geneoverlap"`, `sort.chr` can be used to sort the chromosomes. `sort.chr==0` sort the genes according to its fitness which could reveal trends in gene-fitness. `sort.chr < 0` no sort at all, the chromosomes

	are shown as they were obtained. `sort.chr > 0` controls the chromosome sorting by the prescence of “top-ranked” genes and the recursive level (as higher as slower).
`freq.col`	For `type=="genefrequency"`, `freq.col` is the colour for non “top-ranked” genes.
`freq.all.labels`	For `type=="genefrequency"`, `freq.all.labels` plot the names for all “top-ranked” genes.
`rank.lwd`	For `type=="generank"` (and others), `rnk.lwd` is the line width (see `lwd`).
`rank.order`	For `type=="generank"` (and others), `rank.order` controls the order of ranked genes.
`genes.names`	`TRUE` for plotting gene names (from `BigBang` object). `FALSE` use gene indexes instead. Character vector for user-specification.
`rankindex.log`	Change the log plot parameter for `type=="rankindex"`.
`coverage.log`	Change the log plot parameter for `type=="genecoverage"`.
`classFunc`	Specify the classification function when a `type=="confusion"` and a confusion matrix is needed.
`classes`	Specify the classes (overwriting the `BigBang` default) when a `type=="confusion"` and a confusion matrix is needed.
`confusion.all`	`TRUE` draw mean probability values for all combinations in the confusion plot.
`contrast`	Contrast factor for same colour/section in ranks. 0=All genes in same section are exactly the same colour. 1="Maximum" contrast factor.
`coverage`	For `type="genecoverage"`, `coverage` specify the points for comparison. For instance 0.5 meant the number of top-ranked genes needed that cover 50% of total genes present in all chromosomes.
`samples`	Specify the sample names (overwriting the `BigBang` default)
`samples.cex`	Specify the character size for ploting the sample names.
`nbf`	If `type=“fitnessboxes”`, `nbf` specifies the divisor of the number of boxes in the plot. Defaults to 1.
`net.th`	If `type=genenetwork”`, it specifies the connections to plot. `net.th < 1` specifies to plot connections whose distance <= net.th. `net.th >= 1` specifies to plot the highest net.th connections for each node. Default is 2.
`net.method`	If `type=genenetwork”`, it specifies the method to compute the coordinates. Methods are `c("isoMDS","cmdscale","sammon")`.
`node.size`	If `type=genenetwork”`, it specifies the size of the node.
`node.name`	If `type=genenetwork”`, it specifies the naming scheme, which can be `c("index","rowname`
`node.namecol`	If `type=genenetwork”`, it specifies the color of the node names.
`main,xlab, ylab,xlim,ylim,cex,pch`	`BigBang` defaults for common plot parameters. Their usage overwrite the default value.
`...`	Other plot parameters (not always passed to subsequent routines).

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`.

Examples

```
cr <- Chromosome(genes=newCollection(Gene(shape1=1, shape2=1000),5))
ni <- Niche(chromosomes=newRandomCollection(cr, 10))
wo <- World(niches=newRandomCollection(ni,2))
ga <- Galgo(populations=newRandomCollection(wo,1), goalFitness = 0.75,
            callBackFunc=plot,
            fitnessFunc=function(chr, parent) 5/sd(as.numeric(chr)))

#evolve(ga) ## not needed here

bb <- BigBang(galgo=ga, maxSolutions=10, maxBigBangs=10)
blast(bb)
plot(bb)
plot(bb, type=c("fitness","genefrequency"))
plot(bb, type="generations")
```

`meanGeneration.BigBang` *Computes the mean number of generations requiered to reach a given fitness value*

Description

Computes the mean number of generations requiered to reach a given fitness value. We have seen that this value is actually closer to the median of the final generation.

Usage

```
## S3 method for class 'BigBang':
meanGeneration(o, filter="none", subset=TRUE, fitness=o$galgo$goalFitness, ...)
```

Arguments

`filter`	The `BigBang` object can save information about solutions that did not reach the `goalFitness`. `filter=="solutions"` ensures that only chromosomes that reach the `goalFitness` are considered. `fitlter=="none"` take all chromosomes. `filter=="nosolutions"` consider only no-solutions (for comparative purposes).

`subset`	Second level of filter. `subset` can be a vector specifying which filtered chromosomes are used. It can be a logical vector or a numeric vector (indexes in order given by `$bestChromosomes` in `BigBang` object variable). If it is a numeric vector length one, a positive value means take those top chromosomes sorted by fitness, a negative value take those at bottom.
`fitness`	The fitness value desired. The default is `$galgo$goalFitness`.

Details

This function use `meanFitness` to compute the mean number of generations from solutions, then it finds the generation whose fitness mean value is not below the specified fitness.

Value

Return the expected mean generation.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`. `*meanFitness()`

Examples

```
#bb is a BigBang object
meanGeneration(bb)
```

`geneFrequency.BigBang`

Computes the frequency of genes based on chromosomes

Description

Computes the frequency of genes based on chromosomes. It really returns `getFrequiences` using the `$new` variable, adding gene names, and filtering for `cutoff`.

Usage

```
## S3 method for class 'BigBang':
geneFrequency(o,
        filter="none",
        subset=TRUE,
        gene.names=TRUE,
        cutoff=-1,
        value=c("frequency", "indexes", "ranks"),
        ...)
```

Arguments

`filter`	The `BigBang` object can save information about solutions that did not reach the `goalFitness`. `filter=="solutions"` ensures that only chromosomes that reach the `goalFitness` are considered. `fitlter=="none"` take all chromosomes. `filter=="nosolutions"` consider only no-solutions (for comparative purposes).
`subset`	Second level of filter. `subset` can be a vector specifying which filtered chromosomes are used. It can be a logical vector or a numeric vector (indexes in order given by `$bestChromosomes` in `BigBang` object variable). If it is a numeric vector length one, a positive value means take those top chromosomes sorted by fitness, a negative value take those at bottom.
`gene.names`	`TRUE` for naming the result with the stored `$geneNames` in oject `BigBang`. Other character vector to name-specific.
`cutoff`	Only genes whose frequency is greather than `cutoff` are repored.
`value`	The result. `"frequency","indexes","ranks"`

Value

A table when `value=="frequency"`, otherwise, a vector.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`.

Examples

```
#bb is a BigBang object
geneFrequency(bb)
```

`geneRankStability.BigBang`

Computes the rank history for top-ranked genes

Description

Computes the rank history for top-ranked genes.

Usage

```
## S3 method for class 'BigBang':
geneRankStability(o,
        filter="none",
        subset=TRUE,
        gene.names=TRUE,
        lastSolutionFirst=TRUE,
        ...)
```

Arguments

`filter`	The `BigBang` object can save information about solutions that did not reach the `goalFitness`. `filter=="solutions"` ensures that only chromosomes that reach the `goalFitness` are considered. `fitlter=="none"` take all chromosomes. `filter=="nosolutions"` consider only no-solutions (for comparative purposes).
`subset`	Second level of filter. `subset` can be a vector specifying which filtered chromosomes are used. It can be a logical vector or a numeric vector (indexes in order given by `$bestChromosomes` in `BigBang` object variable). If it is a numeric vector length one, a positive value means take those top chromosomes sorted by fitness, a negative value take those at bottom.
`gene.names`	`TRUE` for naming the result with the stored `$geneNames` in oject `BigBang`. Other character to name-specific.
`lastSolutionFirst`	Order of the results. `TRUE` the las solutions is given in the first column.

Value

A matrix which genes are fit in rows and solutions in columns.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`.

Examples

```
#bb is a BigBang object
geneRankStability(bb)
```

meanFitness.BigBang

Computes the “mean” fitness from several solutions

Description

Computes the “mean” fitness from several solutions.

Usage

```
## S3 method for class 'BigBang':
meanFitness(o, filter="none", subset=TRUE, ...)
```

Arguments

`filter`	The `BigBang` object can save information about solutions that did not reach the `goalFitness`. `filter=="solutions"` ensures that only chromosomes that reach the `goalFitness` are considered. `fitlter=="none"` take all chromosomes. `filter=="nosolutions"` consider only no-solutions (for comparative purposes).
`subset`	Second level of filter. `subset` can be a vector specifying which filtered chromosomes are used. It can be a logical vector or a numeric vector (indexes in order given by `$bestChromosomes` in `BigBang` object variable). If it is a numeric vector length one, a positive value means take those top chromosomes sorted by fitness, a negative value take those at bottom.

Details

The mean is built considering all solutions. For solutions that have finished earlier, the final fitness is used for futher genertions.

Value

A vector with the mean fitness in each generation.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`.

Examples

```
#bb is a BigBang object
geneRankStability(bb)
```

`classPrediction` *Function used to predict class evaluating a fitness function in many train-test sets*

Description

Function used to predict class evaluating a fitness function in many train-test sets.

Usage

```
classPrediction(chr, parent, splits = 1:length(parent$data$splitTrain), set = pa
```

Arguments

`chr`	Chromosome or vector object.
`parent`	Parent object, commonly BigBang object.
`splits`	Which sets of splits will be used to compute the fitness function. Default to all splits defined in `parent$data$splitTrain`.
`set`	Weigths used in training and test sets. Vector of two values. The first is the weight of train error. The second is the weight of test error. The default value is taken from parent$data$testErrorWeights whose default is c(0,1) (considering only test error).
`mode`	The type of value to return. `"sum"` returns a matrix with the number of times a sample (rows) has been predicted as any class (columns). The values are proportional to train and test weights. `"probability"` convertion of `"sum"` to probabilities dividing each row by its sum. `"class"` returns a vector of the predicted class given by majority vote.

Value

A matrix or vector depending on `mode` paramater.

Note

This function is designed to be used in forwardSelectionModels

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K.

See Also

`forwardSelectionModels`

Examples

classPredictionMatrix.BigBang

Predicts class for samples from chromosomes

Description

Predicts class for samples from chromosomes.

Usage

```
## S3 method for class 'BigBang':
classPredictionMatrix(o,
        filter="none",
        subset=TRUE,
        classFunc=o$data$classFunc,
        classes=NULL,
        sampleNames=NULL,
        chromosomes=NULL,
        verbose=TRUE,
        use.cache=TRUE,
        ...)
```

Arguments

`filter`	The `BigBang` object can save information about solutions that did not reach the `goalFitness`. `filter=="solutions"` ensures that only chromosomes that reach the `goalFitness` are considered. `fitlter=="none"` take all chromosomes. `filter=="nosolutions"` consider only no-solutions (for comparative purposes).
`subset`	Second level of filter. `subset` can be a vector specifying which filtered chromosomes are used. It can be a logical vector or a numeric vector (indexes in order given by `$bestChromosomes` in `BigBang` object variable). If it is a numeric vector length one, a positive value means take those top chromosomes sorted by fitness, a negative value take those at bottom.
`classFunc`	The function that provides the class prediction.
`classes`	The known classes if they are different than those in `BigBang$classes` (or `BigBang$data$classes`).
`sampleNames`	Sample names if they are different than those in `BigBang$classes` (or `BigBang$data$class`
`chromosomes`	Specific chromosome list. The default is use the solution from `BigBang` object filtered by `filter` and `subset`.
`verbose`	Display processing information.
`use.cache`	Save/Restore values from previous computations with same parameters.

Details

`classFunc` is called for each chromosome, therefore this routine can be time consuming depending on the behaviour of `classFunc`. The default `classFunc` from `configBB.VarSel` computes the class by majority of votes using all splits. Use `...` for specifying `splits`, `set` or any other parameter for `classFunc`.

Value

A matrix whose rows are samples and columns are classes. Each value is the number of times the sample was predicted as that class.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`. `*confusionMatrix()`.

Examples

```
#bb is a BigBang object
cpm <- classPredictionMatrix(bb)
cpm
cm <- confusionMatrix(bb)
cm
#equivalent and quicker because classPredictionMatrix is provided
cm <- confusionMatrix(bb, cpm)
cm

specificityClass(bb, cm)
specificityClass(bb, cpm)
specificityClass(bb)
# all are equivalent
sensitivityClass(bb, cpm)
sensitivityClass(bb, cm)
sensitivityClass(bb)
# all are equivalent
```

`confusionMatrix.BigBang`

Computes the class confusion matrix from a class prediction matrix

---

Description

Computes the class confusion matrix from a class prediction matrix.

Usage

```
## S3 method for class 'BigBang':
confusionMatrix(o, cm, ...)
```

Arguments

`cm`	The confusion matrix, If not specified `classPredictionMatrix` method is call using the `BigBang` object provided and `...`
`...`	Further parameters passed to `classPredictionMatrix` when `cm` is not specified.

Details

The matrix is computed getting the predicted class proportions for all samples; accumulating the proportions; finally producing propotions for all classes. This procedure is equivalent to having the same weights (priors) for all classes.

Value

A matrix with original classes in rows and predicted classes in columns. Each value represent the "probability" for a sample within a given class to be predicted as any other class.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`. `*classPredictionMatrix()`, `*confusionMatrix()`.

Examples

```
#bb is a BigBang object
cpm <- classPredictionMatrix(bb)
cpm
cm <- confusionMatrix(bb)
cm
#equivalent and quicker because classPredictionMatrix is provided
cm <- confusionMatrix(bb, cpm)
cm

specificityClass(bb, cm)
specificityClass(bb, cpm)
specificityClass(bb)
# all are equivalent
sensitivityClass(bb, cpm)
sensitivityClass(bb, cm)
sensitivityClass(bb)
# all are equivalent
```

sensitivityClass.BigBang

Computes the sensitivity of class prediction

Description

Computes the sensitivity of class prediction.

Usage

```
## S3 method for class 'BigBang':
sensitivityClass(o, cm, ...)
```

Arguments

`cm`	The confusion matrix or the class prediction matrix. If missing, `confusionMatrix` method is called using the object and `...` as other arguments
`..`	Further parameters when `cm` is missing.

Details

Sensitivity is the probability that a sample of class `X` will be predicted as the same class `X`. High sensitivity detect true positives. Sensitivity = TP / (TP + FN) TP - True Positives: Example for class A, TP = Paa FN - False Negatives: Example for class A, FN = Pab + Pac + Pax Confusion Matrix: [Predicted Class] ClassA ClassB ClassC "misclass" ClassA Paa Pab Pac Pax ClassB Pba Pbb Pbc Pbx ClassC Pca Pcb Pcc Pcx

Value

A vector with the sensitivities of prediction for every class.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`. `*classPredictionMatrix()`, `*confusionMatrix()`.

Examples

```
#bb is a BigBang object
cpm <- classPredictionMatrix(bb)
cpm
cm <- confusionMatrix(bb)
cm
#equivalent and quicker because classPredictionMatrix is provided
```

```
cm <- confusionMatrix(bb, cpm)
cm

specificityClass(bb, cm)
specificityClass(bb, cpm)
specificityClass(bb)
# all are equivalent
sensitivityClass(bb, cpm)
sensitivityClass(bb, cm)
sensitivityClass(bb)
# all are equivalent
```

specificityClass.BigBang

Computes the specificity of class prediction

Description

Computes the specificity of class prediction.

Usage

```
## S3 method for class 'BigBang':
specificityClass(o, cm, ...)
```

Arguments

`cm`	The confusion matrix or the class prediction matrix. If missing, `confusionMatrix` method is called using the object and `...` as other arguments
`..`	Further parameters when `cm` is missing.

Details

Specificity is the probability that a sample of class different to `X` will NOT be predicted as class `X`. High specificity avoids false positives. Specificity = TN / (TN + FP) TN - True Negatives: For class A, TN = Pbb + Pbc + Pbx + Pcb + Pcc + Pcx FP - False Positives: For class A, FP = Pba + Pca Confusion Matrix: [Predicted Class] ClassA ClassB ClassC "misclass" ClassA Paa Pab Pac Pax ClassB Pba Pbb Pbc Pbx ClassC Pca Pcb Pcc Pcx

Value

A vector with the specificity of prediction for every class.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`. `*classPredictionMatrix()`, `*confusionMatrix()`.

Examples

```
#bb is a BigBang object
cpm <- classPredictionMatrix(bb)
cpm
cm <- confusionMatrix(bb)
cm
#equivalent and quicker because classPredictionMatrix is provided
cm <- confusionMatrix(bb, cpm)
cm

specificityClass(bb, cm)
specificityClass(bb, cpm)
specificityClass(bb)
# all are equivalent
sensitivityClass(bb, cpm)
sensitivityClass(bb, cm)
sensitivityClass(bb)
# all are equivalent
```

---

fitnessSplits.BigBang

Computes the fitness function from chromosomes for different splits

---

Description

Computes the fitness function from chromosomes for different splits.

Usage

```
## S3 method for class 'BigBang':
fitnessSplits(o,
        filter="none",
        subset=TRUE,
        fitnessFunc=o$data$modelSelectionFunc,
        maxCache=1e+06,
        chromosomes=NULL,
        use.cache=TRUE,
        ...)
```

Arguments

`filter`	The `BigBang` object can save information about solutions that did not reach the `goalFitness`. `filter=="solutions"` ensures that only chromosomes that reach the `goalFitness` are considered. `fitlter=="none"` take all chromosomes. `filter=="nosolutions"` consider only no-solutions (for comparative purposes).

`subset`	Second level of filter. `subset` can be a vector specifying which filtered chromosomes are used. It can be a logical vector or a numeric vector (indexes in order given by `$bestChromosomes` in `BigBang` object variable). If it is a numeric vector length one, a positive value means take those top chromosomes sorted by fitness, a negative value take those at bottom.
`fitnessFunc`	The function that provides the fitness for every chromosome. If the fitness is "split-sensitive" it should returns only one value (like the common `$galgo$fitnessFunc` variable). If the fitness does the splitting process itself (like `$data$modelSelectionFunc`), the result should be a vector of a fitness value for every split. The default use `$data$modelSelectionFunc`.
`maxCache`	The maximum number of values to be saved in the `BigBang` object (all variables starting with `"fitnessSplits"`). Useful for saving results between R sessions.
`chromosomes`	The chromosomes to process. The default is using `filter` and `subset` to extract the chromosomes from the `BigBang` object.
`use.cache`	Save/Restore values from previous computations with same parameters.

Value

A Matrix with chromosomes in rows and splits in columns. Each value is the result of the fitness function in a given chromosome on an split.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`. `*plot()`.

Examples

```
#bb is a BigBang object
fs <- fitnessSplits(bb)
fs
fs <- fitnessSplits(bb, fitnessFunc=bb$galgo$fitnessFunc)
fs
fs <- fitnessSplits(bb, fitnessFunc=bb$data$modelSelectionFunc) # default
fs
```

geneCoverage.BigBang

Computes the fraction of genes present in the top-rank from the total genes present in chromosomes

Description

Computes the fraction of genes present in the top-rank from the total genes present in chromosomes.

Usage

```
## S3 method for class 'BigBang':
geneCoverage(o, filter="none", subset=TRUE, chromosomes=NULL, ...)
```

Arguments

`filter`	The `BigBang` object can save information about solutions that did not reach the `goalFitness`. `filter=="solutions"` ensures that only chromosomes that reach the `goalFitness` are considered. `fitlter=="none"` take all chromosomes. `filter=="nosolutions"` consider only no-solutions (for comparative purposes).
`subset`	Second level of filter. `subset` can be a vector specifying which filtered chromosomes are used. It can be a logical vector or a numeric vector (indexes in order given by `$bestChromosomes` in `BigBang` object variable). If it is a numeric vector length one, a positive value means take those top chromosomes sorted by fitness, a negative value take those at bottom.
`chromosomes`	The chromosomes to process. The default is using `filter` and `subset` to extract the chromosomes from the `BigBang` object.

Value

A vector with the fraction of genes present in each rank from the total genes present in chromosomes.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`. `*plot()`.

Examples

```
#bb is a BigBang object
gc <- geneCoverage(bb)
gc
```

`forwardSelectionModels.BigBang`
Gets the "best" models using top-ranked genes and a forward-selection strategy

Description

Gets the "best" models using top-ranked genes and a forward-selection strategy.

Usage

```
## S3 method for class 'BigBang':
forwardSelectionModels(.O,
        filter="none",
        subset=TRUE,
        geneIndexSet=NULL,
        starti=NULL,
        endi=NULL,
        fitnessFunc=if (!is.function(.O$data$modelSelectionFunc)) .O$galgo$fitne
        minFitness=NULL,
        plot=TRUE,
        decision=c("overall", "average"),
        plot.type=c("lines", "boxplot"),
        approach=c("fitness", "error"),
        pch=20,
        result=c("all", "models", "fitness"),
        threshold=0.99,
        main=.O$main,
        mord=50,
        mcol=8,
        rcol=(if (mcol < 2) c(rep(1, mord), 0) else c(cut(1:mord, breaks = mcol,
        classFunc=.O$data$classFunc,
        compute.classes=is.function(classFunc),
        cex=1,
        ...)
```

Arguments

`filter`	The `BigBang` object can save information about solutions that did not reach the `goalFitness`. `filter=="solutions"` ensures that only chromosomes that reach the `goalFitness` are considered. `fitlter=="none"` take all chromosomes. `filter=="nosolutions"` consider only no-solutions (for comparative purposes).
`subset`	Second level of filter. `subset` can be a vector specifying which filtered chromosomes are used. It can be a logical vector or a numeric vector (indexes in order given by `$bestChromosomes` in `BigBang` object variable). If it is a numeric vector length one, a positive value means take those top chromosomes sorted by fitness, a negative value take those at bottom.
`geneIndexSet`	The genes index to use (ignoring `filter` and `subset`). If this is not specified the indexes are computed using `filter` and `subset`.

`starti`	Vector of initial index positions of models to test. If specified, should be the same length than `endi`. If omitted, the default repeat `1` until the same length than `endi`.
`endi`	Vector of final index positions of models to test.
`fitnessFunc`	The function that evaluate the performance (fitness) of every model (chromosome). The real measure is the "mean" computed from the resulted values for every chromosome. Thus `fitnessFunc` can returns a single numeric value (as in `$galgo$fitnessFunc`) or a numeric vector (as in `$data$modelSelectionfunc`). The default is `$data$modelSelectionFunc` unless it is `NULL` and `$galgo$fitnessFunc` is used.
`minFitness`	The minimum fitness requested. All models with mean fitness above this value will be reported. `NULL` specify the usage of the maximum fitness from the results. `"se*sp"` use the maximum value computed by multipling the sensitivity and specificity when `compute.classes==TRUE`.
`decision`	Specify how to select the model. `"overall"` select the model based on the accuracy of all samples whereas `"average"` selects the model based in the average accuracy per class. If the number of samples per class is exactly the same, both results are equal. The default is `"overall"`. If `classFunc` is not specified or `compute.classes==FALSE`, `decision` is forced to `"overall"`.
`plot`	Logical value indicating whether the result should be displayed.
`plot.type`	`"lines"` draws a line joining points. `"boxplot"` add a boxplot when the `fitnessFunc` returns more than one value.
`approach`	`"fitness"` draws fitness. `"error"` draws error (1-fitness).
`result`	Specify the desired output. `"models"` will report only the models above the `minFitness`. `"fitness"` will report only the fitness of the models above the `minFitness`. `"all"` (default) will report both models and fitness in a list including all computed fitnesses and class prediction accuracies (if `compute.classes==TRUE`).
`threshold`	Specify the percentage of `minFitness` for selecting models.
`mord`	Specify the number of top-ranked genes (`*plot()` and others *** MISSING ***). Defaults to 50. It should not be less than the maximum `endi`.
`mcol`	Specify the number of section for top-rank colouring.(`*plot()` and others *** MISSING ***)
`rcol`	Specify the colours of sections.(`*plot()` and others *** MISSING ***)
`classFunc`	Function that predict the class. The default is `$data$classFunc`.
`compute.classes`	Specify that class accuracies are desired (and plotted). In non-classification problems, it should be `FALSE`.
`pch,main,cex`	Plot parameters.
`...`	Other parameters used for `plot`, `fitnessFunc` and `classFunc`.

Details

It is expected that the `fitnessFunc` computes the *overall* fitness (the proportion of correctly classify samples regardless of their classes). However, this value could be slightly different to the curve marked as `"(avg)"` which is the average fitness per class. This difference is due to the different number of samples per class and the number of times specifc samples where used to be part of the test set in both, the fitness function and the class function.

Value

Depends on `result`.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`. `*plot()`, `*heatmapModels()`, `*pcaModels()`.

Examples

```
    #bb is a BigBang object
    fsm <- forwardSelectionModels(bb)
    fsm
    names(fsm)
    heatmapModels(fsm, subset=1)
    fsm <- forwardSelectionModels(bb, minFitness=0.9,
    fitnessFunc=bb$galgo$fitnessFunc)
    heatmapModels(fsm, subset=1)
    pcaModels(fsm, subset=1)
    fitnessSplits(bb, chromosomes=list(fsm$models[[1]]))
```

`heatmapModels.BigBang` *Plots models using heatmap plot*

Description

Plots models using heatmap plot.

Usage

```
## S3 method for class 'BigBang':
heatmapModels(O,
        models,
        data=O$data$data,
        geneNames=paste(pad(1:length(O$geneNames), char = "  "), " : ", O$geneNa
        traspose=TRUE,
        subset=NULL,
        main=O$main,
        scale=if (traspose) "column" else "row",
        col=c(rgb(0, 8:0/8, 0), rgb(1:8/8, 0, 0)),
        RowSideColors=NULL,
        ColSideColors=NULL,
        hclustfun=function(x) hclust(x, method = "ward"),
        ...)
```

Arguments

`models`	The models(chromosomes) to plot. It can be a chromosome list or models resulted from `forwardSelectionModel`.
`data`	Data if this is not provided in `$data$data` from the `BigBang` object.
`geneNames`	Names for the genes. The default uses the `$geneNames` from `BigBang` object.
`traspose`	Traspose the data (for display and data restrictions).
`subset`	To limit the usage of `models`.
`scale,col,RowSideColors,ColSideColors`	Heatmap parameters. Provided for compatibility. If col is -1,-2,-3, or -4, standard microarray colors are used. If length(col)==3, these three colours are used to build a gradient.
`geneColors`	A list of specific RowSideColors parameter for every model.
`sampleColors`	Colors for samples.
`hclustfun`	Function to heatmap. The default use “ward” method. Use `hclustfun=hclust` to restore the original heatmap behaviour.
`...`	Other parameters for `heatmap` function.

Value

Returns nothing.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`. `*plot()`, `*forwardSelectionModels()`, `heatmap`().

Examples

```
#bb is a BigBang object
heatmapModels(bb, bb$bestChromosomes[1])

fsm <- forwardSelectionModels(bb)
fsm
names(fsm)
heatmapModels(fsm, subset=1)
fsm <- forwardSelectionModels(bb, minFitness=0.9,
fitnessFunc=bb$galgo$fitnessFunc)
heatmapModels(fsm, subset=1)
pcaModels(fsm, subset=1)
fitnessSplits(bb, chromosomes=list(fsm$models[[1]]))
```

pcaModels.BigBang *Plots models in principal components space*

Description

Plots models in principal components space.

Usage

```
## S3 method for class 'BigBang':
pcaModels(O,
        models,
        data=O$data$data,
        traspose=FALSE,
        center=TRUE,
        scale=TRUE,
        subset=NULL,
        main=O$main,
        sampleColors=NULL,
        sampleNames=NULL,
        npc=4,
        pch=19,
        gap=0,
        classes=NULL,
        ...)
```

Arguments

`models`	The models(chromosomes) to plot. It can be a chromosome list or models resulted from `forwardSelectionModel`.
`data`	Data if this is not provided in `$data$data` from the `BigBang` object.
`traspose`	Traspose the data (for display and data restrictions).
`subset`	To limit the usage of `models`.
`center`	Logical value indicating whether scalling by genes to mean 0. See `prcomp`.
`scale`	Logical value indicating whether scalling by genes to 1 variance. See `prcomp`.
`main,gap,pch`	Plot parameters (method pairs). If `pch==NULL`, `sampleColors` are used instead.
`sampleColors`	Colors for samples.
`sampleNames`	To plot the samples names. Use the variable `$sampleNames` to from the `BigBang` object.
`classes`	Sample classes. The default is using `$classes` from bigbang object.
`...`	Other parameters for `pairs` (or `plot`) function.

Value

Returns the results of prcomp in a list.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`., `*plot()`, `*forwardSelectionModels()`, `prcomp`(), `princomp`().

Examples

```
#bb is a BigBang object
pcaModels(bb, bb$bestChromosomes[1])

fsm <- forwardSelectionModels(bb)
fsm
names(fsm)
heatmapModels(fsm, subset=1)
fsm <- forwardSelectionModels(bb, minFitness=0.9,
fitnessFunc=bb$galgo$fitnessFunc)
heatmapModels(fsm, subset=1)
pcaModels(fsm, subset=1)
fitnessSplits(bb, chromosomes=list(fsm$models[[1]]))
```

`mergeBangs.BigBang` *Merges the information from other BigBang objects*

Description

Merges the information from other BigBang objects. It is assumed that all `BigBang` objects compute the same results, thus they are "mergeable". Possibly from parallelization.

Usage

```
## S3 method for class 'BigBang':
mergeBangs(mb, ..., clone=FALSE)
```

Arguments

`...`	List of `BigBang` objects.
`clone`	Logical. Specify if the original object must be cloned before merging.

Value

Returns nothing. However, the original `BigBang` object has been modified adding the others `BigBang` objects.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`.

Examples

```
#bb, bbmachine2, bbmachine3 are BigBang objects
bb
plot(bb)
mergeBangs(bb, bbmachine2, bbmachine3)
bb       # accumulated solutions
plot(bb) # accumulated solutions
```

`addRandomSolutions.BigBang`

Adds random pre-existed solutions

Description

Adds random pre-existed solutions. The purpose is to “simulate” new solutions in order to see gene stability and gene frequency. The results are biased but can be used to estimate the how many more solutions are needed to stabilize certain number of genes.

Usage

```
## S3 method for class 'BigBang':
addRandomSolutions(no, n=length(no$bestChromosomes), ...)
```

Arguments

`n`	number of solutions to add

Value

Returns nothing. However, the original `BigBang` object has been modified adding the random solutions.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`.

Examples

```
#bb is a BigBang object
bb
plot(bb)
addRandomSolutions(bb, 1000)
plot(bb) # accumulated solutions
```

`geneImportanceNetwork.BigBang`

Computes the number of times a couple of top-ranked-genes are present in models

---

Description

Computes the number of times top-ranked-genes are present in models.

Usage

```
## S3 method for class 'BigBang':
geneImportanceNetwork(o,
        filter="none",
        subset=TRUE,
        mord=50,
        inc.rank=FALSE,
        inc.index=FALSE,
        ...)
```

Arguments

`filter`	The `BigBang` object can save information about solutions that did not reach the `goalFitness`. `filter=="solutions"` ensures that only chromosomes that reach the `goalFitness` are considered. `fitlter=="none"` take all chromosomes. `filter=="nosolutions"` consider only no-solutions (for comparative purposes).
`subset`	Second level of filter. `subset` can be a vector specifying which filtered chromosomes are used. It can be a logical vector or a numeric vector (indexes in order given by `$bestChromosomes` in `BigBang` object variable). If it is a numeric vector length one, a positive value means take those top chromosomes sorted by fitness, a negative value take those at bottom.
`mord`	The number of "top-ranked-genes" to highlight.
`inc.rank`	Incluye the gene rank in rownames and colnames.
`inc.index`	Incluye the gene index in rownames and colnames.

Value

Returns a matrix with number of overlaps for every top-ranked-gene pairs. The order correspond to rank.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`. `*distanceImportanceNetwork()`.

Examples

```
#bb is a BigBang object
bb
gin <- geneImportanceNetwork(bb)
gin
gin <- geneImportanceNetwork(bbm, mord=5)
gin
```

`assignParallelFile.BigBang` *Assigns a different saveFile value for parallelization*

Description

Assigns a different saveFile value for parallelization. The assignation is done looking for a existing filename with a consecutive number starting from 1. If a filename with a particular consecutive number is not found, it is assigned.

Usage

```
## S3 method for class 'BigBang':
assignParallelFile(bb,
        save=TRUE,
        prefix="parallel-",
        use.random=TRUE,
        compatibilize=TRUE,
        ...)
```

Arguments

`save`	Specify to save the file to "lock" the name and avoid other parallel process to use it. Defaults to true.
`prefix`	Prefix used in the parallel files for identification. Defaults to "parallel-".
`use.random`	Specify if an integer random value between 0 and 9999 is added to avoid duplicated filenames in parallel process started at the same time. Defaults to true.

Value

Returns the file name assigned to the bigbang object for parallelization.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`.

Examples

```
#Initial process:
#load data and configure initial objects, run once
library(galgo)
bb <- configBB.varSel(..., saveFile="bb.parallel.Rdata", ...)
saveObject(bb)
#

#Parallel process:
#run as many process as you want
library(galgo)
loadObject("bb.parallel.Rdata")
assignParallelFile(bb)
blast(bb)
#

#Analysis Process:
library(galgo)
loadObject("bb.parallel.Rdata")
loadParallelFiles(bb)
#
```

loadParallelFiles.BigBang

Load all files saved during the parallelization

Description

Load all files saved during the parallelization.

Usage

```
## S3 method for class 'BigBang':
loadParallelFiles(bb, prefix="parallel-", ...)
```

Arguments

`prefix`	Prefix used in the parallel files for identification. Defaults to "parallel-".

Value

Returns the file names loaded to the bigbang object.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`.

Examples

```
        #Initial process:
        #load data and configure initial objects, run once
        library(galgo)
        bb <- configBB.varSel(..., saveFile="bb.parallel.Rdata", ...)
        saveObject(bb)
        #

        #Parallel process:
        #run as many process as you want
        library(galgo)
        loadObject("bb.parallel.Rdata")
        assignParallelFile(bb)
        blast(bb)
        #

        #Analysis Process:
```

```
library(galgo)
loadObject("bb.parallel.Rdata")
loadParallelFiles(bb)
#
```

---

filterSolution.BigBang

Filters solutions

Description

Filters solutions.

Usage

```
## S3 method for class 'BigBang':
filterSolution(o, filter=c("none", "solutions", "nosolutions"), subset, ...)
```

Arguments

`filter`	The `BigBang` object can save information about solutions that did not reach the `goalFitness`. `filter=="solutions"` ensures that only chromosomes that reach the `goalFitness` are considered. `fitlter=="none"` take all chromosomes. `filter=="nosolutions"` consider only no-solutions (for comparative purposes).
`subset`	Second level of filter. `subset` can be a vector specifying which filtered chromosomes are used. It can be a logical vector or a numeric vector (indexes in order given by `$bestChromosomes` in `BigBang` object variable).

Value

Returns a logical vector indicating which chromosomes are valid for the given filter.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K. http://www.bip.bham.ac.uk/bioinf

References

Goldberg, David E. 1989 *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. ISBN: 0201157675

See Also

For more information see `BigBang`.

Examples

```
ok <- filterSolution(bb, filter="solutions", subset=1:100)
sum(ok)
```

geneBackwardElimination

Searches for shorter or better models using backward elimination strategy

Description

Searches for shorter or better models using backward elimination strategy. Recursively eliminates variables/genes from a chromosome one by one computing the fitness function. This function is specially designed to be used in the `BigBang` object and for variable selection problems.

Usage

```
geneBackwardElimination(chr, bigbang, result, minChromosomeSize, fitnessFunc, fi
```

Arguments

`chr`	Original chromosome object (or numeric vector).
`bigbang`	The `BigBang` object to be used to call the fitness function.
`result`	The type of result needed. `"highest"` returns the visited chromosome whose fitness was highest. Ties are resolved using the shortest chromosome and finally by random. `"shortest"` returns the visited chromosome whose length was minimum and fitness greather than or equal to the original. Ties are resolved by highest fitness and finally by random. `"visited"` returns a list of all visited chromosomes. `"selected"` only the chromosomes with fitness greather than or equal to original fitness.
`minChromosomeSize`	The minimum possible size of a chromosome. The default is 2.
`fitnessFunc`	The fitness function used to evaluate the chromosomes. The default is the usage of `bigbang$galgo$fitnessFunc`.
`fitnessAid`	To avoid local minima, `fitnessAid` is an amount to be reduced to original fitness in order to try search for better fitness. When it is negative, it is interpreted as percentage value to reduce from the original fitness. If `fitnessAid` is positive, it is substracted from original fitness.

Details

Removes one gene/variable at the time and compute the fitness. If the fitness is greather than or equal to original “reduced” fitness, another attempt to remove other variable will be performed. The result might be a reduced chromosome with same or better fitness.

Value

A chromosome when `result=="highest" or result=="smallest"` and a data frame otherwise.

Author(s)

Victor Trevino

See Also

BigBang, robustGeneBackwardElimination.

Examples

---

robustGeneBackwardElimination

Searches for shorter or better models using backward elimination strategy

---

Description

Searches for shorter or better models using backward elimination strategy. Recursively eliminates variables/genes from a chromosome one by one computing the fitness function. This function is specially designed to be used in the `BigBang` object and for variable selection problems.

Usage

```
robustGeneBackwardElimination(chr, bigbang, result, minChromosomeSize, fitnessFu
```

Arguments

`chr`	Original chromosome object (or numeric vector).
`bigbang`	The `BigBang` object to be used to call the fitness function.
`result`	The type of result needed. `"highest"` returns the visited chromosome whose fitness was highest. Ties are resolved using the shortest chromosome and finally by random. `"shortest"` returns the visited chromosome whose length was minimum and fitness greather than or equal to the original. Ties are resolved by highest fitness and finally by random. `"visited"` returns a list of all visited chromosomes. `"selected"` only the chromosomes with fitness greather than or equal to original fitness.
`minChromosomeSize`	The minimum possible size of a chromosome. The default is 2.
`fitnessFunc`	The fitness function used to evaluate the chromosomes. The default is the usage of `bigbang$data$modelSelectionFunc`.
`fitnessAid`	To avoid local minima, `fitnessAid` is an amount to be reduced to original fitness in order to try search for better fitness. When it is negative, it is interpreted as percentage value to reduce from the original fitness. If `fitnessAid` is positive, it is substracted from original fitness.

Details

Removes one gene/variable at the time and compute the fitness. If the fitness is greather than or equal to original “reduced” fitness, another attempt to remove other variable will be performed. The result might be a reduced chromosome with same or better fitness.

Value

A chromosome when `result=="highest" or result=="smallest"` and a data frame otherwise.

Author(s)

Victor Trevino

See Also

`BigBang`, `geneBackwardElimination`.

Examples

`modelSelection` *Function used to evaluate a fitness function in many train-test sets*

Description

Function used to evaluate a fitness function in many train-test sets

Usage

```
modelSelection(chr, parent, splits = 1:length(parent$data$splitTrain), set = par
```

Arguments

`chr`	Chromosome or vector object.
`parent`	Parent object, commonly BigBang object.
`splits`	Which sets of splits will be used to compute the fitness function. Default to all splits defined in `parent$data$splitTrain`.
`set`	Weigths used in training and test sets. Vector of two values. The first is the weight of train error. The second is the weight of test error. The default value is taken from parent$data$testErrorWeights whose default is c(0,1) (considering only test error).

Value

A vector with the fitness computed for each split weighted according to `set` parameter.

Note

This function is designed to be used in forwardSelectionModels

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K.

See Also

forwardSelectionModels

Examples

---

as.list.Object *Convert a variable of class Object to a list*

Description

Object variables behave as lists, however they are really enviroments. Sometimes it is necesary to use the variable as a list instead of an Object. This function converts the Object to a list.

Usage

```
as.list(x, ...)
```

Arguments

x	Variable of class Object

Value

Returns a list with values equivalent to the Object.

Note

Values that contain functions will be assigned to .GlobalEnv enviroment.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K.

See Also

as.list

Examples

```
xO <- Object()
xO$var = "hello"
class(xO)
xOL <- as.list(xO)
xOL
class(xOL)
```

`unObject` *Converts variables from class Object (and derived classes) to list*

Description

Converts objects derived from class Object to a list object that preserve the properties (data) and can be accessed using the same R syntax. It is primarly used to explore the data or to save the object as an R object independent of the original methods.

Usage

```
unObject(x)
```

Arguments

`x`	Variable of class Object (or list containing some Object).

Details

In R.oo package, all objects are internally represented as enviroment objects (see R.oo package) to give the "by value" functionality. However, this representation is not suitable to save, retrive or explore the data as easy as common objects in R. This method converts an object derived from class Object to a common list object preserving all data except the original methods. It is very useful when an object of class Object contains other objects derived from same class Object.

Value

Return a list containg all values of the object. If `x` contains a list or other Object, these are represented also as a list.The original class of the object is stored in "Class." value.

Warning

The CPU time consumed by this method depends on the complexity of `x`. It is commonly very fast but can be a nuisance when `x` contains many nested objects of class Object (or many lists containing Objects).

Note

If properties (values) inside an object Object contains a function object, the enviroment is set to `.GlobalEnv` for convenience. This method is also implemented for list object because it could contain another Object.

Author(s)

Victor Trevino

See Also

`Object`, `S3Method`, `reObject`.

Examples

```
library(R.oo)  # needed library
o <- Object()
o$x = 1
o$y = 2
o$x
o$y
o
class(o)
names(o)
uo <- unObject(o)
uo
```

`reObject` *Creates proper extended Object from a list obtained by unObject*

Description

Rebuilds an object to its original class from a list which was usually obtained using unObject. The original class is deduced using the `Class.` value and its S3 constructor will be called using all other values as properties.

Usage

```
reObject(x, showStructure = 0)
```

Arguments

`x`	The list attempted to convert to its original Object.
`showStructure`	Flag to show/debug the conversion course. It can be 1, 2 or 0.

Details

The original class (`x$Class.` value) is called without any parameter, then all properties (names) in the list are set using `assign`. The procedure is recursive called if an object of class list is found inside x. If the original object was extended from Object, this object have to be already defined using S3 methodology, otherwise an error would occur.

Value

Object of original class given by `x$Class.`

Warning

It could take some seconds for large and/or complex objects.

Note

It is very important that if the original class was extendend from Object, this class and its methods are already defined, otherwise unexpected behaviour and/or errors would occur.

Author(s)

Victor Trevino

See Also

`unObject`.

Examples

```
library(R.oo)  # needed library
o <- Object()
o$x = 1
o$y = 2
o$x
o$y
o
class(o)
names(o)
uo <- unObject(o)
uo
x <- reObject(uo)
class(x)
names(x)
x$x
x$y


### saving/retriving
library(R.oo)
o <- Object()
o$x = 1
o$y = 2
uo <- unObject(o)
save(uo, file="uo.Rdata")
### perhaps other session here
library(R.oo)
#if your object requiere other sub-class (extend Object) and/or method definition,
#load it here before using reObject otherwise an error would occur.
load("uo.Rdata")
class(uo)             ## uo now is a list
uo
x <- reObject(uo)
class(x)                ### now x is Object
names(x)
x$x
x$y
x
```

`loadObject` *Load saved data of class Object and use reObject as necessary*

Description

Load the data from a file into the `.GlobalEnv` (or any other enviroment). If variables were converted to a list using `unObject`, this variables are converted to original object using `reObject` method.

Usage

```
loadObject(file, envir)
```

Arguments

`file`	The file to load.
`envir`	The environment to load the data. The default is `.GlobalEnv`.
`verbose`	Displays progress.
`reobjectize`	Specify if reObject method should be called. Defaults to TRUE.
`compatibilize`	Compatibilze chromosomes built on previous versions.

Details

Load the data from a file into the `.GlobalEnv` (or any other enviroment). If variables were converted to a list using `unObject`, this variables are converted to original object using `reObject` method.

Value

A data frame with variable names and class of loaded objects.

Warning

It could take some seconds for large and/or complex objects/files.

Author(s)

Victor Trevino

See Also

`unObject`, `reObject`.

Examples

```
library(R.oo)  # needed library
o <- Object()
o$x = 1
o$y = 2
o$x
o$y
o
class(o)
names(o)
uo <- unObject(o)
uo
class(uo)
```

```
save(uo, file="uo.Rdata")

### perhaps other session here
library(R.oo)
loadObject("uo.Rdata")
uo
class(uo)
# the class is the original from the original object (o in this case)

### equivalent to:
library(R.oo)
load("uo.Rdata")
uo <- reObject(uo)
uo
class(uo)
```

`configBB.VarSel` *Creates and configure all objects needed for a "variable selection for classificacion" problem*

Description

Creates and configure all objects needed for a "variable selection for classificacion" problem. It configures Gene, Chromosome, Niche, World, Galgo and BigBang objects.

Usage

```
configBB.VarSel(
        file=NULL,
        data=NULL,
        classes=NULL,
        train=rep(2/3,333),
        test=1-train,

        main="project",

        classification.method=c("knn","mlhd","svm","nearcent","rpart","nnet","us
        classification.test.error=c(0,1),
        classification.train.error=c("kfolds","splits","loocv","resubstitution")
        classification.train.Ksets=-1, # -1 : max(min(round(13-n/11),n),3) n=sam
        classification.train.splitFactor=2/3,
        classification.rutines=c("C","R"),
        classification.userFitnessFunc=NULL,

        scale=(classification.method[1]

        knn.k=3,
        knn.l=1,
        knn.distance=c("euclidean", "maximum", "manhattan", "canberra", "binary"

        nearcent.method=c("mean","median"),
        svm.kernel=c("radial","polynomial","linear","sigmoid"),
        svm.type=c("C-classification", "nu-classification", "one-classification"
        svm.nu=0.5,
        svm.degree=4,
        svm.cost=1,

        nnet.size=2,
        nnet.decay=5e-4,
        nnet.skip=TRUE,
        nnet.rang=0.1,

        geneFunc=runifInt,
        chromosomeSize=5,
        populationSize=-1,
        niches=2,
        worlds=1,
        immigration=c(rep(0,18),.5,1),
        crossoverPoints=round(chromosomeSize/2,0),
        offspringScaleFactor=1,
        offspringMeanFactor=0.85,
        offspringPowerFactor=2,
        elitism=c(rep(1,9),.5),
        goalFitness=0.90,
        galgoVerbose=20,
        maxGenerations=200,
        minGenerations=10,
        galgoUserData=NULL, # additional user data for galgo

        maxBigBangs=1000,
        maxSolutions=1000,
        onlySolutions=FALSE,
        collectMode="bigbang",
        bigbangVerbose=1,
        saveFile="bigbang.Rdata",
        saveFrequency=50,
        saveVariable="bigbang",
        callBackFuncGALGO=function(...) 1,
        callBackFuncBB=plot,
        callEnhancerFunc=function(chr, parent) NULL,
        saveGeneBreaks=NULL,
        geneNames=NULL,
        sampleNames=NULL,
        bigbangUserData=NULL # additional user data for bigbang
        )
```

Arguments

`file`	The file containing the data. First row should be sample names. First column should be variable names (genes). Second row must be the class for every sample if `classes` is not provided.
`data`	If a file is not provided, `data` is the a data matrix or data frame with samples in columns and genes in rows (with its respective colnames and rownames set). If

`data` is provided, `class` must be specified.

`classes` if a file is not provided, specifies the classes for the data. If the `file` is provided and classes is specified, the second row of the file is considered as data.

`train` A vector of the proportion of random samples to be used as training sets. The number of sets is determined by the length of `train`. The `train+test` should never be greather than 1. All sets are randomly chosen with the same proportion of samples per class than the original sample set.

`test` A vector of the proportion of random samples to be used as testing sets. The number of sets is determined by the length of `train`. All sets are randomly chosen with the same proportion of samples per class than the original sample set.

`main` A string or ID related to your project that will be used in all plots and would help you to distinguish results from different studies.

`classification.method`
The method to be used for classification. The current available methods (in this package) are `"knn", "mlhd", "svm", "nearcent" (nearest centroid), "rpart" (recursive partitioning trees), and "nnet" (neural networks, experimental, not recommendable), "user" is for classification problems but the user provides a specific function.`

`classification.test.error`
Vector of two weights specifing how the fitness function is evaluated to compute the test error. The first value is the weight of training and the second the weight of test. The default is c(0,1) which consider only test error. The sum of this values should be 1.

`classification.train.error`
Specify how the training set is divided to compute the error in the training set (in `evolve` method for `Galgo` object). The fitness function really compute `1-error` where `error` is always computed from the proportion of samples that has been incorrectly classified. `"kfolds"` (k-fold-cross-validation) compute `K` non overlapping sets (`classification.train.Ksets`) attempting to conserve class proportions. `"splits"` compute `K` (`classification.train.Ksets`) random splits. `"loocv"` (leave-one-out-cross-validation) compute `K=training samples`. `"resubstitution"` no folding at all; it is faster and provided for quick overviews.

`classification.train.Ksets`
The number of training set folds/splits. Negative means automatic detection (n=samples, max(min(round(13-n/11),n),3)).

`classification.train.splitFactor`
When `classification.train.error=="splits"`, specifies the proportion of samples used in spliting the training set.

`classification.rutines`
For most of the methods, `R` and `C` code has been provided. `C` code is preferred for performance reason, however finding mistakes is easier in R. Besides, the example code could be used as a guide for new user fitness functions. `"rpart"` has not `C` code. `"svm"` has only some improvments removing redundancy checks.

`classification.userFitnessFunc`
For `classification.method == "user"`, specify the function that would be used to compute the accuracy and class prediction. The required prototype is `function(chr, parent, tr, te, result)` where `chr` is the chromosome to be evaluated, a convertion using `as.numeric` is commonly needed

	to extract the exact values from the chromosome. `parent` would be the `BigBang` object where all their variables are exposed. The fitness function commonly use `parent$data$data`, which has been trasposed. `tr` is the vector of samples (rows) that MUST be used as training and `te` the samples that must be used as test. They can correspond to training and test in the evolution or in any other context (as the computation of the confusion matrix or the forward selection). The fitness function should return the result in two different formats, which is specified in the `result` parameter. `result` is `0` (zero) when the predicted class for the test is required (as an integer, not as a factor) otherwise the it is expected the number of correctly classified samples from the test vector.
`scale`	`TRUE` instruct to scale all rows for zero mean and unitary variance.
`knn.k`	For KNN method, `knn.k` is the number of nearest neighbours to consider.
`knn.l`	For KNN method, `knn.l` is the number
`knn.distance`	The distance to be used in KNN method. Possible values are `"euclidean", "maximum", "manhattan", "canberra", "binary", "minkowski", "pearson", "kendall", "spearman", "absolutepearson","absolutekendal "absolutespearman"` (see `dist` method).
`nearcent.method`	For nearest centroid method, `nearcent.method` specify the method for computing the centroid (`"mean", "median"`).
`svm.kernel`	For SVM (support vector machines) method, specify the kernel method `"radial","polynomial or "sigmoid"` (see `svm` method in `e1071` package).
`svm.type`	For SVM method, specify the type of classificacion.
`svm.nu`	For SVM method and `nu-classification` specify the `nu` value.
`svm.degree`	For SVM method and `polynomial` kernel, specify the degreee value.
`svm.cost`	For SVM method, specify the `C` value (cost).
`nnet.`	Parameters for neural networks classification. See `nnet` package.
`geneFunc`	The function that provides random values for genes. The default is runifInt, which generates a random integer value with a uniform distribution.
`chromosomeSize`	Specify the chromosome size (the number of variables/genes to be included in a model). Defaults to 5. See `Gene` and `Chromosome` objects.
`populationSize`	Specify the number of chromosomes per niche. Defaults is min(20,20+(2000-nrow(data))/400). See `Chromosome` and `Niche` objects.
`niches`	Specify the number of niches. Defaults to 2. See `Niche`, `World` and `Galgo` objects.
`worlds`	Specify the number of worlds. Defaults to 1. See `World` and `Galgo` objects.
`immigration`	Specify the migration criteria.
`crossoverPoints`	Specify the active positions for crossover operator. Defaults to a single point in the middle of the chromosome. See `Niche` object.
`offspringScaleFactor`	Scale factor for offspring generation. Defaults 1. See `Niche` object.
`offspringMeanFactor`	Mean factor for offspring generation. Defaults to 0.85. See `Niche` object.

`offspringPowerFactor`	Power factor for offspring generation. Defaults to 2. See `Niche` object.
`elitism`	Elitism probability/flag/vector. Defaults to c(1,1,1,1,1,1,1,1,1,0.5) (elitism present for 9 generations followed by a 50% chance, then repeated). See `Niche` object.
`goalFitness`	Specify the desired fitness value (fraction of correct classification). Defaults to 0.90. See `Galgo` object.
`galgoVerbose`	`verbose` parameter for `Galgo` object.
`maxGenerations`	Maximum number of generations. Defaults to 200. See `Galgo` object.
`minGenerations`	Minimum number of generations. Defaults to 10. See `Galgo` object.
`galgoUserData`	Additional user data for the `Galgo` object. See `Galgo` object.
`maxBigBangs`	Maximum number of bigbang cycles. Defaults to 1000. See `BigBang` object.
`maxSolutions`	Maximum number of solutions collected. Defaults to 1000. See `BigBang` object.
`onlySolutions`	Save only when a solution is reach. Defaults to FALSE (to use all the information, then a filter can be used afterwards). See `BigBang` object.
`collectMode`	information to collect. Defaults to `"bigbang"`. See `BigBang` object.
`bigbangVerbose`	Verbose flag for `BigBang` object. Defaults to 1. See `BigBang` object.
`saveFile`	File name where the data is saved. Defaults to `NULL` which implies the name is a concatenation of `classification.method`, method specific parameters, `file` and `".Rdata"`. See `BigBang` object.
`saveFrequency`	How often the “current” solutions are saved. Defaults to 50. See `BigBang` object.
`saveVariable`	Internal `R` variable name of the saved file. Defaults to “bigbang”. See `BigBang` object.
`callBackFuncGALGO`	`callBackFunc` for `Galgo` object. See `Galgo` object.
`callBackFuncBB`	`callBackFunc` for `BigBang` object. See `BigBang` object.
`callEnhancerFunc`	`callEnhancerFunc` for `BigBang` object. See `BigBang` object.
`saveGeneBreaks`	`saveGeneBreaks` vector for `BigBang` object. Defaults to `NULL` which means to be computed automatically (recommended). See `BigBang` object.
`geneNames`	The gene (variable) names if they differ from the first column in `file` or `rownames(data)`.
`sampleNames`	The sample names if they differ from first row in `file` or `colnames(data)`.
`bigbangUserData`	Additional user data for `BigBang` object (stored in `$data` variable in `BigBang` object returned).

Details

Wrapper function. Configure all objects from parameters.

Value

A ready to use bigbang object.

*** TO DO: EXPLAIN THE STRUCTURE OF "DATA" ***

Author(s)

Victor Trevino

See Also

`BigBang`.

Examples

```
bb <- configBB.VarSel(...)
bb
blast(bb)
```

configBB.VarSelMisc

Creates and configure all objects needed for a "variable selection" problem

Description

Creates and configure all objects needed for a "variable selection" problem. It configures Gene, Chromosome, Niche, World, Galgo and BigBang objects.

Usage

```
configBB.VarSelMisc(
        file=NULL,
        data=NULL,
        strata=NULL,
        train=rep(2/3,333),
        test=1-train,

        main="project",

        test.error=c(0,1),
        train.error=c("kfolds","splits","loocv","resubstitution"),
        train.Ksets=-1, # -1 : automatic detection : max(min(round(13-n/11),n),3
        train.splitFactor=2/3,
        fitnessFunc=NULL,

        scale=FALSE,

        geneFunc=runifInt,
        chromosomeSize=5,
        populationSize=-1,
        niches=1,
        worlds=1,
        immigration=c(rep(0,18),.5,1),
        crossoverPoints=round(chromosomeSize/2,0),
        offspringScaleFactor=1,
        offspringMeanFactor=0.85,
        offspringPowerFactor=2,
        elitism=c(rep(1,9),.5),
        goalFitness=0.90,
        galgoVerbose=20,
        maxGenerations=200,
        minGenerations=10,
        galgoUserData=NULL, # additional user data for galgo

        maxBigBangs=1000,
        maxSolutions=500,
        onlySolutions=FALSE,
        collectMode="bigbang",
        bigbangVerbose=1,
        saveFile="?.Rdata",
        saveFrequency=50,
        saveVariable="bigbang",
        callBackFuncGALGO=function(...) 1,
        callBackFuncBB=plot,
        callEnhancerFunc=function(chr, parent) NULL,
        saveGeneBreaks=NULL,
        geneNames=NULL,
        sampleNames=NULL,
        bigbangUserData=NULL # additional user data for bigbang
        )
```

Arguments

`file`	The file containing the data. First row should be sample names. First column should be variable names (genes). Second row must be the class or strata for every sample if `strata` is not provided. The strata is used to balance the train-test sets relative to different strata. If there are only one strata, use the same value for all samples.
`data`	If a file is not provided, `data` is the a data matrix or data frame with samples in columns and genes in rows (with its respective colnames and rownames set). If `data` is provided, `strata` must be specified.
`strata`	if a file is not provided, specifies the classes or strata of the data. If the `file` is provided and strata is specified, the second row of the file is considered as data. The strata is used to balance the train-test sets relative to different strata. If there are only one strata, use the same value for all samples.
`train`	A vector of the proportion of random samples to be used as training sets. The number of sets is determined by the length of `train`. The `train+test` should never be greather than 1. All sets are randomly chosen with the same proportion of samples per class than the original sample set.
`test`	A vector of the proportion of random samples to be used as testing sets. The number of sets is determined by the length of `train`. All sets are randomly chosen with the same proportion of samples per class than the original sample set.

`main`	A string or ID related to your project that will be used in all plots and would help you to distinguish results from different studies.
`test.error`	Vector of two weights specifing how the fitness function is evaluated to compute the test error. The first value is the weight of training and the second the weight of test. The default is c(0,1) which consider only test error. The sum of this values should be 1.
`train.error`	Specify how the training set is divided to compute the error in the training set (in `evolve` method for `Galgo` object). `"splits"` compute `K` (`train.Ksets`) random splits. `"loocv"` (leave-one-out-cross-validation) compute `K=training samples`. `"resubstitution"` no folding at all; it is faster and provided for quick overviews.
`train.Ksets`	The number of training set folds/splits. Negative means automatic detection (n=samples, max(min(round(13-n/11),n),3)).
`train.splitFactor`	When `train.error=="splits"`, specifies the proportion of samples used in spliting the training set.
`fitnessFunc`	Specify the function that would be used to compute the accuracy. The required prototype is `function(chr, parent, tr, te, result)` where `chr` is the chromosome to be evaluated. `parent` would be the `BigBang` object where all their variables are exposed. The fitness function commonly use `parent$data$data`, which has been trasposed. `tr` is the vector of samples (rows) that MUST be used as training and `te` the samples that must be used as test.
`scale`	`TRUE` instruct to scale all rows for zero mean and unitary variance.
`geneFunc`	Specify the function that mutate genes. The default is using an integer uniform distribution function (runifInt).
`chromosomeSize`	Specify the chromosome size (the number of variables/genes to be included in a model). Defaults to 5. See `Gene` and `Chromosome` objects.
`populationSize`	Specify the number of chromosomes per niche. Defaults is min(20,20+(2000-nrow(data))/400). See `Chromosome` and `Niche` objects.
`niches`	Specify the number of niches. Defaults to 2. See `Niche`, `World` and `Galgo` objects.
`worlds`	Specify the number of worlds. Defaults to 1. See `World` and `Galgo` objects.
`immigration`	Specify the migration criteria.
`crossoverPoints`	Specify the active positions for crossover operator. Defaults to a single point in the middle of the chromosome. See `Niche` object.
`offspringScaleFactor`	Scale factor for offspring generation. Defaults 1. See `Niche` object.
`offspringMeanFactor`	Mean factor for offspring generation. Defaults to 0.85. See `Niche` object.
`offspringPowerFactor`	Power factor for offspring generation. Defaults to 2. See `Niche` object.
`elitism`	Elitism probability/flag/vector. Defaults to c(1,1,1,1,1,1,1,1,1,0.5) (elitism present for 9 generations followed by a 50% chance, then repeated). See `Niche` object.

`goalFitness`	Specify the desired fitness value (fraction of correct classification). Defaults to 0.90. See `Galgo` object.
`galgoVerbose`	`verbose` parameter for `Galgo` object.
`maxGenerations`	Maximum number of generations. Defaults to 200. See `Galgo` object.
`minGenerations`	Minimum number of generations. Defaults to 10. See `Galgo` object.
`galgoUserData`	Additional user data for the `Galgo` object. See `Galgo` object.
`maxBigBangs`	Maximum number of bigbang cycles. Defaults to 1000. See `BigBang` object.
`maxSolutions`	Maximum number of solutions collected. Defaults to 1000. See `BigBang` object.
`onlySolutions`	Save only when a solution is reach. Defaults to FALSE (to use all the information, then a filter can be used afterwards). See `BigBang` object.
`collectMode`	information to collect. Defaults to `"bigbang"`. See `BigBang` object.
`bigbangVerbose`	Verbose flag for `BigBang` object. Defaults to 1. See `BigBang` object.
`saveFile`	File name where the data is saved. Defaults to `NULL` which implies the name is a concatenation of `classification.method`, method specific parameters, `file` and `".Rdata"`. See `BigBang` object.
`saveFrequency`	How often the “current” solutions are saved. Defaults to 50. See `BigBang` object.
`saveVariable`	Internal `R` variable name of the saved file. Defaults to “bigbang”. See `BigBang` object.
`callBackFuncGALGO`	`callBackFunc` for `Galgo` object. See `Galgo` object.
`callBackFuncBB`	`callBackFunc` for `BigBang` object. See `BigBang` object.
`callEnhancerFunc`	`callEnhancerFunc` for `BigBang` object. See `BigBang` object.
`saveGeneBreaks`	`saveGeneBreaks` vector for `BigBang` object. Defaults to `NULL` which means to be computed automatically (recommended). See `BigBang` object.
`geneNames`	The gene (variable) names if they differ from the first column in `file` or `rownames(data)`.
`sampleNames`	The sample names if they differ from first row in `file` or `colnames(data)`.
`bigbangUserData`	Additional user data for `BigBang` object (stored in `$data` variable in `BigBang` object returned).

Details

Wrapper function. Configure all objects from parameters.

Value

A ready to use bigbang object.

*** TO DO: EXPLAIN THE STRUCTURE OF "DATA" ***

Author(s)

Victor Trevino

See Also

`BigBang`.

Examples

```
bb <- configBB.VarSelMisc(...)
bb
blast(bb)
```

`runifInt` *Generation of random uniform integer values*

Description

Usage

```
runifInt(.O, n, mn, mx)
```

Arguments

`.O`	Gene objct
`n`	Number of random values to generate
`mn`	Minimum value
`mx`	Maximum value

Details

Value

A vector with random values drawn from a uniform distribution.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K.

See Also

`runif`

Examples

`ALL` *Acute Lymphoblastic Leukemia data (Yeoh et. al., 2002) for GALGO package*

Description

Acute Lymphoblastic Leukemia for GALGO package data published by Yeoh et. al. The original 360 pediatric acute leukemia samples were filtered by class. 233 samples are included corresponding to 5 classes EMLLA, HYP+50, MLL, T, and TEL. The Affymetrix microarray HG$_U95Av2containing12,600probesetswerefilt$

Usage

```
data(ALL)
```

Format

The format is: 2,435 Rows : Genes 233 Columns : Samples Row Names : ProbeId Col Names : Samples Id 5 Classes : Lukemia Types: "EMLLA"=E2A-PBX1, "T"=T-ALL, "HYP+50"=Hyperdiploid > 50 Chromosomes, "MLL"=MLL rearragment, and "TEL"=TEL-AML1

Details

ALL data is complmented by ALL.classes which contain the classes for each column sample.

References

Eng-Juh Yeoh, Mary E. Ross, Sheila A. Shurtleff, W. Kent Williams, Divyen Patel, Rami Mahfouz, Fred G. Behm, Susana C. Raimondi, Mary V. Relling, Anami Patel, Cheng Cheng, Dario Campana,, Dawn Wilkins, Xiaodong Zhou, Jinyan Li, Huiqing Liu, Ching-Hon Pui, William E. Evans, Clayton Naeve, Limsoon Wong, and James R. Downing. *Classification, subtype discovery, and prediction of outcome in pediatric acute lymphoblastic leukemia by gene expression profiling*. Cancer Cell. March 2002.

Examples

```
data(ALL)
data(ALL.classes)
```

`ALL.classes` *Acute Lymphoblastic Leukemia data (Yeoh et. al., 2002) for GALGO package*

Description

Acute Lymphoblastic Leukemia for GALGO package data published by Yeoh et. al. The original 360 pediatric acute leukemia samples were filtered by class. 233 samples are included corresponding to 5 classes EMLLA, HYP+50, MLL, T, and TEL. The Affymetrix microarray HG$_U95Av2containing12,600probesetswerefilt$

Usage

```
data(ALL.classes)
```

Format

5 Classes : Lukemia Types: "EMLLA"=E2A-PBX1, "T"=T-ALL, "HYP+50"=Hyperdiploid > 50 Chromosomes, "MLL"=MLL rearragment, and "TEL"=TEL-AML1

Details

ALL.classes is complementary for ALL data which contain the expression values for the genes in all samples.

References

Eng-Juh Yeoh, Mary E. Ross, Sheila A. Shurtleff, W. Kent Williams, Divyen Patel, Rami Mahfouz, Fred G. Behm, Susana C. Raimondi, Mary V. Relling, Anami Patel, Cheng Cheng, Dario Campana,, Dawn Wilkins, Xiaodong Zhou, Jinyan Li, Huiqing Liu, Ching-Hon Pui, William E. Evans, Clayton Naeve, Limsoon Wong, and James R. Downing. *Classification, subtype discovery, and prediction of outcome in pediatric acute lymphoblastic leukemia by gene expression profiling*. Cancer Cell. March 2002.

Examples

```
data(ALL)
data(ALL.classes)
```

`galgo.dist` *Computes the distance in GALGO for KNN based methods*

Description

KNN function does not include other common distances. This function includes more distances computations.

Usage

```
galgo.dist(x, method, p = 2)
```

Arguments

`x`	Matrix to compute distnaces
`method`	Any of `"euclidean", "maximum", "manhattan", "canberra", "binary", "minkowski"`. See `dist` function.
`p`	

Details

Value

A vector class dist.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K.

See Also

`dist`

Examples

`knn.C.predict` *Class prediction using KNN method calling the C code*

Description

Usage

```
knn.C.predict(chr, parent, tr, te, result)
```

Arguments

`chr`	Chromosome. Must be integer, use as.integer().
`parent`	Bigbang object.
`tr`	Sample indexes for training vector. Must be integer, use as.integer().
`te`	Sample indexes for test vector. Must be integer, use as.integer().
`result`	0 indicates to return class prediction, non-zero returns the proportion of samples with same class prediction. Must be integer, use as.integer().

Value

Vector of classes (integer) or numeric value. Depends on `result` argument.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K.

See Also

`knn`

`knn.R.predict` *Class prediction using KNN method calling the R code*

Description

R code is slower than C code. This function is included for debugging and educational purposes.

Usage

```
knn.R.predict(chr, parent, tr, te, result)
```

Arguments

`chr`	Chromosome. Must be integer, use as.integer().
`parent`	Bigbang object.
`tr`	Sample indexes for training vector. Must be integer, use as.integer().
`te`	Sample indexes for test vector. Must be integer, use as.integer().
`result`	0 indicates to return class prediction, non-zero returns the proportion of samples with same class prediction. Must be integer, use as.integer().

Value

Vector of classes (integer) or numeric value. Depends on `result` argument.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K.

See Also

`knn`

`mlhd.C.predict` *Class prediction using Maximum Likelihood Discriminant Functions method calling the C code*

Description

Usage

```
mlhd.C.predict(chr, parent, tr, te, result)
```

Arguments

`chr`	Chromosome. Must be integer, use as.integer().
`parent`	Bigbang object.
`tr`	Sample indexes for training vector. Must be integer, use as.integer().
`te`	Sample indexes for test vector. Must be integer, use as.integer().
`result`	0 indicates to return class prediction, non-zero returns the proportion of samples with same class prediction. Must be integer, use as.integer().

Value

Vector of classes (integer) or numeric value. Depends on `result` argument.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K.

`mlhd.R.predict` *Class prediction using Maximum Likelihood Discriminant Functions method calling the R code*

Description

R code is slower than C code. This function is included for debugging and educational purposes.

Usage

```
mlhd.R.predict(chr, parent, tr, te, result)
```

Arguments

`chr`	Chromosome. Must be integer, use as.integer().
`parent`	Bigbang object.
`tr`	Sample indexes for training vector. Must be integer, use as.integer().
`te`	Sample indexes for test vector. Must be integer, use as.integer().
`result`	0 indicates to return class prediction, non-zero returns the proportion of samples with same class prediction. Must be integer, use as.integer().

Value

Vector of classes (integer) or numeric value. Depends on `result` argument.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K.

`nearcent.C.predict` *Class prediction using the nearest centroid method calling the C code*

Description

Usage

```
nearcent.C.predict(chr, parent, tr, te, result)
```

Arguments

`chr`	Chromosome. Must be integer, use as.integer().
`parent`	Bigbang object.
`tr`	Sample indexes for training vector. Must be integer, use as.integer().
`te`	Sample indexes for test vector. Must be integer, use as.integer().
`result`	0 indicates to return class prediction, non-zero returns the proportion of samples with same class prediction. Must be integer, use as.integer().

Value

Vector of classes (integer) or numeric value. Depends on `result` argument.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K.

`nearcent.R.predict` *Class prediction using the nearest centroid method calling the R code*

Description

R code is slower than C code. This function is included for debugging and educational purposes.

Usage

```
nearcent.R.predict(chr, parent, tr, te, result)
```

Arguments

`chr`	Chromosome. Must be integer, use as.integer().
`parent`	Bigbang object.
`tr`	Sample indexes for training vector. Must be integer, use as.integer().
`te`	Sample indexes for test vector. Must be integer, use as.integer().
`result`	0 indicates to return class prediction, non-zero returns the proportion of samples with same class prediction. Must be integer, use as.integer().

Value

Vector of classes (integer) or numeric value. Depends on `result` argument.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K.

`nnet.R.predict` *Class prediction using the neural networks method calling the R code*

Description

Usage

```
nnet.R.predict(chr, parent, tr, te, result)
```

Arguments

`chr`	Chromosome. Must be integer, use as.integer().
`parent`	Bigbang object.
`tr`	Sample indexes for training vector. Must be integer, use as.integer().
`te`	Sample indexes for test vector. Must be integer, use as.integer().
`result`	0 indicates to return class prediction, non-zero returns the proportion of samples with same class prediction. Must be integer, use as.integer().

Value

Vector of classes (integer) or numeric value. Depends on `result` argument.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K.

`rpart.R.predict` *Class prediction using the recursive tree partiotions method calling the R code*

Description

Usage

```
rpart.R.predict(chr, parent, tr, te, result)
```

Arguments

`chr`	Chromosome. Must be integer, use as.integer().
`parent`	Bigbang object.
`tr`	Sample indexes for training vector. Must be integer, use as.integer().
`te`	Sample indexes for test vector. Must be integer, use as.integer().
`result`	0 indicates to return class prediction, non-zero returns the proportion of samples with same class prediction. Must be integer, use as.integer().

Value

Vector of classes (integer) or numeric value. Depends on `result` argument.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K.

`svm.C.predict` *Class prediction using support vector machines method calling the C/R code*

Description

This function really calls the C function that is provided in svm package. The only difference with svm function is that many checks are removed in order to speed up the process. However, it is responsability of the user use valid values.

Usage

```
svm.C.predict(chr, parent, tr, te, result)
```

Arguments

`chr`	Chromosome. Must be integer, use as.integer().
`parent`	Bigbang object.
`tr`	Sample indexes for training vector. Must be integer, use as.integer().
`te`	Sample indexes for test vector. Must be integer, use as.integer().
`result`	0 indicates to return class prediction, non-zero returns the proportion of samples with same class prediction. Must be integer, use as.integer().

Value

Vector of classes (integer) or numeric value. Depends on `result` argument.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K.

`svm.R.predict` *Class prediction using support vector machines method calling the R code*

Description

This function just call svm R code.

Usage

```
svm.R.predict(chr, parent, tr, te, result)
```

Arguments

`chr`	Chromosome. Must be integer, use as.integer().
`parent`	Bigbang object.
`tr`	Sample indexes for training vector. Must be integer, use as.integer().
`te`	Sample indexes for test vector. Must be integer, use as.integer().
`result`	0 indicates to return class prediction, non-zero returns the proportion of samples with same class prediction. Must be integer, use as.integer().

Value

Vector of classes (integer) or numeric value. Depends on `result` argument.

Author(s)

Victor Trevino. Francesco Falciani Group. University of Birmingham, U.K.

Index

∗Topic **classes**
- Bag, 1
- BigBang, 82
- Chromosome, 13
- Galgo, 66
- Gene, 4
- Niche, 25
- World, 48

∗Topic **datasets**
- ALL, 141
- ALL.classes, 141

∗Topic **hplot**
- plot.BigBang, 93
- plot.Galgo, 81
- plot.Niche, 36
- plot.World, 56

∗Topic **internal**
- activeChromosomeSet.BigBang, 92
- addRandomSolutions.BigBang, 117
- as.double.Chromosome, 20
- as.double.Gene, 9
- as.double.Niche, 32
- as.matrix.Gene, 10
- as.matrix.Niche, 32
- assignParallelFile.BigBang, 119
- best.Galgo, 76
- best.Niche, 40
- best.World, 60
- bestFitness.Galgo, 78
- bestFitness.Niche, 42
- bestFitness.World, 62
- blast.BigBang, 87
- classPredictionMatrix.BigBang, 103
- clone.Chromosome, 16
- clone.Galgo, 69
- clone.Niche, 29
- clone.World, 51
- confusionMatrix.BigBang, 104
- crossover.Niche, 44
- decode.Chromosome, 22
- evaluate.Galgo, 75
- evaluate.Niche, 47
- evaluate.World, 65
- evolve.Galgo, 80
- filterSolution.BigBang, 122
- fitnessSplits.BigBang, 108
- forwardSelectionModels.BigBang, 111
- geneCoverage.BigBang, 110
- geneFrequency.BigBang, 98
- geneImportanceNetwork.BigBang, 118
- geneRankStability.BigBang, 99
- generateRandom.Chromosome, 17
- generateRandom.Galgo, 71
- generateRandom.Gene, 6
- generateRandom.Niche, 30
- generateRandom.World, 52
- genes.Chromosome, 17
- getFitness.Niche, 44
- getFrequencies.BigBang, 91
- heatmapModels.BigBang, 113
- length.Bag, 3
- length.Chromosome, 18
- length.Galgo, 72
- length.Niche, 31
- length.World, 53
- loadParallelFiles.BigBang, 121
- max.Galgo, 77
- max.Niche, 41
- max.World, 61
- maxFitness.Galgo, 79
- maxFitness.Niche, 43
- maxFitness.World, 63
- meanFitness.BigBang, 101
- meanGeneration.BigBang, 97
- mergeBangs.BigBang, 116
- mutate.Chromosome, 21
- mutate.Gene, 7
- mutate.Niche, 39
- newCollection.Chromosome, 23

newCollection.Gene, 11
newCollection.Niche, 37
newCollection.World, 58
newRandomCollection.Chromosome, 24
newRandomCollection.Gene, 12
newRandomCollection.Niche, 38
newRandomCollection.World, 59
offspring.Niche, 45
pcaModels.BigBang, 115
plot.BigBang, 93
plot.Galgo, 81
plot.Niche, 36
plot.World, 56
print.Bag, 2
print.BigBang, 89
print.Chromosome, 20
print.Galgo, 72
print.Gene, 8
print.Niche, 33
print.World, 54
progeny.Niche, 46
progeny.World, 64
refreshStats.Galgo, 74
refreshStats.Niche, 35
refreshStats.World, 55
reInit.Chromosome, 15
reInit.Galgo, 70
reInit.Gene, 6
reInit.Niche, 28
reInit.World, 51
saveObject.BigBang, 88
sensitivityClass.BigBang, 106
specificityClass.BigBang, 107
summary.Bag, 3
summary.BigBang, 90
summary.Chromosome, 19
summary.Galgo, 73
summary.Gene, 10
summary.Niche, 34
summary.World, 55
∗Topic **list**
Bag, 1
∗Topic **methods**
activeChromosomeSet.BigBang, 92
addRandomSolutions.BigBang, 117
as.double.Chromosome, 20
as.double.Gene, 9
as.double.Niche, 32
as.list.Object, 126
as.matrix.Gene, 10
as.matrix.Niche, 32
assignParallelFile.BigBang, 119
best.Galgo, 76
best.Niche, 40
best.World, 60
bestFitness.Galgo, 78
bestFitness.Niche, 42
bestFitness.World, 62
BigBang, 82
blast.BigBang, 87
Chromosome, 13
classPrediction, 102
classPredictionMatrix.BigBang, 103
clone.Chromosome, 16
clone.Galgo, 69
clone.Niche, 29
clone.World, 51
configBB.VarSel, 131
configBB.VarSelMisc, 136
confusionMatrix.BigBang, 104
crossover.Niche, 44
decode.Chromosome, 22
evaluate.Galgo, 75
evaluate.Niche, 47
evaluate.World, 65
evolve.Galgo, 80
filterSolution.BigBang, 122
fitnessSplits.BigBang, 108
forwardSelectionModels.BigBang, 111
Galgo, 66
galgo.dist, 142
Gene, 4
geneBackwardElimination, 123
geneCoverage.BigBang, 110
geneFrequency.BigBang, 98
geneImportanceNetwork.BigBang, 118
geneRankStability.BigBang, 99
generateRandom.Chromosome, 17
generateRandom.Galgo, 71
generateRandom.Gene, 6
generateRandom.Niche, 30
generateRandom.World, 52
genes.Chromosome, 17
getFitness.Niche, 44
getFrequencies.BigBang, 91
heatmapModels.BigBang, 113
knn.C.predict, 143

knn.R.predict, 144
length.Bag, 3
length.Chromosome, 18
length.Galgo, 72
length.Niche, 31
length.World, 53
loadObject, 129
loadParallelFiles.BigBang, 121
max.Galgo, 77
max.Niche, 41
max.World, 61
maxFitness.Galgo, 79
maxFitness.Niche, 43
maxFitness.World, 63
meanFitness.BigBang, 101
meanGeneration.BigBang, 97
mergeBangs.BigBang, 116
mlhd.C.predict, 144
mlhd.R.predict, 145
modelSelection, 125
mutate.Chromosome, 21
mutate.Gene, 7
mutate.Niche, 39
nearcent.C.predict, 146
nearcent.R.predict, 146
newCollection.Chromosome, 23
newCollection.Gene, 11
newCollection.Niche, 37
newCollection.World, 58
newRandomCollection.Chromosome, 24
newRandomCollection.Gene, 12
newRandomCollection.Niche, 38
newRandomCollection.World, 59
Niche, 25
nnet.R.predict, 147
offspring.Niche, 45
pcaModels.BigBang, 115
plot.BigBang, 93
plot.Galgo, 81
plot.Niche, 36
plot.World, 56
print.Bag, 2
print.BigBang, 89
print.Chromosome, 20
print.Galgo, 72
print.Gene, 8
print.Niche, 33
print.World, 54
progeny.Niche, 46
progeny.World, 64
refreshStats.Galgo, 74
refreshStats.Niche, 35
refreshStats.World, 55
reInit.Chromosome, 15
reInit.Galgo, 70
reInit.Gene, 6
reInit.Niche, 28
reInit.World, 51
reObject, 128
robustGeneBackwardElimination, 124
rpart.R.predict, 147
runifInt, 140
saveObject.BigBang, 88
sensitivityClass.BigBang, 106
specificityClass.BigBang, 107
summary.Bag, 3
summary.BigBang, 90
summary.Chromosome, 19
summary.Galgo, 73
summary.Gene, 10
summary.Niche, 34
summary.World, 55
svm.C.predict, 148
svm.R.predict, 149
unObject, 127
World, 48

∗Topic **print**
print.Bag, 2
print.BigBang, 89
print.Chromosome, 20
print.Galgo, 72
print.Gene, 8
print.Niche, 33
print.World, 54
refreshStats.Galgo, 74
refreshStats.Niche, 35
refreshStats.World, 55
summary.Bag, 3
summary.BigBang, 90
summary.Chromosome, 19
summary.Galgo, 73
summary.Gene, 10
summary.Niche, 34
summary.World, 55

∗Topic **programming**
BigBang, 82
Chromosome, 13
Galgo, 66
Gene, 4
Niche, 25
World, 48

⋆as.double, *10*
⋆as.list, *11*, *13*, *18*, *24*, *25*, *30*, *37*, *38*, *53*, *58*, *59*, *71*
⋆best, *25*, *41–44*, *61–63*, *77–79*
⋆bestFitness, *40*, *41*, *43*, *44*, *60*, *61*, *63*, *77*, *79*
⋆classPredictionMatrix, *105*, *106*, *108*
⋆confusionMatrix, *104–106*, *108*
⋆crossover, *25*, *47*
⋆distanceImportanceNetwork, *119*
⋆evaluate, *66*
⋆forwardSelectionModels, *114*, *116*
⋆geneFrequency, *91*
⋆generateRandom, *7*, *8*
⋆genes, *9*, *10*, *19*, *21*, *23*
⋆heatmapModels, *113*
⋆length, *31*
⋆length.Niche, *53*, *72*
⋆length.World, *72*
⋆max, *40*, *42–44*, *60*, *62*, *63*, *77–79*
⋆maxFitness, *40–42*, *60–62*, *77*, *78*
⋆meanFitness, *98*
⋆mutate, *6*, *7*, *22*
⋆newCollection, *13*, *18*, *24*, *25*, *30*, *37*, *38*, *53*, *58*, *59*, *71*
⋆newRandomCollection, *11*, *18*, *24*, *30*, *37*, *53*, *58*, *71*
⋆offspring, *47*
⋆pcaModels, *113*
⋆plot, *109*, *110*, *112–114*, *116*
⋆print, *3*, *11*, *19*, *32*, *34*, *55*, *74*, *90*
⋆progeny, *25*, *45*, *46*
⋆scaling, *45*, *46*
⋆summary, *9*, *20*, *34*, *35*, *54*, *56*, *73*, *74*, *89*
⋆unObject, *11*, *13*, *18*, *24*, *25*, *30*, *37*, *38*, *53*, *58*, *59*, *71*

activeChromosomeSet, *85*
activeChromosomeSet,BigBang-method (*activeChromosomeSet.BigBang*), 92
activeChromosomeSet.BigBang, 92
addCount, *85*
addRandomSolutions, *85*
addRandomSolutions,BigBang-method (*addRandomSolutions.BigBang*), 117
addRandomSolutions.BigBang, 117
ALL, 141
ALL.classes, 141
as.double, *5*, *14*, *27*
as.double,Chromosome-method (*as.double.Chromosome*), 20
as.double,Gene-method (*as.double.Gene*), 9
as.double,Niche-method (*as.double.Niche*), 32
as.double.Chromosome, 20
as.double.Gene, 9
as.double.Niche, 32
as.list, *126*
as.list (*as.list.Object*), 126
as.list.Object, 126
as.matrix, *5*, *27*, *85*
as.matrix,Gene-method (*as.matrix.Gene*), 10
as.matrix,Niche-method (*as.matrix.Niche*), 32
as.matrix.Gene, 10
as.matrix.Niche, 32
assignParallelFile, *85*
assignParallelFile,BigBang-method (*assignParallelFile.BigBang*), 119
assignParallelFile.BigBang, 119

Bag, 1, *2–4*
Bag.length (*length.Bag*), 3
Bag.print (*print.Bag*), 2
Bag.summary (*summary.Bag*), 3
best, *27*, *50*, *68*
best,Galgo-method (*best.Galgo*), 76
best,Niche-method (*best.Niche*), 40
best,World-method (*best.World*), 60
best.Galgo, 76
best.Niche, 40
best.World, 60
bestFitness, *27*, *50*, *68*
bestFitness,Galgo-method (*bestFitness.Galgo*), 78
bestFitness,Niche-method (*bestFitness.Niche*), 42
bestFitness,World-method (*bestFitness.World*), 62
bestFitness.Galgo, 78
bestFitness.Niche, 42
bestFitness.World, 62
BigBang, *5*, *14*, *28*, *50*, *68*, 82, *87–92*, *97–101*, *104–106*, *108–110*, *113*, *114*, *116–122*, *124*, *125*, *136*, *140*
BigBang.activeChromosomeSet (*activeChromosomeSet.BigBang*), 92

BigBang.addRandomSolutions (*addRandomSolutions.BigBang*), 117
BigBang.assignParallelFile (*assignParallelFile.BigBang*), 119
BigBang.blast (*blast.BigBang*), 87
BigBang.classPredictionMatrix (*classPredictionMatrix.BigBang*), 103
BigBang.confusionMatrix (*confusionMatrix.BigBang*), 104
BigBang.filterSolution (*filterSolution.BigBang*), 122
BigBang.fitnessSplits (*fitnessSplits.BigBang*), 108
BigBang.forwardSelectionModels (*forwardSelectionModels.BigBang*), 111
BigBang.geneCoverage (*geneCoverage.BigBang*), 110
BigBang.geneFrequency (*geneFrequency.BigBang*), 98
BigBang.geneImportanceNetwork (*geneImportanceNetwork.BigBang*), 118
BigBang.geneRankStability (*geneRankStability.BigBang*), 99
BigBang.getFrequencies (*getFrequencies.BigBang*), 91
BigBang.heatmapModels (*heatmapModels.BigBang*), 113
BigBang.loadParallelFiles (*loadParallelFiles.BigBang*), 121
BigBang.meanFitness (*meanFitness.BigBang*), 101
BigBang.meanGeneration (*meanGeneration.BigBang*), 97
BigBang.mergeBangs (*mergeBangs.BigBang*), 116
BigBang.pcaModels (*pcaModels.BigBang*), 115
BigBang.plot (*plot.BigBang*), 93
BigBang.print (*print.BigBang*), 89
BigBang.saveObject (*saveObject.BigBang*), 88
BigBang.sensitivityClass (*sensitivityClass.BigBang*), 106
BigBang.specificityClass (*specificityClass.BigBang*), 107
BigBang.summary (*summary.BigBang*), 90
blast, *85*
blast,BigBang-method (*blast.BigBang*), 87
blast.BigBang, 87
buildCount, *85*

Chromosome, *5*, *10*, *11*, 13, *13*, *15–25*, *28*, *38*, *50*, *68*, *86*
Chromosome.as.double (*as.double.Chromosome*), 20
Chromosome.clone (*clone.Chromosome*), 16
Chromosome.decode (*decode.Chromosome*), 22
Chromosome.generateRandom (*generateRandom.Chromosome*), 17
Chromosome.genes (*genes.Chromosome*), 17
Chromosome.length (*length.Chromosome*), 18
Chromosome.mutate (*mutate.Chromosome*), 21
Chromosome.newCollection (*newCollection.Chromosome*), 23
Chromosome.newRandomCollection (*newRandomCollection.Chromosome*), 24
Chromosome.print (*print.Chromosome*), 20
Chromosome.reInit (*reInit.Chromosome*), 15
Chromosome.summary (*summary.Chromosome*), 19
classPrediction, 102
classPredictionMatrix, *85*
classPredictionMatrix,BigBang-method (*classPredictionMatrix.BigBang*), 103
classPredictionMatrix.BigBang, 103
clone, *14*, *27*, *50*, *68*

clone,Chromosome-method (*clone.Chromosome*), 16
clone,Galgo-method (*clone.Galgo*), 69
clone,Niche-method (*clone.Niche*), 29
clone,World-method (*clone.World*), 51
clone.Chromosome, 16
clone.Galgo, 69
clone.Niche, 29
clone.World, 51
computeCount, *85*
configBB.VarSel, *68*, *86*, 131
configBB.VarSelMisc, *68*, *86*, 136
confusionMatrix, *85*
confusionMatrix,BigBang-method (*confusionMatrix.BigBang*), 104
confusionMatrix.BigBang, 104
crossover, *27*
crossover,Niche-method (*crossover.Niche*), 44
crossover.Niche, 44

decode, *14*
decode,Chromosome-method (*decode.Chromosome*), 22
decode.Chromosome, 22
dist, *143*
distanceImportanceNetwork, *85*

evaluate, *27*, *50*, *68*
evaluate,Galgo-method (*evaluate.Galgo*), 75
evaluate,Niche-method (*evaluate.Niche*), 47
evaluate,World-method (*evaluate.World*), 65
evaluate.Galgo, 75
evaluate.Niche, 47
evaluate.World, 65
evolve, *68*
evolve,Galgo-method (*evolve.Galgo*), 80
evolve.Galgo, 80, *87*

filterSolution, *85*
filterSolution,BigBang-method (*filterSolution.BigBang*), 122
filterSolution.BigBang, 122
fitnessSplits, *85*
fitnessSplits,BigBang-method (*fitnessSplits.BigBang*), 108
fitnessSplits.BigBang, 108
formatChromosome, *85*
forwardSelectionModels, *85*, *102*, *126*
forwardSelectionModels,BigBang-method (*forwardSelectionModels.BigBang*), 111
forwardSelectionModels.BigBang, 111

Galgo, *5*, *14*, *28*, *50*, 66, *69–81*, *86*
Galgo.best (*best.Galgo*), 76
Galgo.bestFitness (*bestFitness.Galgo*), 78
Galgo.clone (*clone.Galgo*), 69
galgo.dist, 142
Galgo.evaluate (*evaluate.Galgo*), 75
Galgo.evolve (*evolve.Galgo*), 80
Galgo.generateRandom (*generateRandom.Galgo*), 71
Galgo.length (*length.Galgo*), 72
Galgo.max (*max.Galgo*), 77
Galgo.maxFitness (*maxFitness.Galgo*), 79
Galgo.plot (*plot.Galgo*), 81
Galgo.print (*print.Galgo*), 72
Galgo.refreshStats (*refreshStats.Galgo*), 74
Galgo.reInit (*reInit.Galgo*), 70
Galgo.summary (*summary.Galgo*), 73
Gene, 4, *6–11*, *13*, *14*, *17*, *28*, *50*, *68*, *86*
Gene.as.double (*as.double.Gene*), 9
Gene.as.matrix (*as.matrix.Gene*), 10
Gene.generateRandom (*generateRandom.Gene*), 6
Gene.mutate (*mutate.Gene*), 7
Gene.newCollection (*newCollection.Gene*), 11
Gene.newRandomCollection (*newRandomCollection.Gene*), 12
Gene.print (*print.Gene*), 8
Gene.reInit (*reInit.Gene*), 6
Gene.summary (*summary.Gene*), 10
geneBackwardElimination, 123, *125*
geneCoverage, *85*
geneCoverage,BigBang-method (*geneCoverage.BigBang*), 110
geneCoverage.BigBang, 110

geneFrequency, *85*
geneFrequency,BigBang-method
(geneFrequency.BigBang), 98
geneFrequency.BigBang, 98
geneImportanceNetwork, *85*
geneImportanceNetwork,BigBang-method
(geneImportanceNetwork.BigBang), 118
geneImportanceNetwork.BigBang, 118
geneRankStability, *85*
geneRankStability,BigBang-method
(geneRankStability.BigBang), 99
geneRankStability.BigBang, 99
generateRandom, *5*, *14*, *27*, *50*, *68*
generateRandom,Chromosome-method
(generateRandom.Chromosome), 17
generateRandom,Galgo-method
(generateRandom.Galgo), 71
generateRandom,Gene-method
(generateRandom.Gene), 6
generateRandom,Niche-method
(generateRandom.Niche), 30
generateRandom,World-method
(generateRandom.World), 52
generateRandom.Chromosome, 17
generateRandom.Galgo, 71
generateRandom.Gene, 6
generateRandom.Niche, 30
generateRandom.World, 52
genes, *14*
genes,Chromosome-method
(genes.Chromosome), 17
genes.Chromosome, 17
getFitness, *27*
getFitness,Niche-method
(getFitness.Niche), 44
getFitness.Niche, 44
getFrequencies, *85*
getFrequencies,BigBang-method
(getFrequencies.BigBang), 91
getFrequencies.BigBang, 91

heatmap, *114*
heatmapModels, *85*
heatmapModels,BigBang-method
(heatmapModels.BigBang), 113
heatmapModels.BigBang, 113

knn, *143*, *144*
knn.C.predict, 143
knn.R.predict, 144

length, *2*, *14*, *27*, *50*, *68*
length,Bag-method *(length.Bag)*, 3
length,Chromosome-method
(length.Chromosome), 18
length,Galgo-method
(length.Galgo), 72
length,Niche-method
(length.Niche), 31
length,World-method
(length.World), 53
length.Bag, 3
length.Chromosome, 18
length.Galgo, 72
length.Niche, 31
length.World, 53
list, *2*
loadObject, 129
loadParallelFiles, *85*
loadParallelFiles,BigBang-method
(loadParallelFiles.BigBang), 121
loadParallelFiles.BigBang, 121

max, *27*, *50*, *68*
max,Galgo-method *(max.Galgo)*, 77
max,Niche-method *(max.Niche)*, 41
max,World-method *(max.World)*, 61
max.Galgo, 77
max.Niche, 41
max.World, 61
maxFitness, *27*, *50*, *68*
maxFitness,Galgo-method
(maxFitness.Galgo), 79
maxFitness,Niche-method
(maxFitness.Niche), 43
maxFitness,World-method
(maxFitness.World), 63
maxFitness.Galgo, 79
maxFitness.Niche, 43
maxFitness.World, 63
meanFitness, *85*
meanFitness,BigBang-method
(meanFitness.BigBang), 101
meanFitness.BigBang, 101
meanGeneration, *85*
meanGeneration,BigBang-method
(meanGeneration.BigBang), 97
meanGeneration.BigBang, 97

mergeBangs, *85*
mergeBangs,BigBang-method (*mergeBangs.BigBang*), 116
mergeBangs.BigBang, 116
mlhd.C.predict, 144
mlhd.R.predict, 145
modelSelection, 125
mutate, *5*, *14*, *27*
mutate,Chromosome-method (*mutate.Chromosome*), 21
mutate,Gene-method (*mutate.Gene*), 7
mutate,Niche-method (*mutate.Niche*), 39
mutate.Chromosome, 21, *39*
mutate.Gene, 7, *39*
mutate.Niche, 39

nearcent.C.predict, 146
nearcent.R.predict, 146
newCollection, *5*, *14*, *27*, *50*
newCollection,Chromosome-method (*newCollection.Chromosome*), 23
newCollection,Gene-method (*newCollection.Gene*), 11
newCollection,Niche-method (*newCollection.Niche*), 37
newCollection,World-method (*newCollection.World*), 58
newCollection.Chromosome, 23
newCollection.Gene, 11
newCollection.Niche, 37
newCollection.World, 58
newRandomCollection, *5*, *14*, *27*, *50*
newRandomCollection,Chromosome-method (*newRandomCollection.Chromosome*), 24
newRandomCollection,Gene-method (*newRandomCollection.Gene*), 12
newRandomCollection,Niche-method (*newRandomCollection.Niche*), 38
newRandomCollection,World-method (*newRandomCollection.World*), 59
newRandomCollection.Chromosome, 24
newRandomCollection.Gene, 12
newRandomCollection.Niche, 38
newRandomCollection.World, 59
Niche, *5*, *14*, *24*, 25, *29–35*, *37–48*, *50*, *59*, *64*, *68*, *86*
Niche.as.double (*as.double.Niche*), 32
Niche.as.matrix (*as.matrix.Niche*), 32
Niche.best (*best.Niche*), 40
Niche.bestFitness (*bestFitness.Niche*), 42
Niche.clone (*clone.Niche*), 29
Niche.crossover (*crossover.Niche*), 44
Niche.evaluate (*evaluate.Niche*), 47
Niche.generateRandom (*generateRandom.Niche*), 30
Niche.getFitness (*getFitness.Niche*), 44
Niche.length (*length.Niche*), 31
Niche.max (*max.Niche*), 41
Niche.maxFitness (*maxFitness.Niche*), 43
Niche.mutate (*mutate.Niche*), 39
Niche.newCollection (*newCollection.Niche*), 37
Niche.newRandomCollection (*newRandomCollection.Niche*), 38
Niche.offspring (*offspring.Niche*), 45
Niche.plot (*plot.Niche*), 36
Niche.print (*print.Niche*), 33
Niche.progeny (*progeny.Niche*), 46
Niche.refreshStats (*refreshStats.Niche*), 35
Niche.reInit (*reInit.Niche*), 28
Niche.summary (*summary.Niche*), 34
nnet.R.predict, 147

Object, *1*, *5*, *14*, *16*, *27*, *29*, *49*, *52*, *68*, *69*, *85*, *127*
offspring, *27*
offspring,Niche-method (*offspring.Niche*), 45
offspring.Niche, 45, *64*

pcaModels, *86*
pcaModels,BigBang-method (*pcaModels.BigBang*), 115
pcaModels.BigBang, 115
plot, *27*, *50*, *68*, *86*
plot,BigBang-method (*plot.BigBang*), 93

plot,Galgo-method *(plot.Galgo)*, 81
plot,Niche-method *(plot.Niche)*, 36
plot,World-method *(plot.World)*, 56
plot.BigBang, 93
plot.Galgo, 81
plot.Niche, 36
plot.World, 56
prcomp, *116*
predict, *86*
princomp, *116*
print, *2*, *5*, *14*, *27*, *50*, *68*, *86*
print,Bag-method *(print.Bag)*, 2
print,BigBang-method *(print.BigBang)*, 89
print,Chromosome-method *(print.Chromosome)*, 20
print,Galgo-method *(print.Galgo)*, 72
print,Gene-method *(print.Gene)*, 8
print,Niche-method *(print.Niche)*, 33
print,World-method *(print.World)*, 54
print.Bag, 2
print.BigBang, 89
print.Chromosome, 20
print.Galgo, 72
print.Gene, 8
print.Niche, 33
print.World, 54
progeny, *28*, *50*
progeny,Niche-method *(progeny.Niche)*, 46
progeny,World-method *(progeny.World)*, 64
progeny.Niche, 46, *64*
progeny.World, 64

refreshStats, *28*, *50*, *68*
refreshStats,Galgo-method *(refreshStats.Galgo)*, 74
refreshStats,Niche-method *(refreshStats.Niche)*, 35
refreshStats,World-method *(refreshStats.World)*, 55
refreshStats.Galgo, 74
refreshStats.Niche, 35
refreshStats.World, 55
reInit, *5*, *14*, *28*, *50*, *68*
reInit,Chromosome-method *(reInit.Chromosome)*, 15
reInit,Galgo-method *(reInit.Galgo)*, 70
reInit,Gene-method *(reInit.Gene)*, 6
reInit,Niche-method *(reInit.Niche)*, 28
reInit,World-method *(reInit.World)*, 51
reInit.Chromosome, 15
reInit.Galgo, 70
reInit.Gene, 6
reInit.Niche, 28
reInit.World, 51
reObject, *127*, 128, *130*
robustGeneBackwardElimination, 124, *124*
rpart.R.predict, 147
runif, *140*
runIfInt, *5*
runifInt, 140

S3Method, *127*
saveObject, *86*
saveObject,BigBang-method *(saveObject.BigBang)*, 88
saveObject.BigBang, 88
scaling, *28*
sensitivityClass, *86*
sensitivityClass,BigBang-method *(sensitivityClass.BigBang)*, 106
sensitivityClass.BigBang, 106
specificityClass, *86*
specificityClass,BigBang-method *(specificityClass.BigBang)*, 107
specificityClass.BigBang, 107
summary, *2*, *5*, *14*, *28*, *50*, *68*, *86*
summary,Bag-method *(summary.Bag)*, 3
summary,BigBang-method *(summary.BigBang)*, 90
summary,Chromosome-method *(summary.Chromosome)*, 19
summary,Galgo-method *(summary.Galgo)*, 73
summary,Gene-method *(summary.Gene)*, 10
summary,Niche-method *(summary.Niche)*, 34
summary,World-method *(summary.World)*, 55
summary.Bag, 3
summary.BigBang, 90
summary.Chromosome, 19

`summary.Galgo`, 73
`summary.Gene`, 10
`summary.Niche`, 34
`summary.World`, 55
`svm.C.predict`, 148
`svm.R.predict`, 149

`unObject`, 127, *129*, *130*

`World`, *5*, *14*, *25*, *28*, 48, *51–65*, *68*, *86*
`World.best` *(`best.World`)*, 60
`World.bestFitness`
    (`bestFitness.World`), 62
`World.clone` *(`clone.World`)*, 51
`World.evaluate` *(`evaluate.World`)*,
    65
`World.generateRandom`
    (`generateRandom.World`), 52
`World.length` *(`length.World`)*, 53
`World.max` *(`max.World`)*, 61
`World.maxFitness`
    (`maxFitness.World`), 63
`World.newCollection`
    (`newCollection.World`), 58
`World.newRandomCollection`
    (`newRandomCollection.World`),
    59
`World.plot` *(`plot.World`)*, 56
`World.print` *(`print.World`)*, 54
`World.progeny` *(`progeny.World`)*, 64
`World.refreshStats`
    (`refreshStats.World`), 55
`World.reInit` *(`reInit.World`)*, 51
`World.summary` *(`summary.World`)*, 55