# Package 'OncoSimulR'

April 15, 2017

**Type** Package

**Title** Forward Genetic Simulation of Cancer Progression with Epistasis

**Version** 2.4.0

**Date** 2016-09-22

**Author** Ramon Diaz-Uriarte [aut, cre],
Mark Taylor [ctb]

**Maintainer** Ramon Diaz-Uriarte <rdiaz02@gmail.com>

**Description** Functions for forward population genetic simulation in asexual populations, with special focus on cancer progression. Fitness can be an arbitrary function of genetic interactions between multiple genes or modules of genes, including epistasis, order restrictions in mutation accumulation, and order effects. Mutation rates can differ between genes, and we can include mutator/antimutator genes (to model mutator phenotypes). Simulations use continuous-time models and can include driver and passenger genes and modules. Also included are functions for: simulating random DAGs of the type found in Oncogenetic Tress, Conjunctive Bayesian Networks, and other tumor progression models; plotting and sampling from single or multiple realizations of the simulations, including single-cell sampling; plotting the parent-child relationships of the clones; generating random fitness landscapes (Rough Mount Fuji, House of Cards, and additive models) and plotting them.

**biocViews** BiologicalQuestion, SomaticMutation

**License** GPL (>= 3)

**URL** https://github.com/rdiaz02/OncoSimul,
https://popmodels.cancercontrol.cancer.gov/gsr/packages/oncosimulr/

**BugReports** https://github.com/rdiaz02/OncoSimul/issues

**Depends** R (>= 3.3.0)

**Imports** Rcpp (>= 0.12.4), parallel, data.table, graph, Rgraphviz, gtools, igraph, methods, RColorBrewer, grDevices, car, dplyr, smatr, ggplot2, ggrepel

**Suggests** BiocStyle, knitr, Oncotree, testthat (>= 1.0.0), rmarkdown, bookdown

**LinkingTo** Rcpp

**VignetteBuilder** knitr

**NeedsCompilation** yes

# R topics documented:

allFitnessEffects . . . 2
evalAllGenotypes . . . 8
example-missing-drivers . . . 12
examplePosets . . . 13
examplesFitnessEffects . . . 14
mcfLs . . . 14
oncoSimulIndiv . . . 15
OncoSimulWide2Long . . . 27
plot.fitnessEffects . . . 28
plot.oncosimul . . . 30
plotClonePhylog . . . 36
plotFitnessLandscape . . . 37
plotPoset . . . 41
poset . . . 42
rfitness . . . 44
samplePop . . . 46
simOGraph . . . 48
to_Magellan . . . 49

**Index** **51**

---

`allFitnessEffects` *Create fitness and mutation effects specification from restrictions, epistasis, and order effects.*

---

## Description

Given one or more of a set of poset restrictions, epistatic interactions, order effects, and genes without interactions, as well as, optionally, a mapping of genes to modules, return the complete fitness specification.

For mutator effects, given one or more of a set of epistatic interactions and genes without interactions, as well as, optionally, a mapping of genes to modules, return the complete specification of how mutations affect the mutation rate.

The output of these functions is not intended for user consumption, but as a way of preparing data to be sent to the C++ code.

## Usage

```
allFitnessEffects(rT = NULL, epistasis = NULL, orderEffects = NULL,
  noIntGenes = NULL, geneToModule = NULL, drvNames = NULL,
  genotFitness = NULL,  keepInput = TRUE)

allMutatorEffects(epistasis = NULL, noIntGenes = NULL,
                   geneToModule = NULL,
                  keepInput =  TRUE)
```

## Arguments

`rT` A restriction table that is an extended version of a poset (see `poset` ). A restriction table is a data frame where each row shows one edge between a parent and a child. A restriction table contains exactly these columns, in this order:

**parent** The identifiers of the parent nodes, in a parent-child relationship. There must be at least on entry with the name "Root".

**child** The identifiers of the child nodes.

**s** A numeric vector with the fitness effect that applies if the relationship is satisfied.

**sh** A numeric vector with the fitness effect that applies if the relationship is not satisfied. This provides a way of explicitly modeling deviatons from the restrictions in the graph, and is discussed in Diaz-Uriarte, 2015.

**typeDep** The type of dependency. Three possible types of relationship exist:

**AND, monotonic, or CMPN** Like in the CBN model, all parent nodes must be present for a relationship to be satisfied. Specify it as "AND" or "MN" or "monotone".

**OR, semimonotonic, or DMPN** A single parent node is enough for a relationship to be satisfied. Specify it as "OR" or "SM" or "semimonotone".

**XOR or XMPN** Exactly one parent node must be mutated for a relationship to be satisfied. Specify it as "XOR" or "xmpn" or "XMPN".

In addition, for the nodes that depend only on the root node, you can use "–" or "-" if you want (though using any of the other three would have the same effects if a node that connects to root only connects to root).

`epistasis` A named numeric vector. The names identify the relationship, and the numeric value is the fitness (or mutator) effect. For the names, each of the genes or modules involved is separated by a ":". A negative sign denotes the absence of that term.

`orderEffects` A named numeric vector, as for `epistasis`. A ">" separates the names of the genes of modules of a relationship, so that "U > Z" means that the relationship is satisfied when mutation U has happened before mutation Z.

`noIntGenes` A numeric vector (optionally named) with the fitness coefficients (or mutator multiplier factor) of genes (only genes, not modules) that show no interactions. These genes cannot be part of modules. But you can specify modules that have no epistatic interactions. See examples and vignette.

Of course, avoid using potentially confusing characters in the names. In particular, "," and ">" are not allowed as gene names.

`geneToModule` A named character vector that allows to match genes and modules. The names are the modules, and each of the values is a character vector with the gene names, separated by a comma, that correspond to a module. Note that modules cannot share genes. There is no need for modules to contain more than one gene. If you specify a geneToModule argument, and you used a restriction table, the `geneToModule` must necessarily contain, in the first position, "Root" (since the restriction table contains a node named "Root"). See examples below.

`drvNames` The names of genes that are considered drivers. This is only used for: a) deciding when to stop the simulations, in case you use number of drivers as a simulation stopping criterion (see `oncoSimulIndiv`); b) for summarization purposes (e.g., how many drivers are mutated); c) in figures. But you need not specifiy anything if you do not want to, and you can pass an empty vector (as `character(0)`). The default has changed with respect to v.2.1.3 and previous:

it used to be to assume that all genes that were not in the noIntGenes were drivers. The fault now is to assume nothing: if you want drvNames you have to specify them.

genotFitness A matrix or data frame that contains explicitly the mapping of genotypes to fitness. For now, we only allow epistasis-like relations between genes (so you cannot code order effects this way).

Genotypes can be specified in two ways:

- As a matrix (or data frame) with g + 1 columns. Each of the first g columns contains a 1 or a 0 indicating that the gene of that column is mutated or not. Column g+ 1 contains the fitness values. This is, for instance, the output you will get from rfitness. If the matrix has all columns named, those will be used for the names of the genes.
- As a two column data frame. The second column is fitness, and the first column are genotypes, given as a character vector. For instance, a row "A, B" would mean the genotype with both A and B mutated.

In all cases, fitness must be >= 0. If any possible genotype is missing, its fitness is assumed to be 1.

keepInput If TRUE, whether to keep the original input. This is only useful for human consumption of the output. It is useful because it is easier to decode, say, the restriction table from the data frame than from the internal representation. But if you want, you can set it to FALSE and the object will be a little bit smaller.

## Details

allFitnessEffects is used for extremely flexible specification of fitness and mutator effects, including posets, XOR relationships, synthetic mortality and synthetic viability, arbitrary forms of epistatis, arbitrary forms of order effects, etc. Please, see the vignette for detailed and commented examples.

allMutatorEffects provide the same flexibility, but without order and posets (this might be included in the future, but I have seen no empirical or theoretical argument for their existence or relevance as of now, so I do not add them to minimize unneeded complexity).

If you use both for simulations in the same call to, say, oncoSimulIndiv, all the genes specified in allMutatorEffects MUST be included in the allFitnessEffects object. If you want to have genes that have no direct effect on fitness, but that affect mutation rate, you MUST specify them in the call to allFitnessEffects, for instance as noIntGenes with an effect of 0.

If you use genotFitness then you cannot pass modules, noIntgenes, epistasis, or rT. This makes sense, because using genotFitness is saying "this is the mapping of genotypes to fitness. Period", so we should not allow further modifications from other terms.

If you use genotFitness you need to be careful when you use Bozic's model (as you get a death rate of 0).

If you use genotFitness note that we force the WT (wildtype) to always be 1 so fitnesses are rescaled.

## Value

An object of class "fitnessEffects" or "mutatorEffects". This is just a list, but it is not intended for human consumption. The components are:

long.rt The restriction table in "long format", so as to be easy to parse by the C++ code.

long.epistasis Ditto, but for the epistasis specification.

`long.orderEffects`
: Ditto for the order effects.

`long.geneNoInt`
: Ditto for the non-interaction genes.

`geneModule`
: Similar, for the gene-module correspondence.

`graph`
: An `igraph` object that shows the restrictions, epistasis and order effects, and is useful for plotting.

`drv`
: The numeric identifiers of the drivers. The numbers correspond to the internal numeric coding of the genes.

`rT`
: If `keepInput` is TRUE, the original restriction table.

`epistasis`
: If `keepInput` is TRUE, the original epistasis vector.

`orderEffects`
: If `keepInput` is TRUE, the original order effects vector.

`noIntGenes`
: If `keepInput` is TRUE, the original noIntGenes.

## Note

Please, note that the meaning of the fitness effects in the McFarland model is not the same as in the original paper; the fitness coefficients are transformed to allow for a simpler fitness function as a product of terms. This differs with respect to v.1. See the vignette for details.

The names of the genes and modules can be fairly arbitrary. But if you try hard you can confuse the parser. For instance, using gene or module names that contain "," or ":", or ">" is likely to get you into trouble. Of course, you know you should not try to use those characters because you know those characters have special meanings to separate names or indicate epistasis or order relationships. Right now, using those characters as names is caught (and result in stopping) if passed as names for noIntGenes.

## Author(s)

Ramon Diaz-Uriarte

## References

Diaz-Uriarte, R. (2015). Identifying restrictions in the order of accumulation of mutations during tumor progression: effects of passengers, evolutionary models, and sampling http://www.biomedcentral.com/1471-2105/16/41/abstract

McFarland, C.~D. et al. (2013). Impact of deleterious passenger mutations on cancer progression. *Proceedings of the National Academy of Sciences of the United States of America\/*, **110**(8), 2910–5.

## See Also

`evalGenotype`, `oncoSimulIndiv`, `plot.fitnessEffects`, `evalGenotypeFitAndMut`, `rfitness`, `plotFitnessLandscape`

## Examples

```
## A simple poset or CBN-like example

cs <-  data.frame(parent = c(rep("Root", 4), "a", "b", "d", "e", "c"),
                  child = c("a", "b", "d", "e", "c", "c", rep("g", 3)),
                  s = 0.1,
                  sh = -0.9,
                  typeDep = "MN")
```

```
cbn1 <- allFitnessEffects(cs)

plot(cbn1)


## A more complex example, that includes a restriction table
## order effects, epistasis, genes without interactions, and moduels
p4 <- data.frame(parent = c(rep("Root", 4), "A", "B", "D", "E", "C", "F"),
                 child = c("A", "B", "D", "E", "C", "C", "F", "F", "G", "G"),
                 s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, 0.2, 0.2, 0.3, 0.3),
                 sh = c(rep(0, 4), c(-.9, -.9), c(-.95, -.95), c(-.99, -.99)),
                 typeDep = c(rep("--", 4),
                     "XMPN", "XMPN", "MN", "MN", "SM", "SM"))

oe <- c("C > F" = -0.1, "H > I" = 0.12)
sm <- c("I:J"  = -1)
sv <- c("-K:M" = -.5, "K:-M" = -.5)
epist <- c(sm, sv)

modules <- c("Root" = "Root", "A" = "a1",
             "B" = "b1, b2", "C" = "c1",
             "D" = "d1, d2", "E" = "e1",
             "F" = "f1, f2", "G" = "g1",
             "H" = "h1, h2", "I" = "i1",
             "J" = "j1, j2", "K" = "k1, k2", "M" = "m1")

set.seed(1) ## for repeatability
noint <- rexp(5, 10)
names(noint) <- paste0("n", 1:5)

fea <- allFitnessEffects(rT = p4, epistasis = epist, orderEffects = oe,
                         noIntGenes = noint, geneToModule = modules)

plot(fea)


## Modules that show, between them,
## no epistasis (so multiplicative effects).
## We specify the individual terms, but no value for the ":".

fnme <- allFitnessEffects(epistasis = c("A" = 0.1,
                                        "B" = 0.2),
                          geneToModule = c("A" = "a1, a2",
                                           "B" = "b1"))

evalAllGenotypes(fnme, order = FALSE, addwt = TRUE)


## Epistasis for fitness and simple mutator effects

fe <- allFitnessEffects(epistasis = c("a : b" = 0.3,
                                      "b : c" = 0.5),
                            noIntGenes = c("e" = 0.1))

fm <- allMutatorEffects(noIntGenes = c("a" = 10,
                                       "c" = 5))
```

```
evalAllGenotypesFitAndMut(fe, fm, order = FALSE)


## Simple fitness effects (noIntGenes) and modules
## for mutators

fe2 <- allFitnessEffects(noIntGenes =
                         c(a1 = 0.1, a2 = 0.2,
                           b1 = 0.01, b2 = 0.3, b3 = 0.2,
                           c1 = 0.3, c2 = -0.2))

fm2 <- allMutatorEffects(epistasis = c("A" = 5,
                                       "B" = 10,
                                       "C" = 3),
                         geneToModule = c("A" = "a1, a2",
                                          "B" = "b1, b2, b3",
                                          "C" = "c1, c2"))

evalAllGenotypesFitAndMut(fe2, fm2, order = FALSE)



## Passing fitness directly, a complete fitness specification
## with a two column data frame with genotypes as character vectors

(m4 <- data.frame(G = c("A, B", "A", "WT", "B"), F = c(3, 2, 1, 4)))
fem4 <- allFitnessEffects(genotFitness = m4)

## Verify it interprets what it should: m4 is the same as the evaluation
## of the fitness effects (note row reordering)
evalAllGenotypes(fem4, addwt = TRUE, order = FALSE)


## Passing fitness directly, a complete fitness specification
## that uses a three column matrix

m5 <- cbind(c(0, 1, 0, 1), c(0, 0, 1, 1), c(1, 2, 3, 5.5))
fem5 <- allFitnessEffects(genotFitness = m5)

## Verify it interprets what it should: m5 is the same as the evaluation
## of the fitness effects
evalAllGenotypes(fem5, addwt = TRUE, order = FALSE)


## Passing fitness directly, an incomplete fitness specification
## that uses a three column matrix

m6 <- cbind(c(1, 1), c(1, 0), c(2, 3))
fem6 <- allFitnessEffects(genotFitness = m6)
evalAllGenotypes(fem6, addwt = TRUE, order = FALSE)



## Plotting a fitness landscape

fe2 <- allFitnessEffects(noIntGenes =
```

```
                          c(a1 = 0.1,
                            b1 = 0.01,
                            c1 = 0.3))

plot(evalAllGenotypes(fe2, order = FALSE))

## same as
plotFitnessLandscape(evalAllGenotypes(fe2, order = FALSE))

## same as
plotFitnessLandscape(fe2)


## Reinitialize the seed
set.seed(NULL)
```

---

`evalAllGenotypes` *Evaluate fitness/mutator effects of one or all possible genotypes.*

---

## Description

Given a fitnessEffects/mutatorEffects description, obtain the fitness/mutator effects of a single or all genotypes.

## Usage

```
evalGenotype(genotype, fitnessEffects, verbose = FALSE, echo = FALSE,
model = "")

evalGenotypeMut(genotype, mutatorEffects, verbose = FALSE, echo = FALSE)

evalAllGenotypes(fitnessEffects, order = FALSE, max = 256, addwt = FALSE,
model = "")

evalAllGenotypesMut(mutatorEffects, max = 256, addwt = FALSE)


evalGenotypeFitAndMut(genotype, fitnessEffects,
                      mutatorEffects, verbose = FALSE, echo = FALSE,
                      model = "")

evalAllGenotypesFitAndMut(fitnessEffects, mutatorEffects,
                          order = FALSE, max = 256, addwt = FALSE,
                          model = "")
```

## Arguments

| | |
|---|---|
| `genotype` | (For `evalGenotype`). A genotype, as a character vector, with genes separated by "," or ">", or as a numeric vector. Use the same integers or characters used in the fitnessEffects object. This is a genotype in terms of genes, not modules. |

| | |
|---|---|
| | Using "," or ">" makes no difference: the sequence is always taken as the order in which mutations occurred. Whether order matters or not is encoded in the `fitnessEffects` object. |
| `fitnessEffects` | A `fitnessEffects` object, as produced by `allFitnessEffects`. |
| `mutatorEffects` | A `mutatorEffects` object, as produced by `allMutatorEffects`. |
| `order` | (For `evalAllGenotypes`). If TRUE, then order matters. If order matters, then generate not only all possible combinations of the genes, but all possible permutations for each combination. |
| `max` | (For `evalAllGenotypes`). By default, no output is shown if the number of possible genotypes exceeds the max. Increase as needed. |
| `addwt` | (For `evalAllGenotypes`). Add the wildtype (no mutations) explicitly? |
| `model` | Either nothing (the default) or "Bozic". If "Bozic" then the fitness effects contribute to decreasing the Death rate. Otherwise Birth rate is shown (and labeled as Fitness). |
| `verbose` | (For `evalGenotype`). If set to TRUE, print out the individual terms that are added to 1 (or subtracted from 1, if `model` is "Bozic"). |
| `echo` | (For `evalGenotype`). If set to TRUE, show the input genotype and print out a message with the death rate or fitness value. Useful for some examples, as shown in the vignette. |

## Value

For `evalGenotype` either the value of fitness or (if `verbose = TRUE`) the value of fitness and its individual components.

For `evalAllGenotypes` a data frame with two columns, the Genotype and the Fitness (or Death Rate, if Bozic). The notation for the Genotype colum is a follows: when order does not matter, a comma "," separates the identifiers of mutated genes. When order matters, a genotype shown as "x > y _ z" means that a mutation in "x" happened before a mutation in "y"; there is also a mutation in "z" (which could have happened before or after either of "x" or "y"), but "z" is a gene for which order does not matter. In all cases, a "WT" denotes the wild-type (or, actually, the genotype without any mutations).

If you use both `fitnessEffects` and `mutatorEffects` in a call, all the genes specified in `mutatorEffects` MUST be included in the `fitnessEffects` object. If you want to have genes that have no direct effect on fitness, but that affect mutation rate, you MUST specify them in the call to `fitnessEffects`, for instance as `noIntGenes` with an effect of 0.

## Note

Fitness is used in a slight abuse of the language. Right now, mutations contribute to the birth rate for all models except Bozic, where they modify the death rate. The general expression for fitness is the usual multiplicative one of $\prod(1 + s_i)$, where each $s_i$ refers to the fitness effect of the given gene. When dealing with death rates, we use $\prod(1 - s_i)$.

Modules are, of course, taken into account if present (i.e., fitness is specified in terms of modules, but the genotype is specified in terms of genes).

About the naming. This is the convention used: "All" means we will go over all possible genotypes. A function that ends as "Genotypes" returns only fitness effects (for backwards compatibility and because mutator effects are not always used). A function that ends as "Genotype(s)Mut" returns only the mutator effects. A function that ends as "FitAndMut" will return both fitness and mutator effects.

Functions that return ONLY fitness or ONLY mutator effects are kept as separate functions because they free you from specifyin mutator/fitness effects if you only want to play with one of them.

## Author(s)

Ramon Diaz-Uriarte

## See Also

allFitnessEffects.

## Examples

```
# A three-gene epistasis example
sa <- 0.1
sb <- 0.15
sc <- 0.2
sab <- 0.3
sbc <- -0.25
sabc <- 0.4

sac <- (1 + sa) * (1 + sc) - 1

E3A <- allFitnessEffects(epistasis =
                           c("A:-B:-C" = sa,
                             "-A:B:-C" = sb,
                             "-A:-B:C" = sc,
                             "A:B:-C" = sab,
                             "-A:B:C" = sbc,
                             "A:-B:C" = sac,
                             "A : B : C" = sabc)
                                                )

evalAllGenotypes(E3A, order = FALSE, addwt = FALSE)
evalAllGenotypes(E3A, order = FALSE, addwt = TRUE,  model = "Bozic")

evalGenotype("B, C", E3A, verbose = TRUE)

## Order effects and modules
ofe2 <- allFitnessEffects(orderEffects = c("F > D" = -0.3, "D > F" = 0.4),
                          geneToModule =
                              c("Root" = "Root",
                                "F" = "f1, f2, f3",
                                "D" = "d1, d2") )

evalAllGenotypes(ofe2, order = TRUE, max = 325)[1:15, ]

## Next two are identical
evalGenotype("d1 > d2 > f3", ofe2, verbose = TRUE)
evalGenotype("d1 , d2 , f3", ofe2, verbose = TRUE)

## This is different
evalGenotype("f3 , d1 , d2", ofe2, verbose = TRUE)
## but identical to this one
evalGenotype("f3 > d1 > d2", ofe2, verbose = TRUE)
```

```
## Restrictions in mutations as a graph. Modules present.

p4 <- data.frame(parent = c(rep("Root", 4), "A", "B", "D", "E", "C", "F"),
                 child = c("A", "B", "D", "E", "C", "C", "F", "F", "G", "G"),
                 s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, 0.2, 0.2, 0.3, 0.3),
                 sh = c(rep(0, 4), c(-.9, -.9), c(-.95, -.95), c(-.99, -.99)),
                 typeDep = c(rep("--", 4),
                     "XMPN", "XMPN", "MN", "MN", "SM", "SM"))
fp4m <- allFitnessEffects(p4,
                          geneToModule = c("Root" = "Root", "A" = "a1",
                              "B" = "b1, b2", "C" = "c1",
                              "D" = "d1, d2", "E" = "e1",
                              "F" = "f1, f2", "G" = "g1"))

evalAllGenotypes(fp4m, order = FALSE, max = 1024, addwt = TRUE)[1:15, ]

evalGenotype("b1, b2, e1, f2, a1", fp4m, verbose = TRUE)

## Of course, this is identical; b1 and b2 are same module
## and order is not present here

evalGenotype("a1, b2, e1, f2", fp4m, verbose = TRUE)

evalGenotype("a1 > b2 > e1 > f2", fp4m, verbose = TRUE)

## We can use the exact same integer numeric id codes as in the
##   fitnessEffects geneModule component:

evalGenotype(c(1L, 3L, 7L, 9L), fp4m, verbose = TRUE)



## Epistasis for fitness and simple mutator effects

fe <- allFitnessEffects(epistasis = c("a : b" = 0.3,
                                      "b : c" = 0.5),
                        noIntGenes = c("e" = 0.1))

fm <- allMutatorEffects(noIntGenes = c("a" = 10,
                                       "c" = 5))

evalAllGenotypesFitAndMut(fe, fm, order = "FALSE")


## Simple fitness effects (noIntGenes) and modules
## for mutators

fe2 <- allFitnessEffects(noIntGenes =
                         c(a1 = 0.1, a2 = 0.2,
                           b1 = 0.01, b2 = 0.3, b3 = 0.2,
                           c1 = 0.3, c2 = -0.2))

fm2 <- allMutatorEffects(epistasis = c("A" = 5,
                                       "B" = 10,
                                       "C" = 3),
                         geneToModule = c("A" = "a1, a2",
                                          "B" = "b1, b2, b3",
```

```
                                          "C" = "c1, c2"))

## Show only all the fitness effects
evalAllGenotypes(fe2, order = FALSE)

## Show only all mutator effects
evalAllGenotypesMut(fm2)

## Show all fitness and mutator
evalAllGenotypesFitAndMut(fe2, fm2, order = FALSE)

## This is probably not what you want
try(evalAllGenotypesMut(fe2))
## ... nor this
try(evalAllGenotypes(fm2))

## Show the fitness effect of a specific genotype
evalGenotype("a1, c2", fe2, verbose = TRUE)

## Show the mutator effect of a specific genotype
evalGenotypeMut("a1, c2", fm2, verbose = TRUE)

## Fitness and mutator of a specific genotype
evalGenotypeFitAndMut("a1, c2", fe2, fm2, verbose = TRUE)

## This is probably not what you want
try(evalGenotype("a1, c2", fm2, verbose = TRUE))

## Not what you want either
try(evalGenotypeMut("a1, c2", fe2, verbose = TRUE))
```

---

`example-missing-drivers` *An example where there are intermediate missing drivers.*

---

## Description

An example where there are intermediate missing drivers. This is fictitious and I've never seen it. But it is here to check plots work even if there are no cases of some intermediate value of drivers (2 in this case). b11 contains the full, original data, whereas b12 contains the same data where there are no cases with exactly 2 drivers.

## Usage

```
data("ex_missing_drivers_b11"); data("ex_missing_drivers_b12")
```

## Format

Two objects of class "oncosimul".

## See Also

`plot.oncosimul`

### Examples

```
data(ex_missing_drivers_b11)
plot(ex_missing_drivers_b11, type = "line")
dev.new()
data(ex_missing_drivers_b12)
plot(ex_missing_drivers_b12, type = "line")
```

---

## examplePosets *Example posets*

---

### Description

Some example posets. For simplicity, all the posets are in a single list. You can access each poset by accessing each element of the list. The first digit or pair of digits denotes the number of nodes.

Poset 1101 is the same as the one in Gerstung et al., 2009 (figure 2A, poset 2). Poset 701 is the same as the one in Gerstung et al., 2011 (figure 2B, left, the pancreatic cancer poset). Those posets were entered manually at the command line: see poset.

### Usage

```
data("examplePosets")
```

### Format

The format is: List of 13 $ p1101: num [1:10, 1:2] 1 1 3 3 3 7 7 8 9 10 ... $ p1102: num [1:9, 1:2] 1 1 3 3 3 7 7 9 10 2 ... $ p1103: num [1:9, 1:2] 1 1 3 3 3 7 7 8 10 2 ... $ p1104: num [1:9, 1:2] 1 1 3 3 7 7 9 2 10 2 ... $ p901 : num [1:8, 1:2] 1 2 4 5 7 8 5 1 2 3 ... $ p902 : num [1:6, 1:2] 1 2 4 5 7 5 2 3 5 6 ... $ p903 : num [1:6, 1:2] 1 2 5 7 8 1 2 3 6 8 ... $ p904 : num [1:6, 1:2] 1 4 5 5 1 7 2 5 8 6 ... $ p701 : num [1:9, 1:2] 1 1 1 1 2 3 4 4 5 2 ... $ p702 : num [1:6, 1:2] 1 1 1 1 2 4 2 3 4 5 ... $ p703 : num [1:6, 1:2] 1 1 1 1 3 5 2 3 4 5 ... $ p704 : num [1:6, 1:2] 1 1 1 1 4 5 2 3 4 5 ... $ p705 : num [1:6, 1:2] 1 2 1 1 1 2 2 5 4 6 ...

### Source

Gerstung et al., 2009. Quantifying cancer progression with conjunctive Bayesian networks. *Bioinformatics*, 21: 2809–2815.

Gerstung et al., 2011. The Temporal Order of Genetic and Pathway Alterations in Tumorigenesis. *PLoS ONE*, 6.

### See Also

poset

### Examples

```
data(examplePosets)

## Plot all of them
par(mfrow = c(3, 5))

invisible(sapply(names(examplePosets),
                 function(x) {plotPoset(examplePosets[[x]],
```

```
                    main = x,
                    box = TRUE)}))
```

---

examplesFitnessEffects *Examples of fitness effects*

---

## Description

Some examples fitnessEffects objects. This is a collection, in a list, of most of the fitnessEffects created (using allFitnessEffects) for the vignette. See the vignette for descriptions and references.

## Usage

```
data("examplesFitnessEffects")
```

## Format

The format is a list of fitnessEffects objects.

## See Also

allFitnessEffects

## Examples

```
data(examplesFitnessEffects)

plot(examplesFitnessEffects[["fea"]])

evalAllGenotypes(examplesFitnessEffects[["cbn1"]], order = FALSE)
```

---

mcfLs *mcfLs simulation from the vignette*

---

## Description

Trimmed output from the simulation mcfLs in the vignette. This is a somewhat long run, and we have stored here the object (after trimming the Genotype matrix) to allow for plotting it.

## Usage

```
data("mcfLs")
```

## Format

An object of class "oncosimul2". A list.

## See Also

plot.oncosimul

## Examples

```
data(mcfLs)

plot(mcfLs, addtot = TRUE, lwdClone = 0.9, log = "")
summary(mcfLs)
```

---

oncoSimulIndiv *Simulate tumor progression for one or more individuals, optionally returning just a sample in time.*

---

## Description

Simulate tumor progression including possible restrictions in the order of driver mutations. Optionally add passenger mutations. Simulation is done using the BNB algorithm of Mather et al., 2012.

## Usage

```
oncoSimulIndiv(fp, model = "Exp",
               numPassengers = 0, mu = 1e-6, muEF = NULL,
               detectionSize = 1e8, detectionDrivers = 4,
               detectionProb = NA,
               sampleEvery = ifelse(model %in% c("Bozic", "Exp"), 1,
                            0.025),
               initSize = 500, s = 0.1, sh = -1,
               K = initSize/(exp(1) - 1), keepEvery = sampleEvery,
               minDetectDrvCloneSz = "auto",
               extraTime = 0,
               finalTime = 0.25 * 25 * 365, onlyCancer = TRUE,
               keepPhylog = FALSE,
               mutationPropGrowth = ifelse(model == "Bozic",
                                           FALSE, TRUE),
               max.memory = 2000, max.wall.time = 200,
               max.num.tries = 500,
               errorHitWallTime = TRUE,
               errorHitMaxTries = TRUE,
               verbosity = 0,
               initMutant = NULL,
               seed = NULL)

oncoSimulPop(Nindiv, fp, model = "Exp", numPassengers = 0, mu = 1e-6,
               muEF = NULL,
               detectionSize = 1e8, detectionDrivers = 4,
               detectionProb = NA,
               sampleEvery = ifelse(model %in% c("Bozic", "Exp"), 1,
                            0.025),
```

```
                  initSize = 500, s = 0.1, sh = -1,
                  K = initSize/(exp(1) - 1), keepEvery = sampleEvery,
                  minDetectDrvCloneSz = "auto",
                  extraTime = 0,
                  finalTime = 0.25 * 25 * 365, onlyCancer = TRUE,
                  keepPhylog = FALSE,
                  mutationPropGrowth = ifelse(model == "Bozic",
                                                  FALSE, TRUE),
                  max.memory = 2000, max.wall.time = 200,
                  max.num.tries = 500,
                  errorHitWallTime = TRUE,
                  errorHitMaxTries = TRUE,
                  initMutant = NULL,
                  verbosity = 0,
                  mc.cores = detectCores(),
                  seed = "auto")


oncoSimulSample(Nindiv,
                  fp,
                  model = "Exp",
                  numPassengers = 0,
                  mu = 1e-6,
                  muEF = NULL,
                  detectionSize = round(runif(Nindiv, 1e5, 1e8)),

                  detectionDrivers = {
                                  if(inherits(fp, "fitnessEffects")) {
                                      if(length(fp$drv)) {
                                          nd <- (2: round(0.75 * length(fp$drv)))
                                      } else {
                                          nd <- 9e6
                                      }
                                  } else {
                                      nd <- (2 : round(0.75 * max(fp)))
                                  }
                                  if (length(nd) == 1)
                                      nd <- c(nd, nd)
                                  sample(nd, Nindiv,
                                         replace = TRUE)
                              },
                  detectionProb = NA,
                  sampleEvery = ifelse(model %in% c("Bozic", "Exp"), 1,
                      0.025),
                  initSize = 500,
                  s = 0.1,
                  sh = -1,
                  K = initSize/(exp(1) - 1),
                  minDetectDrvCloneSz = "auto",
                  extraTime = 0,
                  finalTime = 0.25 * 25 * 365,
                  onlyCancer = TRUE, keepPhylog = FALSE,
```

```
                    mutationPropGrowth = ifelse(model == "Bozic",
                                                FALSE, TRUE),
                    max.memory = 2000,
                    max.wall.time.total = 600,
                    max.num.tries.total = 500 * Nindiv,
                    typeSample = "whole",
                    thresholdWhole = 0.5,
                    initMutant = NULL,
                    verbosity  = 1,
                    showProgress = FALSE,
                    seed = "auto")
```

## Arguments

| | |
|---|---|
| `Nindiv` | Number of individuals or number of different trajectories to simulate. |
| `fp` | Either a poset that specifies the order restrictions (see `poset` if you want to use the specification as in v.1. Otherwise, a fitnessEffects object (see `allFitnessEffects`).<br>Other arguments below (s, sh, numPassengers) make sense only if you use a poset, as they are included in the fitnessEffects object. |
| `model` | One of "Bozic", "Exp", "McFarlandLog" (the last one can be abbreviated to "McFL"). The default is "Exp". |
| `numPassengers` | This has no effect if you use the `allFitnessEffects` specification.<br>If you use the specification of v.1., the number of passenger genes. Note that using v.1 the total number of genes (drivers plus passengers) must be smaller than 64.<br>All driver genes should be included in the poset (even if they depend on no one and no one depends on them), and will be numbered from 1 to the total number of driver genes. Thus, passenger genes will be numbered from (number of driver genes + 1):(number of drivers + number of passengers). |
| `mu` | Mutation rate. Can be a single value or a named vector. If a single value, all genes will have the same mutation rate. If a named vector, the entries in the vector specify the gene-specific mutation rate. If you pass a vector, it must be named, and it must have entries for all the genes in the fitness specification. Passing a vector is only available when using fitnessEffects objects for fitness specification.<br>See also `mutationPropGrowth`. |
| `muEF` | Mutator effects. A mutatorEffects object as obtained from `allMutatorEffects`. This specifies how mutations in certain genes change the mutation rate over all the genome. Therefore, this allows you to specify mutator phenotypes: models where mutation of one (or more) gene(s) leads to an increase in the mutation rate. This is only available for version 2 (and above) specifications.<br>All the genes specified in `muEF` MUST be included in `fp`. If you want to have genes that have no direct effect on fitness, but that affect mutation rate, you MUST specify them in `fp`, for instance as `noIntGenes` with an effect of 0.<br>If you use mutator effects you must also use fitnessEffects in `fp`. |
| `detectionSize` | What is the minimal number of cells for cancer to be detected. For `oncoSimulSample` this can be a vector.<br>If set to NA, `detectionSize` plays no role in stopping the simulations. |

`detectionDrivers`

The minimal number of drivers (not modules, drivers, whether or not they are from the same module) present in any clone for cancer to be detected. For `oncoSimulSample` this can be a vector.

For `oncoSimulSample`, if there are drivers (either because you are using a v.1 object or because you are using a fitnessEffects object with a `drvNames` component —see `allFitnessEffects`—) the default is a vector of drivers from a uniform between 2 and 0.75 the total number of drivers. If there are no drivers (because you are using a fitnessEffects object without a `drvNames`, either because you specified it explicitly or because all of the genes are in the noIntGenes component) the simulations should not stop based on the number of drivers (and, thus, the default is set to 9e6).

If set to NA, `detectionDrivers` plays no role in stopping the simulations.

`detectionProb` Vector of arguments for the mechanism where probability of detection depends on size. If `NA`, this mechanism is not used. If 'default', the vector will be populated with default values. Otherwise, a named vector with some of the following named elements (see 'Details'):

- PDBaseline: Baseline size subtracted to total population size to compute the probability of detection. If not given explicitly, the default is `1.2 * initSize`.
- p2: The probability of detection at population size `n2`. If you specificy `p2` you must also specify `n2` and you must not specify `cPDetect`. The fault is 0.1.
- n2: The population size at which probability of detection is `p2`. The default is `2 * initSize`.
- cPDetect: The change in probability of detection with size. If you specify it, you should not specify either of `p2` or `n2`. See 'Details'.
- checkSizePEvery: Time between successive checks for the probability of exiting as a function of population size. If not given explicitly, the default is 20. See 'Details'.

If you only provide some of the elements (except for the pair `p2`, `n2`, where you must provide both if you provide any), the rest will be filled with default values.

This option can not be used with v.1 objects.

`sampleEvery` How often the whole population is sampled. This is not the same as the interval between successive samples that are kept or stored (for that, see `keepEvery`).

For very fast growing clones, you might need to have a small value here to minimize possible numerical problems (such as huge increase in population size between two successive samples that can then lead to problems for random number generators). Likewise, for models with density dependence (such as McF) this value should be very small.

`initSize` Initial population size.

`s` Selection coefficient for drivers. Only relevant if using a poset as this is included in the fitnessEffects object.

`sh` Selection coefficient for drivers with restrictions not satisfied. A value of 0 means there are no penalties for a driver appearing in a clone when its restrictions are not satisfied.

To specify "sh=Inf" (in Diaz-Uriarte, 2015) use sh = -1.

Only relevant if using a poset as this is included in the fitnessEffects object.

`K` Initial population equilibrium size in the McFarland models.

keepEvery
: Time interval between successive whole population samples that are actually stored. This must be larger or equal to sampleEvery. If keepEvery is not a multiple integer of sampleEvery, the interval between successive samples that are stored will be the smallest multiple integer of sampleEvery that is larger than or equal to keepEvery.

  If you want nice plots, set sampleEvery and keepEvery to small values (say, 5 or 2). Otherwise, you can use a sampleEvery of 1 but a keepEvery of 15, so that the return objects are not huge and the code runs a lot faster.

  Setting keepEvery = NA means we only keep the very last sample. This is useful if you only care about the final state of the simulation, not its complete history.

minDetectDrvCloneSz
: A value of 0 or larger than 0 (by default equal to initSize in the McFarland model). If larger than 0, when checking if we are done with a simulation, we verify that the sum of the population sizes of all clones that have a number of mutated drivers larger or equal to detectionDrivers is larger or equal to this minDetectDrvCloneSz.

  The reason for this parameter is to ensure that, say, a clone with a certain number of drivers that would cause the simulation to end has not just appeared and is present in only one individual that might then immediately go extinct. This can be relevant in secenarios such as the McFarland model.

  See also extraTime.

extraTime
: A value larger than zero waits those many additional time periods before exiting after having reached the exit condition (population size, number of drivers).

  The reason for this setting is to prevent the McFL models from always exiting at a time when one clone is increasing its size quickly (see minDetectDrvCloneSz). By setting an extraTime larger than 0, we can sample at points when we are at the plateau.

finalTime
: What is the maximum number of time units that the simulation can run. Set to NA to disable this limit.

onlyCancer
: Return only simulations that reach cancer?

  If set to TRUE, only simulations that satisfy the detectionDrivers or the detectionSize requirements or that are "detected" because of the detectionProb mechanism will be returned: the simulation will be repeated, within the limits set by max.num.tries and max.wall.time (and, for oncoSimulSample also max.num.tries.total and max.wall.time.total), until one which meets the detectionDrivers or detectionSize or one which is detected stochastically under detectionProb is obtained.

  If onlyCancer = FALSE the simulation is returned regardless of final population size or number of drivers in any clone and this includes simulations where the population goes extinct.

keepPhylog
: If TRUE, keep track of when and from which clone each clone is created. See also plotClonePhylog.

mutationPropGrowth
: If TRUE, make mutation rate proportional to growth rate, so clones that grow faster also mutate faster. Thus, $mutation_rate = mu * birth_rate$. This is a simple way of approximating that mutation happens at cell division (it is not strictly making mutation happen at cell division, since mutation is not strictly coupled with division). Of course, this only makes sense in models where birth rate changes.

`initMutant` For v.2: a string with the mutations of the initial mutant, if any. This is the same format as for `evalGenotype`. The default (if you pass nothing) is to start the simulation from the wildtype genotype with nothing mutated. For v.1 we no longer accept initMutant: it will be ignored.

`max.num.tries` Only applies when `onlyCancer = TRUE`. What is the maximum number of times, for an individual simulation, we can repeat the simulation for it to reach cancer? There are certain parameter settings where reaching cancer is extremely unlikely and you might not want to run forever in those cases.

`max.num.tries.total`
Only applies when `onlyCancer = TRUE` and for `oncoSimulSample`. What is the maximum number of times, over all simulations for all individuals in a population sample, that we can repeat the simulations so that cancer is reached for all individuals? The idea is to set a limit on the average minimal probability of reaching cancer for a set of simulations to be accepted.

`max.wall.time` Maximum wall time for each individual simulation run. If the simulation is not done in this time, it is aborted.

`max.wall.time.total`
Maximum wall time for all the simulations (when using `oncoSimulSample`), in seconds. If the simulation is not completed in this time, it is aborted. To prevent problems from a single individual simulation going wild, this limit is also enforced per simulation (so the run can be aborted directly from C++).

`errorHitMaxTries`
If TRUE (the default) a simulation that reaches the maximum number of repetitions allowed is considered not to have succesfully finished and, thus, an error, and no output from it will be reported. This is often what you want.

See `Details`.

`errorHitWallTime`
If TRUE (the default) a simulation that reaches the maximum wall time is considered not to have succesfully finished and, thus, an error, and no output from it will be reported. This is often what you want.

See `Details`.

`max.memory` The largest size (in MB) of the matrix of Populations by Time. If it creating it would use more than this amount of memory, it is not created. This prevents you from accidentally passing parameters that will return an enormous object.

`verbosity` If 0, run silently. Iincreasing values of verbosity provide progressively more information about intermediate steps, possible numerical notes/warnings from the C++ code, etc. Values less than 0 supress some default notes: use with care.

`typeSample` "singleCell" (or "single") for single cell sampling, where the probability of sampling a cell (a clone) is directly proportional to its population size. "wholeTumor" (or "whole") for whole tumor sampling (i.e., this is similar to a biopsy being the entire tumor). See `samplePop`.

`thresholdWhole` In whole tumor sampling, whether a gene is detected as mutated depends on thresholdWhole: a gene is considered mutated if it is altered in at least thresholdWhole proportion of the cells in that individual. See `samplePop`.

`mc.cores` Number of cores to use when simulating more than one individual (i.e., when calling oncoSimulPop).

`showProgress` If TRUE, provide information, during exection, of the individual done, and the number of attempts and time used.

seed — The seed for the C++ PRNG. You can pass a value. If you set it to NULL, then a seed will be generated in R and passed to C++. If you set it to "auto", then if you are using v.1, the behavior is the same as if you set it to NULL (a seed will be generated in R and passed to C++) but if you are using v.2, a random seed will be produced in C++. If you need reproducibility, either pass a value or set it to NULL (setting it to NULL will make the C++ seed reproducible if you use the same seed in R via `set.seed`). However, even using the same value of `seed` is unlikely to give the exact same results between platforms and compilers. Moreover, note that the defaults for `seed` are not the same in `oncoSimulIndiv`, `oncoSimulPop` and `oncoSimulSample`.

When using oncoSimulPop, if you want reproducibility, you might want to, in addition to setting `seed = NULL`, also do `RNGkind("L'Ecuyer-CMRG")` as we use `mclapply`; look at the vignette of **parallel**.

## Details

The basic simulation algorithm implemented is the BNB one of Mather et al., 2012, where I have added modifications to fitness based on the restrictions in the order of mutations.

Full details about the algorithm are provided in Mather et al., 2012. The evolutionary models, including references, and the rest of the parameters are explained in Diaz-Uriarte, 2014, especially in the Supplementary Material. The model called "Bozic" is based on Bozic et al., 2010, and the model called "McFarland" in McFarland et al., 2013.

oncoSimulPop simply calls oncoSimulIndiv multiple times. When run on POSIX systems, it can use multiple cores (via mclapply).

The `summary` methods for these classes return some of the return values (see next) as a one-row (for class oncosimul) or multiple row (for class oncosimulpop) data frame. The `print` methods for these classes simply print the summary.

Changing options `errorHitMaxTries` and `errorHitWallTime` can be useful when conducting many simulations, as in the call to `oncoSimulPop`: setting them to TRUE means nothing is recorded for those simulations where ending conditions are not reached but setting them to FALSE would allow you to record the output; this would potentially result in a mixture where some simulations would not have reached the ending condition, but this might sometimes be what you want. Note, however, that `oncoSimulSample` always has both them to TRUE, as it could not be otherwise.

`GenotypesWDistinctOrderEff` provides the information about order effects that is missing from `Genotypes`. When there are order effects, the `Genotypes` matrix can contain genotypes that are not distinguishable. Suppose there are two genes, the first and the second. In the `Genotype` output you can get two columns where there is a 1 in both genes: those two columns correspond to the two possible orders (first gene mutated first, or first gene mutated after the second). `GenotypesWDistinctOrderEff` disambiguates this. The same is done by `GenotypesLabels`; this is easier to decode for a human (a string of gene labels) but a little bit harder to parse automatically. Note that when you use the default print method for this object, you get, among others, a two-column display with the `GenotypeLabels` information. When order matters, a genotype shown as "x > y _ z" means that a mutation in "x" happened before a mutation in "y"; there is also a mutation in "z" (which could have happened before or after either of "x" or "y"), but "z" is a gene for which order does not matter. When order does not matter, a comma "," separates the identifiers of mutated genes.

Detection of cancer can be a deterministic process, where cancer is always detected (and, thus, simulation ended) when certain conditions are met (`detectionSize`, `detectionDrivers`). Alternatively, it can be stochastic process where probability of detection depends on size. Every so often (see below) we assess population size, and detect cancer or not probabilistically (comparing

the probability of detection for that size with a random uniform number). Probability of detection changes with population size according to the function

$$1 - e^{-cPDetect(populationsize - PDBaseline)}$$

.

You can pass `cPDetect` manually (you will need to set `n2` and `p2` to NA). However, it might be more intuitive to specify the pair `n2`, `p2`, such that the probability of detection is $p2$ for population size $n2$ (and from that pair we solve for the value of `cPDetect`). How often do we check? That is controlled by `checkSizePEvery`, the (minimal) time between successive checks (from among the sampling times given by `sampleEvery`: the interval between successive assessments will be the smallest multiple integer of `sampleEvery` that is larger than `checkSizePEvery` —see vignette for details). `checkSizePEvery` has, by default, a different (and much larger) value than `sampleEvery` both to allow to examine the effects of sampling, and to avoid many costly random number generations.

Please note that `detectionProb` is NOT available with version 1 objects.

## Value

For `oncoSimulIndiv` a list, of class "oncosimul", with the following components:

`pops.by.time` A matrix of the population sizes of the clones, with clones in columns and time in row. Not all clones are shown here, only those that were present in at least on of the keepEvery samples.

`NumClones` Total number of clones in the above matrix. This is not the total number of distinct clones that have appeared over all simulations (which is likely to be larger or much larger).

`TotalPopSize` Total population size at the end.

`Genotypes` A matrix of genotypes. For each of the clones in the pops.by.time matrix, its genotype, with a 0 if the gene is not mutated and a 1 if it is mutated.

`MaxNumDrivers` The largest number of mutated driver genes ever seen in the simulation in any clone.

`MaxDriversLast` The largest number of mutated drivers in any clone at the end of the simulation.

`NumDriversLargestPop`
The number of mutated driver genes in the clone with largest population size.

`LargestClone` Population size of the clone with largest number of population size.

`PropLargestPopLast`
Ratio of LargestClone/TotalPopSize

`FinalTime` The time (in time units) at the end of the simulation.

`NumIter` The number of iterations of the BNB algorithm.

`HittedWallTime` TRUE if we reached the limit of max.wall.time. FALSE otherwise.

`TotalPresentDrivers`
The total number of mutated driver genes, whether or not in the same clone. The number of elements in `OccurringDrivers`, below.

`CountByDriver` A vector of length number of drivers, with the count of the number of clones that have that driver mutated.

`OccurringDrivers`
The actual number of drivers mutated.

`PerSampleStats` A 5 column matrix with a row for each sampling period. The columns are: total population size, population size of the largest clone, the ratio of the two, the largest number of drivers in any clone, and the number of drivers in the clone with the largest population size.

`other` A list that contains statistics for an estimate of the simulation error when using the McFarland model as well as other statistics. For the McFarland model, the relevant value is errorMF, which is -99 unless in the McFarland model. For the McFarland model it is the largest difference of successive death rates. The entries names minDMratio and minBMratio are the smallest ratio, over all simulations, of death rate to mutation rate or birth rate to mutation rate. The BNB algorithm thrives when those are large.

For `oncoSimulPop` a list of length `Nindiv`, and of class `"oncosimulpop"`, where each element of the list is itself a list, of class `oncosimul`, with components as described above.

In v.2, the output is of both class "oncosimul" and "oncosimul2". The oncoSimulIndiv return object differs in

`GenotypesWDistinctOrderEff`
A list of vectors, where each vector corresponds to a genotype in the `Genotypes`, showing (where it matters) the order of mutations. Each vector shows the genotypes, with the numeric codes, showing explicitly the order when it matters. So if you have genes 1, 2, 7 for which order relationships are given, and genes 3, 4, 5, 6 for which other interactions exist, any mutations in 1, 2, 7 are shown first, and in the order they occurred, before showing the rest of the mutations. See details.

`GenotypesLabels`
The genotypes, as character vectors with the original labels provided (i.e., not the integer codes). As before, mutated genes, for those where order matters, come first, and are separated by the rest by a "_". See details.

`OccurringDrivers`
This is the same as in v.1, but we use the labels, not the numeric id codes. Of course, if you entered integers as labels for the genes, you will see numbers (however, as a character string).

## Note

Please, note that the meaning of the fitness effects in the McFarland model is not the same as in the original paper; the fitness coefficients are transformed to allow for a simpler fitness function as a product of terms. This differs with respect to v.1. See the vignette for details.

## Author(s)

Ramon Diaz-Uriarte

## References

Bozic, I., et al., (2010). Accumulation of driver and passenger mutations during tumor progression. *Proceedings of the National Academy of Sciences of the United States of America\/*, **107**, 18545–18550.

Diaz-Uriarte, R. (2015). Identifying restrictions in the order of accumulation of mutations during tumor progression: effects of passengers, evolutionary models, and sampling http://www.biomedcentral.com/1471-2105/16/41/abstract

Gerstung et al., 2011. The Temporal Order of Genetic and Pathway Alterations in Tumorigenesis. *PLoS ONE*, 6.

McFarland, C.~D. et al. (2013). Impact of deleterious passenger mutations on cancer progression. *Proceedings of the National Academy of Sciences of the United States of America\/*, **110**(8), 2910–5.

Mather, W.~H., Hasty, J., and Tsimring, L.~S. (2012). Fast stochastic algorithm for simulating evolutionary population dynamics. *Bioinformatics (Oxford, England)\/*, **28**(9), 1230–1238.

## See Also

plot.oncosimul, examplePosets, samplePop, allFitnessEffects

## Examples

```
#################################
#####
##### Examples using v.1
#####
#################################


## use poset p701
data(examplePosets)
p701 <- examplePosets[["p701"]]

## Exp Model

b1 <- oncoSimulIndiv(p701)
summary(b1)

plot(b1, addtot = TRUE)

## McFarland; use a small sampleEvery, but also a reasonable
##   keepEvery.
## We also modify mutation rate to values similar to those in the
##   original paper.
## Note that detectionSize will play no role
## finalTime is large, since this is a slower process
## initSize is set to 4000 so the default K is larger and we are likely
## to reach cancer. Alternatively, set K = 2000.

m1 <- oncoSimulIndiv(p701,
                     model = "McFL",
                     mu = 5e-7,
                     initSize = 4000,
                     sampleEvery = 0.025,
                     finalTime = 15000,
                     keepEvery = 10,
                     onlyCancer = FALSE)
plot(m1, addtot = TRUE, log = "")




## Simulating 4 individual trajectories
## (I set mc.cores = 2 to comply with --as-cran checks, but you
```

```
##  should either use a reasonable number for your hardware or
##  leave it at its default value).


p1 <- oncoSimulPop(4, p701,
                   keepEvery = 10,
                   mc.cores = 2)
summary(p1)
samplePop(p1)

p2 <- oncoSimulSample(4, p701)


########################################
######
###### Examples using v.2:
######
########################################



#### A model similar to the one in McFarland. We use 2070 genes.

set.seed(456)
nd <- 70
np <- 2000
s <- 0.1
sp <- 1e-3
spp <- -sp/(1 + sp)
mcf1 <- allFitnessEffects(noIntGenes = c(rep(s, nd), rep(spp, np)),
                          drv = seq.int(nd))
mcf1s <-  oncoSimulIndiv(mcf1,
                         model = "McFL",
                         mu = 1e-7,
                         detectionSize = 1e8,
                         detectionDrivers = 100,
                         sampleEvery = 0.02,
                         keepEvery = 2,
                         initSize = 2000,
                         finalTime = 1000,
                         onlyCancer = FALSE)
plot(mcf1s, addtot = TRUE, lwdClone = 0.6, log = "")
summary(mcf1s)
plot(mcf1s)


#### Order effects with modules, and 5 genes without interactions
#### with fitness effects from an exponential distribution

oi <- allFitnessEffects(orderEffects =
               c("F > D" = -0.3, "D > F" = 0.4),
               noIntGenes = rexp(5, 10),
                           geneToModule =
                               c("Root" = "Root",
                                 "F" = "f1, f2, f3",
                                 "D" = "d1, d2") )
oiI1 <- oncoSimulIndiv(oi, model = "Exp")
```

```
oiI1$GenotypesLabels
oiI1   ## note the order and separation by "_"


oiP1 <- oncoSimulPop(2, oi,
                     keepEvery = 10,
                     mc.cores = 2)
summary(oiP1)


## Even if order exists, this cannot reflect it;
## G1 to G10 are d1, d2, f1..,f3, and the 5 genes without
## interaction
samplePop(oiP1)


oiS1 <- oncoSimulSample(2, oi)

## The output contains only the summary of the runs AND
## the sample:
oiS1

## And their sizes do differ
object.size(oiS1)
object.size(oiP1)



######## Using a poset for pancreatic cancer from Gerstung et al.
###      (s and sh are made up for the example; only the structure
###       and names come from Gerstung et al.)


pancr <- allFitnessEffects(data.frame(parent = c("Root", rep("KRAS", 4), "SMAD4", "CDNK2A",
                                          "TP53", "TP53", "MLL3"),
                                      child = c("KRAS","SMAD4", "CDNK2A",
                                          "TP53", "MLL3",
                                          rep("PXDN", 3), rep("TGFBR2", 2)),
                                      s = 0.05,
                                      sh = -0.3,
                                      typeDep = "MN"))
plot(pancr)

### Use an exponential growth model

pancr1 <- oncoSimulIndiv(pancr, model = "Exp")
pancr1
summary(pancr1)
plot(pancr1)
pancr1$GenotypesLabels


## Pop and Sample
pancrPop <- oncoSimulPop(4, pancr,
                         keepEvery = 10,
                        mc.cores = 2)
summary(pancrPop)
```

```
pancrSPop <- samplePop(pancrPop)
pancrSPop

pancrSamp <- oncoSimulSample(2, pancr)
pancrSamp



## Using gene-specific mutation rates
muv <- c("U" = 1e-3, "z" = 1e-7, "e" = 1e-6, "m" = 1e-5, "D" = 1e-4)
ni <- rep(0.01, 5)
names(ni) <- names(muv)
femuv <- allFitnessEffects(noIntGenes = ni)
oncoSimulIndiv(femuv, mu = muv)


## Reinitialize the RNG
set.seed(NULL)
```

---

`OncoSimulWide2Long` *Convert the* `pops.by.time` *component of an* `oncosimul` *object into "long" format.*

---

## Description

Conver the `pops.by.time` component from its "wide" format (with one column for time, and as many columns as clones/genotypes) into "long" format, so that it can be used with other functions, for instance for plots.

## Usage

```
OncoSimulWide2Long(x)
```

## Arguments

| | |
|---|---|
| `x` | An object of class `oncosimul` or `oncosimul2`. |

## Value

A data frame with four columns: Time; Y, the number of cells (the population size); Drivers, a factor with the number of drivers of the given genotype; Genotype, the genotyp.

## Author(s)

Ramon Diaz-Uriarte

## See Also

`oncoSimulIndiv`

## Examples

```
data(examplePosets)
## An object of class oncosimul
p705 <- examplePosets[["p705"]]
p1 <- oncoSimulIndiv(p705)
class(p1)
lp1 <- OncoSimulWide2Long(p1)
head(lp1)
summary(lp1)

## An object of class oncosimul2
data(examplesFitnessEffects)

sm <-  oncoSimulIndiv(examplesFitnessEffects$cbn1,
                      model = "McFL",
                      mu = 5e-7,
                      detectionSize = 1e8,
                      detectionDrivers = 2,
                      sampleEvery = 0.025,
                      keepEvery = 5,
                      initSize = 2000,
                      onlyCancer = FALSE)
class(sm)
lsm <- OncoSimulWide2Long(sm)
head(lsm)
summary(lsm)
```

---

`plot.fitnessEffects` *Plot fitnessEffects objects.*

---

## Description

Plot the restriction table/graph of restrictions, the epistasis, and the order effects in a fitnessEffects object. This is not a plot of the fitness landscape; for that, see `plotFitnessLandscape`.

## Usage

```
## S3 method for class 'fitnessEffects'
plot(x, type = "graphNEL", layout = NULL,
expandModules = FALSE, autofit = FALSE,
scale_char = ifelse(type == "graphNEL", 1/10, 5),
return_g = FALSE, lwdf = 1, ...)
```

## Arguments

| | |
|---|---|
| `x` | A `fitnessEffects` object, as produced by `allFitnessEffects`. |
| `type` | Whether you want a "graphNEL" or an "igraph" graph. |
| `layout` | For "igraph", the layout. For example, if you know you really have only a tree you might want to use `layout.reingold.tilford`. Note that there is very limited support for passing options, etc. In most cases, it is either the default or the `layout.reingold.tilford`. |

| | |
|---|---|
| expandModules | If there are modules with multiple genes, if you set this to TRUE modules will be replaced by their genes. |
| autofit | If TRUE, we try to fit the edges to the labels. This is a very experimental feature, likely to be not very robust. |
| scale_char | If using autofit = TRUE, the scaling factor for the size of the rectangles as a function of the number of characters. You have to play with this because the best value can depend on a number of things. |
| return_g | It TRUE, the graph object (graphNEL or igrap) is returned. |
| lwdf | The multiplier factor for lwd when using "graphNEL". |
| ... | Other arguments passed to plot. Not used for now. |

## Value

A plot.

Order and epistatic relationships have orange edges. OR (semimonotone) relationships blue, and XOR red. All others have black edges (so AND and unique edges from root). Epistatic relationships, being symmetrical, have no arrows between nodes and have a dotted line type. Order relationships have an arrow from the earlier to the later event and have a different dotted line (lty 3).

If return_g is TRUE, you are returned also the graph object (igraph or graphNEL) so that you can manipulate it further.

## Note

The purpose of the plot is to get a quick idea of the relationships. Note that three-way (or higher order) epistatic relationships cannot be shown as such (we would show all possible pairs, but that is not quite the same thing). Likewise, there is no reasonable way to convey the pressence of a "-" in the epistatic relationship.

Genes without interactions are not shown.

## Author(s)

Ramon Diaz-Uriarte

## See Also

allFitnessEffects, plotFitnessLandscape

## Examples

```
cs <-  data.frame(parent = c(rep("Root", 4), "a", "b", "d", "e", "c"),
                  child = c("a", "b", "d", "e", "c", "c", rep("g", 3)),
                  s = 0.1,
                  sh = -0.9,
                  typeDep = "MN")

cbn1 <- allFitnessEffects(cs)
plot(cbn1, "igraph")

library(igraph) ## to make layouts available
plot(cbn1, "igraph", layout = layout.reingold.tilford)
```

```
### A DAG with the three types of relationships
p3 <- data.frame(parent = c(rep("Root", 4), "a", "b", "d", "e", "c", "f"),
                 child = c("a", "b", "d", "e", "c", "c", "f", "f", "g", "g"),
                 s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, 0.2, 0.2, 0.3, 0.3),
                 sh = c(rep(0, 4), c(-.9, -.9), c(-.95, -.95), c(-.99, -.99)),
                 typeDep = c(rep("--", 4),
                     "XMPN", "XMPN", "MN", "MN", "SM", "SM"))
fp3 <- allFitnessEffects(p3)

plot(fp3)

plot(fp3, "igraph", layout = layout.reingold.tilford)


## A more complex example, that includes a restriction table
## order effects, epistasis, genes without interactions, and moduels
p4 <- data.frame(parent = c(rep("Root", 4), "A", "B", "D", "E", "C", "F"),
                 child = c("A", "B", "D", "E", "C", "C", "F", "F", "G", "G"),
                 s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, 0.2, 0.2, 0.3, 0.3),
                 sh = c(rep(0, 4), c(-.9, -.9), c(-.95, -.95), c(-.99, -.99)),
                 typeDep = c(rep("--", 4),
                     "XMPN", "XMPN", "MN", "MN", "SM", "SM"))

oe <- c("C > F" = -0.1, "H > I" = 0.12)
sm <- c("I:J"  = -1)
sv <- c("-K:M" = -.5, "K:-M" = -.5)
epist <- c(sm, sv)

modules <- c("Root" = "Root", "A" = "a1",
             "B" = "b1, b2", "C" = "c1",
             "D" = "d1, d2", "E" = "e1",
             "F" = "f1, f2", "G" = "g1",
             "H" = "h1, h2", "I" = "i1",
             "J" = "j1, j2", "K" = "k1, k2", "M" = "m1")

noint <- rexp(5, 10)
names(noint) <- paste0("n", 1:5)

fea <- allFitnessEffects(rT = p4, epistasis = epist, orderEffects = oe,
                         noIntGenes = noint, geneToModule = modules)


plot(fea)
plot(fea, expandModules = TRUE)
plot(fea, type = "igraph")
```

---

`plot.oncosimul` *Plot simulated tumor progression data.*

---

## Description

Plots data generated from the simulations, either for a single individual or for a population of individuals, with time units in the x axis and nubmer of cells in the y axis.

In "drivers" plots, by default, all clones with the same number of drivers are plotted using the same colour (but different line types), and clones with different number of drivers are plotted in different colours. Plots can alternatively display genotypes instead of drivers.

Plots available are line plots, stacked area, and stream plots.

## Usage

```
## S3 method for class 'oncosimul'
plot(x,
                            show = "drivers",
                            type = ifelse(show == "genotypes",
                                          "stacked", "line"),
                            col = "auto",
                            log = ifelse(type == "line", "y", ""),
                            ltyClone = 2:6,
                            lwdClone = 0.9,
                            ltyDrivers = 1,
                            lwdDrivers = 3,
                            xlab = "Time units",
                            ylab = "Number of cells",
                            plotClones = TRUE,
                            plotDrivers = TRUE,
                            addtot = FALSE,
                            addtotlwd = 0.5,
                            ylim = NULL,
                            xlim = NULL,
                            thinData = FALSE,
                            thinData.keep = 0.1,
                            thinData.min = 2,
                            plotDiversity = FALSE,
                            order.method = "as.is",
                            stream.center = TRUE,
                            stream.frac.rand = 0.01,
                            stream.spar = 0.2,
                            border = NULL,
                            lwdStackedStream = 1,
                            srange = c(0.4, 1),
                            vrange = c(0.8, 1),
                            breakSortColors = "oe",
                            legend.ncols = "auto", ...)

## S3 method for class 'oncosimulpop'
plot(x,
                               ask = TRUE,
                               show = "drivers",
                               type = ifelse(show == "genotypes",
                                             "stacked", "line"),
                               col = "auto",
                               log = ifelse(type == "line", "y", ""),
                               ltyClone = 2:6,
                               lwdClone = 0.9,
```

```
                    ltyDrivers = 1,
                    lwdDrivers = 3,
                    xlab = "Time units",
                    ylab = "Number of cells",
                    plotClones = TRUE,
                    plotDrivers = TRUE,
                    addtot = FALSE,
                    addtotlwd = 0.5,
                    ylim = NULL,
                    xlim = NULL,
                    thinData = FALSE,
                    thinData.keep = 0.1,
                    thinData.min = 2,
                    plotDiversity = FALSE,
                    order.method = "as.is",
                    stream.center = TRUE,
                    stream.frac.rand = 0.01,
                    stream.spar = 0.2,
                    border = NULL,
                    lwdStackedStream = 1,
                    srange = c(0.4, 1),
                    vrange = c(0.8, 1),
                    breakSortColors = "oe",
                    legend.ncols = "auto",
                    ...)
```

## Arguments

| | |
|---|---|
| x | An object of class `oncosimul` (for `plot.oncosimul`) or `oncosimulpop` (for `plot.oncosimulpop`). |
| ask | Same meaning as in `par`. |
| show | One of "drivers" or "genotypes". If "drivers" the legend will reflect the number of drivers. If "genotypes" you will be shown genotypes. You probably want to limit "genotypes" to those cases where only a relatively small number of genotypes exist (or the plot will be an unmanageable mess). The default is "drivers". |
| type | One of "line", "stacked", "stream".<br>If "line", you are shown lines for each genotype or clone. This means that to get an idea of the total population size you need to use `plotDrivers = TRUE` with `addtot = TRUE`, or do the visual calculation in your head.<br>If "stacked" a stacked area plot. If "stream" a stream plot. Since these stack areas, you immediately get the total population. But that also means you cannot use `log`.<br>The default is to use "line" for `show = "drivers"` and "stacked" for `show = "genotypes"`. |
| col | Colour of the lines/areas. For `show = "drivers"` each type of clone (where type is defined by number of drivers) has a different color. For `show = "genotypes"` color refers to genotypes. The vector is recycled as needed.<br>The default is "auto". If you have `show == "genotypes"` we start from the "Dark2" palette from `brewer.pal` in the `RColorBrewer` package and extend the palette via `colorRampPalette`. For `show == "drivers"` and `type == "line"` |

we use a vector of eight colors (that are, then recycled as needed). If you use "stacked" or "stream", however, instead of "line", then we generate colors via a HSV specification that tries to: a) make it easy to differentiate between different drivers (by not having like colors for adjacent numbers of drivers); b) make it easy to have a "representative" driver color while using sligtly different colors for different clones of a driver. See the code by doing `OncoSimulR:::myhsvcols`.

You can specify your own vector of colors, but it will be ignored with `show == "drivers"`.

`log` See `log` in `plot.default`. The default is to have `"y"` for `type == "line"`, and that will make the y axis logarithmic. Stacked and stream area plots do not allow for logarithmic y axis (since those depend on the additivity of areas but `log(a + b) != log(a) + log(b)`).

`ltyClone` Line type for each clone. Recycled as needed. You probably do not want to use lty=1 for any clone, to differentiate from the clone type, unless you change the setting for `ltyDrivers`.

`lwdClone` Line width for clones.

`ltyDrivers` Line type for the driver type.

`lwdDrivers` Line width for the driver type.

`xlab` Same as `xlab` in `plot.default`.

`ylab` Same as `ylab` in `plot.default`.

`plotClones` Should clones be plotted?

`plotDrivers` Should clone types (which are defined by number of drivers), be plotted? (Only applies when using `show = "drivers"`).

`addtot` If TRUE, add a line with the total populatino size.

`addtotlwd` Line width for total population size.

`ylim` If non NULL, limits of the y axis. Same as in `plot.default`. If NULL, the limits are calculated automatically.

`xlim` If non NULL, limits of the x axis. Same as in `plot.default`. If NULL, the limits are calculated automatically. Using a non-NULL range smaller than the range of observed values of time can also lead to speed ups of large figures (since we trim the data).

`thinData` If TRUE, the data plotted is a subset of the original data. The original data are "thinned" in such a way that the origin of each clone is not among the non-shown data (i.e., so that we can see when each clone/driver originates).

Thinning is done to reduce the plot size and to speed up plotting.

Note that thinning is carried out before dealing with the plot axis, so the actual number of points to be plotted could be a lot less (if you reduce the x-axis considerably) than those returned from the thinning. (In extreme cases this could lead to crashes when trying to use stream plots if, say, you end up plotting only three values).

`thinData.keep` The fraction of the data to keep (actually, a lower bound on the fraction of data to keep).

`thinData.min` Any time point for which a clone has a population size > thinData.min will be kept (i.e., will not be removed from) in the data.

`plotDiversity` If TRUE, we also show, on top of the main figure, Shannon's diversity index (and we consider as distinct those genotypes with different order of mutations when order matters).

If you set this to true, using `par(mfrow = c(2, 2))` and similar will not work (since we use `par(fig = )`) to display the diversity as the top plot).

`order.method` For stacked and stream plots. `c("as.is", "max", "first")`. `"as.is"`: plot in order of y column; `"max"`: plot in order of when each y series reaches maximum value. `"first"`: plot in order of when each y series first value > 0.

`stream.center` For stream plots. If TRUE, the stacked polygons will be centered so that the middle, i.e. baseline ("g0"), of the stream is approximately equal to zero. Centering is done before the addition of random wiggle to the baseline.

`stream.frac.rand`
For stream plots. Fraction of the overall data "stream" range used to define the range of random wiggle (uniform distribution) to be added to the baseline 'g0'.

`stream.spar` Setting for smooth.spline function to make a smoothed version of baseline "g0".

`border` For stacked and stream plots. Border colors for polygons corresponding to y columns (will recycle) (see `polygon` for details).

`lwdStackedStream`
border line width for polygons corresponding to y columns (will recycle).

`srange` Range of values of s in the HSV specification of colors (see `col` for details. Only applies when using "stacked" or "stream" plots and `col == "auto"`.)

`vrange` Range of values of v in the HSV specification of colors (see `col` for details.Only applies when using "stacked" or "stream" plots and `col == "auto"`. )

`breakSortColors`
How to try to minimize that similar colors be used for contiguous or nearby driver categories. The default is "oe" which resorts them in alternating way. The other two options are "distave", where we alternate after folding from the mean and "random" where the colors are randomly sorted. Only applies when using "stacked" or "stream" plots and `col == "auto"`.

`legend.ncols` The number of columns of the legend. If `"auto"` (the default), will have one column for six or less entries, and two for more than six.

`...` Other arguments passed to `plots`. For instance, `main`.

## Author(s)

Ramon Diaz-Uriarte. Marc Taylor for stacked and stream plots.

## See Also

`oncoSimulIndiv`

## Examples

```
data(examplePosets)
p701 <- examplePosets[["p701"]]

## Simulate and plot a single individual, including showing
## Shannon's diversity index
b1 <- oncoSimulIndiv(p701)
plot(b1, addtot = TRUE, plotDiversity = TRUE)

## A stacked area plot
plot(b1, type = "stacked", plotDiversity = TRUE)
```

```
## And what if I show a stream plot?
plot(b1, type = "stream", plotDiversity = TRUE)

## Simulate and plot 2 individuals
## (I set mc.cores = 2 to comply with --as-cran checks, but you
##  should either use a reasonable number for your hardware or
##  leave it at its default value).

p1 <- oncoSimulPop(2, p701, mc.cores = 2)

par(mfrow = c(1, 2))
plot(p1, ask = FALSE)

## Stacked; we cannot log here, and harder to see patterns
plot(p1, ask = FALSE, type = "stacked")


## Show individual genotypes and drivers for an
## epistasis case with at most eight genotypes

sa <- 0.1
sb <- -0.2
sab <- 0.25
sac <- -0.1
sbc <- 0.25
sv2 <- allFitnessEffects(epistasis = c("-A : B" = sb,
                                       "A : -B" = sa,
                                       "A : C" = sac,
                                       "A:B" = sab,
                                       "-A:B:C" = sbc),
                         geneToModule = c(
                             "Root" = "Root",
                             "A" = "a1, a2",
                             "B" = "b",
                             "C" = "c"))
evalAllGenotypes(sv2, order = FALSE, addwt = TRUE)
e1 <- oncoSimulIndiv(sv2, model = "McFL",
                     mu = 5e-6,
                     sampleEvery = 0.02,
                     keepEvery = 1,
                     initSize = 2000,
                     finalTime = 3000,
                     onlyCancer = FALSE)

## Drivers and clones
plot(e1, show = "drivers")

## Make genotypes explicit
plot(e1, show = "genotypes")

## Oh, but I want other colors
plot(e1, show = "genotypes", col = rainbow(8))

## and actually I want a line plot
plot(e1, show = "genotypes", type = "line")
```

`plotClonePhylog` *Plot a parent-child relationship of the clones.*

## Description

Plot a parent-child relationship of the clones, controlling which clones are displayed, and whether to shown number of times of appearance, and time of first appearance of a clone.

## Usage

```
plotClonePhylog(x, N = 1, t ="last", timeEvents = FALSE,
                keepEvents = FALSE, fixOverlap = TRUE,
                returnGraph = FALSE, ...)
```

## Arguments

| | |
|---|---|
| `x` | The output from a simulation, as obtained from `oncoSimulIndiv`, `oncoSimulPop`, or `oncoSimulSample` (see `oncoSimulIndiv`). This must be from v.2 and forward (no phylogenetic information is stored for earlier objects). |
| `N` | Show in the plot all clones that have a population size of at least N at time `time` and the parents of those clones (parents are shown regardless of population size —i.e., you can see extinct parents). If you want to show everything that ever appeared, set N = 0. |
| `t` | The time at which `N` should be satisfied. This can either be the string "last", meaning the last time of the simulation, or a range of two values. In the second case, all clones with population size of at least `N` in at least one time point between time[1] and time[2] will be shown (togheter with their parents). |
| `timeEvents` | If TRUE, the vertical position of the nodes in the plot will be proportional to their time of first appearance. |
| `keepEvents` | If TRUE, the graph will show all the birth events. Thus, the number of arrows shows the number of times a clone give rise to another. For large graphs with many events, this slows the graph considerably. |
| `fixOverlap` | When using `timeEvents = TRUE` nodes can overlap (as we modify their vertical location after igraph has done the initial layout). This attempts to fix that problem by randomly relocating, along the X axis, the nodes that have the same X value. |
| `returnGraph` | If TRUE, the igraph object is returned. You can use this to plot the object however you want or obtain the adjacency matrix. |
| `...` | Additional arguments. Currently not used.. |

## Value

A plot is produced. If `returnGraph` the igraph object is returned.

## Note

These are not, technically, proper phylogenetic trees and we use "phylogeny" here in an abuse of terminology. The plots we use, where we show parent child relationships are arguably more helpful in this context. But you could draw proper phylogenies with the information provided.

If you want to obtain the adjacency matrix, this is trivial: just set `returnGraph = TRUE` and use `get.adjacency`. See an example below.

## Author(s)

Ramon Diaz-Uriarte

## See Also

oncoSimulIndiv

## Examples

```
data(examplesFitnessEffects)
tmp <-  oncoSimulIndiv(examplesFitnessEffects[["o3"]],
                       model = "McFL",
                       mu = 5e-5,
                       detectionSize = 1e8,
                       detectionDrivers = 3,
                       sampleEvery = 0.025,
                       max.num.tries = 10,
                       keepEvery = 5,
                       initSize = 2000,
                       finalTime = 3000,
                       onlyCancer = FALSE,
                       keepPhylog = TRUE)

## Show only those with N > 10 at end
plotClonePhylog(tmp, N = 10)

## Show only those with N > 1 between times 5 and 1000
plotClonePhylog(tmp, N = 1, t = c(5, 1000))

## Show everything, even if teminal nodes are extinct
plotClonePhylog(tmp, N = 0)


## Show time when first appeared
plotClonePhylog(tmp, N = 10, timeEvents = TRUE)

## Not run:
## Show each event
## This can take a few seconds
plotClonePhylog(tmp, N = 10, keepEvents = TRUE)

## End(Not run)

## Adjacency matrix
require(igraph)
get.adjacency(plotClonePhylog(tmp, N = 10, returnGraph = TRUE))
```

---

plotFitnessLandscape *Plot a fitness landscape.*

---

## Description

Show a plot of a fitness landscape. The plot is modeled after (actually, mostly a blatant copy of) that of MAGELLAN, http://wwwabi.snv.jussieu.fr/public/Magellan/.

Note: this is not a plot of the fitnessEffects object; for that, see plot.fitnessEffects.

## Usage

```
plotFitnessLandscape(x, show_labels = TRUE,
                     col = c("green4", "red", "yellow"),
                     lty = c(1, 2, 3),
                     use_ggrepel = FALSE,
                     log = FALSE, max_num_genotypes = 2000,
                     only_accessible = FALSE,
                     accessible_th = 0,
                     ...)

## S3 method for class 'genotype_fitness_matrix'
plot(x, show_labels = TRUE,
                                  col = c("green4", "red", "yellow"),
                                  lty = c(1, 2, 3),
                                  use_ggrepel = FALSE,
                                  log = FALSE, max_num_genotypes = 2000,
                                  only_accessible = FALSE,
                                  accessible_th = 0,
                                  ...)

## S3 method for class 'evalAllGenotypes'
plot(x, show_labels = TRUE,
                                  col = c("green4", "red", "yellow"),
                                  lty = c(1, 2, 3),
                                  use_ggrepel = FALSE,
                                  log = FALSE, max_num_genotypes = 2000,
                                  only_accessible = FALSE,
                                  accessible_th = 0,
                                  ...)

## S3 method for class 'evalAllGenotypesMut'
plot(x, show_labels = TRUE,
                                  col = c("green4", "red", "yellow"),
                                  lty = c(1, 2, 3),
                                  use_ggrepel = FALSE,
                                  log = FALSE, max_num_genotypes = 2000,
                                  only_accessible = FALSE,
                                  accessible_th = 0,
                                  ...)
```

## Arguments

x — One of the following:

- A matrix (or data frame) with g + 1 columns. Each of the first g columns contains a 1 or a 0 indicating that the gene of that column is mutated or not.

Column g+ 1 contains the fitness values. This is, for instance, the output you will get from rfitness.

- A two column data frame. The second column is fitness, and the first column are genotypes, given as a character vector. For instance, a row "A, B" would mean the genotype with both A and B mutated.
- The output from a call to evalAllGenotypes. Make sure you use order = FALSE in that call.
- The output from a call to evalAllGenotypesMut. Make sure you use order = FALSE.
- The output from a call to allFitnessEffects.

The first two are the same as the format for the genotFitness component in allFitnessEffects.

show_labels If TRUE, show the genotype labels.

col A three-element vector that gives the colors to use for increase, decreases and no changes in fitness, respectively. The first two colours are also used for peaks and sinks.

lty A three-element vector that gives the line types to use for increase, decreases and no changes in fitness, respectively.

use_ggrepel If TRUE, use the ggrepel package to avoid overlap of labels.

log Log-scale the y axis.

max_num_genotypes
Maximum allowed number of genotypes. For some types of input, we make a call to evalAllGenotypes, and use this as the maximum.

only_accessible
If TRUE, show only accessible paths. A path is considered accesible if, at each mutational step (i.e., with the addition of each mutation) fitness increases by at least accessible_th. If you set only_accessible = TRUE, the number of genotypes displayed can be much smaller than the number of existing genotypes if many of those genotypes are not accessible via any path.

accessible_th The threshold for the minimal change in fitness at each mutation step (i.e., between successive genotypes) to be used if only_accessible = TRUE.

... Other arguments passed to plot. Not used for now.

## Value

A fitness landscape plot: a plot showing paths between genotypes and peaks and sinks (local maxima and minima).

## Note

I have copied most of the ideas (and colors, and labels) of this plot from MAGELLAN (http://wwwabi.snv.jussieu.fr/public/Magellan/) but MAGELLAN has other functionality that is not provided here such as epistasis stats for the landscape, and several visual manipulation options.

One feature of this function that is not available in MAGELLAN is showing genotype labels (i.e., annotated by gene names), which can be helpful if the different genotypes mean something to you.

In addition to the above differences, another difference between this plot and those of MAGELLAN is **how sinks/peaks of more than one genotype are dealt with**. This plot will show as sinks or peaks sets of one or more genotypes that are of identical fitness (and separated by a Haming distance

of one). So a sink or a peak might actually be made of more than one genotype. In MAGELLAN, as far as I can tell, peaks and sinks are always made of a single isolated genotype.

Does this matter? In most realistic cases where not two genotypes can have exactly the same fittnes it does not. In some cases, though, it might matter. Are multi-genotype sinks/peaks really sinks/peaks? Arguably yes: suppose genotypes "AB" and "ABC" both have fitness 0, which is minimal among the fitness in the set of genotypes, and genotypes "A" and "ABCD" have fitness 0.1. To go from "A" to "ABCD", if you want to travel through "AB", you have to go through the valley of "AB" and "ABC"; once in "ABC" you can climb up to "ABCD"; and once in "AB" you can move to "ABC" since it has identical fitness to "AB". Mutatis mutandis for multi-genotype peaks. Ignoring the possibility of peaks/sinks made of more than one genotype actually makes code much simpler.

Sometimes not showing the any links that involve a decrease in fitness can help see non-accessible pathways (in strong selection, no multiple mutations, etc); do this by passing, for instance, an NA for the second element of `col`.

Finally, use common sense: for instance, if you pass a `allFitnessEffects` that specifies for, say, the fitness of a total of 5000 genotypes you'll have to wait a while for the plot to finish.

## Author(s)

Ramon Diaz-Uriarte

## References

MAGELLAN web site: http://wwwabi.snv.jussieu.fr/public/Magellan/

Brouillet, S. et al. (2015). MAGELLAN: a tool to explore small fitness landscapes. *bioRxiv*, **31583**. http://doi.org/10.1101/031583

## See Also

`allFitnessEffects`, `evalAllGenotypes`, `allFitnessEffects`, `rfitness`, `plot.fitnessEffects`

## Examples

```
## Generate random fitness for four genes-genotypes
## and plot landscape.

r1 <- rfitness(4)
plot(r1)


## Specify fitness in a matrix, and plot it

m5 <- cbind(A = c(0, 1, 0, 1), B = c(0, 0, 1, 1), F = c(1, 2, 3, 5.5))
plotFitnessLandscape(m5)


## Specify fitness with allFitnessEffects, and plot it

fe <- allFitnessEffects(epistasis = c("a : b" = 0.3,
                                      "b : c" = 0.5),
                        noIntGenes = c("e" = 0.1))

plot(evalAllGenotypes(fe, order = FALSE))
```

```
## same as
plotFitnessLandscape(evalAllGenotypes(fe, order = FALSE))
```

---

`plotPoset` *Plot a poset.*

---

## Description

Plot a poset. Optionally add a root and change names of nodes.

## Usage

```
plotPoset(x, names = NULL, addroot = FALSE, box = FALSE, ...)
```

## Arguments

| | |
|---|---|
| `x` | A poset. A matrix with two columns where, in each row, the first column is the ancestor and the second the descendant. Note that there might be multiple rows with the same ancestor, and multiple rows with the same descendant. See `poset`. |
| `names` | If not NULL, a vector of names for the nodes, with the same length as the total number of nodes in a poset (which need not be the same as the number of rows; see `poset`). If `addroot = TRUE`, then 1 + the number of nodes in the poset. |
| `addroot` | Add a "Root" node to the graph? |
| `box` | Should the graph be placed inside a box? |
| `...` | Additional arguments to `plot` (actually, `plot.graphNEL` in the `Rgraphviz` package). |

## Details

The poset is converted to a `graphNEL` object.

## Value

A plot is produced.

## Author(s)

Ramon Diaz-Uriarte

## See Also

`examplePosets`, `poset`

## Examples

```
data(examplePosets)
plotPoset(examplePosets[["p1101"]])

## If you will be using that poset a lot, maybe simpler if

poset701 <- examplePosets[["p701"]]
plotPoset(poset701, addroot = TRUE)

## Compare to Pancreatic cancer figure in Gerstung et al., 2011

plotPoset(poset701,
          names = c("KRAS", "SMAD4", "CDNK2A", "TP53",
                    "MLL3","PXDN", "TGFBR2"))

## If you want to show Root explicitly do

plotPoset(poset701, addroot = TRUE,
          names = c("Root", "KRAS", "SMAD4", "CDNK2A", "TP53",
                    "MLL3","PXDN", "TGFBR2"))


## Of course, names are in the order of nodes, so KRAS is for node 1,
##   etc, but the order of entries in the poset does not matter:

poset701b <- poset701[nrow(poset701):1, ]

plotPoset(poset701b,
          names = c("KRAS", "SMAD4", "CDNK2A", "TP53",
                    "MLL3","PXDN", "TGFBR2"))
```

---

`poset` *Poset*

---

## Description

Poset: explanation.

## Arguments

| | |
|---|---|
| `x` | The poset. See details. |

## Details

A poset is a two column matrix. In each row, the first column is the ancestor (or the restriction) and the second column the descendant (or the node that depends on the restriction). Each node is identified by a positive integer. The graph includes all nodes with integers between 1 and the largest integer in the poset.

Each node can be necessary for several nodes: in this case, the same node would appear in the first column in several rows.

A node can depend on two or more nodes (conjunctions): in this case, the same node would appear in the second column in several rows.

There can be nodes that do not depend on anything (except the Root node) and on which no other nodes depend. The simplest and safest way to deal with all possible cases, including these cases, is to have all nodes with at least one entry in the poset, and nodes that depend on no one, and on which no one depends should be placed on the second column (with a 0 on the first column).

Alternatively, any node not named explicitly in the poset, but with a number smaller than the largest number in the poset, is taken to be a node that depends on no one and on which no one depends. See examples below.

This specification of restrictions is for version 1. See `allFitnessEffects` for a much more flexible one for version 2. Both can be used with `oncoSimulIndiv`.

## Author(s)

Ramon Diaz-Uriarte

## References

Posets and similar structures appear in several places. The following two papers use them extensively.

Gerstung et al., 2009. Quantifying cancer progression with conjunctive Bayesian networks. *Bioinformatics*, 21: 2809–2815.

Gerstung et al., 2011. The Temporal Order of Genetic and Pathway Alterations in Tumorigenesis. *PLoS ONE*, 6.

## See Also

`examplePosets`, `plotPoset`, `oncoSimulIndiv`

## Examples

```
## Node 2 and 3 depend on 1, and 4 depends on no one
p1 <- cbind(c(1L, 1L, 0L), c(2L, 3L, 4L))
plotPoset(p1, addroot = TRUE)

## Node 2 and 3 depend on 1, and 4 to 7 depend on no one.
## We do not have nodes 4 to 6 explicitly in the poset.
p2 <- cbind(c(1L, 1L, 0L), c(2L, 3L, 7L))
plotPoset(p2, addroot = TRUE)

## But this is arguably cleaner
p3 <- cbind(c(1L, 1L, rep(0L, 4)), c(2L, 3L, 4:7 ))
plotPoset(p3, addroot = TRUE)

## A simple way to create a poset where no gene (in a set of 15) depends
## on any other.

p4 <- cbind(0L, 15L)
plotPoset(p4, addroot = TRUE)


## Specifying the pancreatic cancer poset in Gerstung et al., 2011
##   (their figure 2B, left). We use numbers, but for nicer plotting we
##   will use names: KRAS is 1, SMAD4 is 2, etc.
```

```
pancreaticCancerPoset <- cbind(c(1, 1, 1, 1, 2, 3, 4, 4, 5),
                               c(2, 3, 4, 5, 6, 6, 6, 7, 7))
storage.mode(pancreaticCancerPoset) <- "integer"

plotPoset(pancreaticCancerPoset,
          names = c("KRAS", "SMAD4", "CDNK2A", "TP53",
                    "MLL3","PXDN", "TGFBR2"))


## Specifying poset 2 in Figure 2A of Gerstung et al., 2009:

poset2 <- cbind(c(1, 1, 3, 3, 3, 7, 7, 8, 9, 10),
                c(2, 3, 4, 5, 6, 8, 9, 10, 10, 11))

storage.mode(poset2) <- "integer"
plotPoset(poset2)
```

---

`rfitness` *Generate random fitness.*

---

## Description

Generate random fitness landscapes under a House of Cards, Rough Mount Fuji, or additive model.

## Usage

```
rfitness(g, c = 0.5, sd = 1, reference = "random", scale = NULL,
         wt_is_1 = TRUE, log = FALSE, min_accessible_genotypes = 0,
         accessible_th = 0)
```

## Arguments

| | |
|---|---|
| `g` | Number of genes. |
| `c` | The decrease in fitness of a genotype per each unit increase in Hamming distance from the reference genotype (see `reference`). |
| `sd` | The standard deviation of the random component (a normal distribution of mean 0 and standard deviation `sd`). |
| `reference` | The reference genotype: for the deterministic, additive part, this is the genotype with maximal fitness, and all other genotypes decrease their fitness by `c` for every unit of Hamming distance from this reference. If "random" a genotype will be randomly chosen as the reference. If "max" the genotype with all positions mutated will be chosen as the reference. If you pass a vector (e.g., `reference = c(1, 0, 1, 0)`) that will be the reference genotype. If "random2" a genotype will be randomly chosen as the reference. In contrast to "random", however, not all genotypes have the same probability of being chosen; here, what is equal is the probability that the reference genotype has 1, 2, ..., g, mutations (and, once a number mutations is chosen, all genotypes with that number of mutations have equal probability of being the reference). |

`scale` Either NULL (nothing is done) or a two-element vector. If a two-element vector, fitness is re-scaled between `scale[1]` (the minimum) and `scale[2]` (the maximum).

`wt_is_1` If TRUE, fitness will be scaled so that the wildtype (the genotype with no mutations) has fitness of 1. This is applied after `scale`, so if you specify both it is most likely that the final fitness will not respect the limits in `scale`.

`log` If TRUE, log-transform fitness.

`min_accessible_genotypes`
If larger than 0, the minimum number of accessible genotypes in the fitness landscape. A genotype is considered accessible if you can reach if from the wildtype by going through at least one path where all changes in fitness are larger or equal to `accessible_th`. The changes in fitness are considered at each mutational step, i.e., at each addition of one mutation we compute the difference between the genotype with `k + 1` mutations minus the ancestor genotype with `k` mutations. Thus, a genotype is considered accessible if there is at least one path where fitness increases at each mutational step by at least `accessible_th`.

If the condition is not satisfied, we continue generating random fitness landscapes with the specified parameters until the condition is satisfied.

`accessible_th` The threshold for the minimal change in fitness at each mutation step (i.e., between successive genotypes) that allows a genotype to be regarded as accessible. This only applies if `min_accessible_genotypes` is larger than 0. So if you want to allow small decreases in fitness in successive steps, use a small negative value for `accessible_th`.

## Details

The model used here follows the Rough Mount Fuji model in Szendro et al., 2013 or Franke et al., 2011. Fitness is given as

$$f(i) = -cd(i, reference) + x_i$$

where $d(i, j)$ is the Hamming distance between genotypes $i$ and $j$ (the number of positions that differ) and $x_i$ is a random variable (in this case, a normal deviate of mean 0 and standard deviation `sd`).

Setting $c = 0$ we obtain a House of Cards model. Setting $sd = 0$ fitness is given by the distance from the reference and if the reference is the genotype with all positions mutated, then we have a fully additive model (fitness increases linearly with the number of positions mutated).

## Value

An matrix with `g + 1` columns. Each column corresponds to a gene, except the last one that corresponds to fitness. 1/0 in a gene column denotes gene mutated/not-mutated. (For ease of use in other functions, this matrix has class "genotype_fitness_matrix".)

If you have specified `min_accessible_genotypes > 0`, the return object has added attributes `accessible_genotypes` and `accessible_th` that show the number of accessible genotypes under the specified threshold.

## Author(s)

Ramon Diaz-Uriarte

## References

Szendro I.~G. et al. (2013). Quantitative analyses of empirical fitness landscapes. *Journal of Statistical Mehcanics: Theory and Experiment\/*, **01**, P01005.

Franke, J. et al. (2011). Evolutionary accessibility of mutational pathways. *PLoS Computational Biology\/*, **7**(8), 1–9.

## See Also

oncoSimulIndiv, plot.genotype_fitness_matrix, evalAllGenotypes allFitnessEffects plotFitnessLandsca

## Examples

```
## Random fitness for four genes-genotypes,
## plotting and simulating an oncogenetic trajectory

r1 <- rfitness(4)
plot(r1)
oncoSimulIndiv(allFitnessEffects(genotFitness = r1))
```

---

samplePop *Obtain a sample from a population of simulations.*

---

## Description

Obtain a sample (a matrix of individuals/samples by genes or, equivalently, a vector of "genotypes") from an oncosimulpop object (i.e., a simulation of multiple individuals) or a single oncosimul object. Sampling schemes include whole tumor and single cell sampling, and sampling at the end of the tumor progression or during the progression of the disease.

## Usage

```
samplePop(x, timeSample = "last", typeSample = "whole",
          thresholdWhole = 0.5, geneNames = NULL, popSizeSample = NULL)
```

## Arguments

| | |
|---|---|
| x | An object of class oncosimulpop. |
| timeSample | "last" means to sample each individual in the very last time period of the simulation. "unif" (or "uniform") means sampling each individual at a time choosen uniformly from all the times recorded in the simulation between the time when the first driver appeared and the final time period. "unif" means that it is almost sure that different individuals will be sampled at different times. "last" does not guarantee that different individuals will be sampled at the same time unit, only that all will be sampled in the last time unit of their simulation.<br>You can, alternatively, specify the population size at which you want the sample to be taken. See argument popSizeSample . |
| typeSample | "singleCell" (or "single") for single cell sampling, where the probability of sampling a cell (a clone) is directly proportional to its population size. "wholeTumor" (or "whole") for whole tumor sampling (i.e., this is similar to a biopsy being the entire tumor). |

`thresholdWhole` In whole tumor sampling, whether a gene is detected as mutated depends on thresholdWhole: a gene is considered mutated if it is altered in at least thresholdWhole proportion of the cells in that individual.

`geneNames` An optional vector of gene names so as to label the column names of the output.

`popSizeSample` An optional vector of total population sizes at which you want the samples to be taken. If you pass this vector, `timeSample` has no effect. The samples will be taken at the first time at which the population size gets as large as (or larger than) the size specified in `popSizeSample`.

This allows you to specify arbitrary sampling schemes with respect to total population size.

## Details

samplePop simply repeats the sampling process in each individual of the oncosimulpop object.

Please see `oncoSimulSample` for a much more efficient way of sampling when you are sure what you want to sample.

Note that if you have set `onlyCancer = FALSE` in the call to `oncoSimulSample`, you can end up trying to sample from simulations where the population size is 0. In this case, you will get a vector/matrix of NAs and a warning.

Similarly, when using `timeSample = "last"` you might end up with a vector of 0 (not NAs) because you are sampling from a population that contains no clones with mutated genes. This event (sampling from a population that contains no clones with mutated genes), by construction, cannot happen when `timeSample = "unif"` as "uniform" sampling is taken here to mean sampling at a time choosen uniformly from all the times recorded in the simulation between the time when the first driver appeared and the final time period. However, you might still get a vector of 0, with uniform sampling, if you sample from a population that contains only a few cells with any mutated genes, and most cells with no mutated genes.

## Value

A matrix. Each row is a "sample genotype", where 0 denotes no alteration and 1 alteration. When using v.2, columns are named with the gene names.

We quote "sample genotype" because when not using single cell, a row (a sample genotype) need not be, of course, any really existing genotype in a population as we are genotyping a whole tumor. Suppose there are really two genotypes present in the population, genotype A, which has gene A mutated and genotype B, which has gene B mutated. Genotype A has a frequency of 60% (so B's frequency is 40%). If you use whole tumor sampling with `thresholdWhole = 0.4` you will obtain a genotype with A and B mutated.

## Author(s)

Ramon Diaz-Uriarte

## References

Diaz-Uriarte, R. (2015). Identifying restrictions in the order of accumulation of mutations during tumor progression: effects of passengers, evolutionary models, and sampling http://www.biomedcentral.com/1471-2105/16/41/abstract

## See Also

`oncoSimulPop`, `oncoSimulSample`

## Examples

```
data(examplePosets)
p705 <- examplePosets[["p705"]]

## (I set mc.cores = 2 to comply with --as-cran checks, but you
##  should either use a reasonable number for your hardware or
##  leave it at its default value).

p1 <- oncoSimulPop(4, p705, mc.cores = 2)
samplePop(p1)

## Sample at fixed sizes. Notice the requested size
## for the last population is larger than the any population size
## so we get NAs

samplePop(p1, popSizeSample = c(1e7, 1e6, 4e5, 1e13))


## Now single cell sampling

r1 <- oncoSimulIndiv(p705)
samplePop(r1, typeSample = "single")
```

---

`simOGraph` *Simulate oncogenetic/CBN/XMPN DAGs.*

---

## Description

Simulate DAGs that represent restrictions in the accumulation of mutations.

## Usage

```
simOGraph(n, h = 4, conjunction = TRUE, nparents = 3,
multilevelParent = TRUE, removeDirectIndirect = TRUE, rootName = "Root")
```

## Arguments

| | |
|---|---|
| `n` | Number of nodes, or edges, in the graph. Like the number of genes. |
| `h` | Approximate height of the graph. See details. |
| `conjunction` | If TRUE, conjunctions (i.e., multiple parents for a node) are allowed. |
| `nparents` | Maximum number of parents of a node, when `conjunction` is TRUE. |
| `multilevelParent` | Can a node have parents at different heights (i.e., parents that are at different distance from the root node)? |
| `removeDirectIndirect` | Ensure that no two nodes are connected both directly (i.e., with an edge between them) and indirectly, through intermediate nodes. If TRUE, the direct connections are removed from the graph starting from the bottom. |
| `rootName` | The name you want to give the "Root" node. |

### Details

This is a simple, heuristic procedure for generating graphs of restrictions that seem compatible with published trees in the oncogenetic literature.

The basic procedure is as follows: nodes (argument `n`) are split into approximately equally sized `h` groups, and then each node from a level is connected to nodes chosen randomly from nodes of the remaing superior (i.e., closer to the Root) levels. The number of edges comes from a uniform distribution between 1 and `nparents`.

The actual depth of the graph can be smaller than `h` because nodes from a level might be connected to superior levels skipping intermediate ones.

See the vignette for further discussion about arguments.

### Value

An adjacency matrix for a directed graph.

### Author(s)

Ramon Diaz-Uriarte

### Examples

```
(a1 <- simOGraph(10))
library(graph) ## for simple plotting
plot(as(a1, "graphNEL"))
```

---

## to_Magellan *Create output for MAGELLAN.*

---

### Description

Create a fitness landscape in a format that is understood by MAGELLAN http://wwwabi.snv.jussieu.fr/public/Magellan/.

### Usage

```
to_Magellan(x, file,
            max_num_genotypes = 2000)
```

### Arguments

| | |
|---|---|
| x | One of the following:<br>• A matrix (or data frame) with g + 1 columns. Each of the first g columns contains a 1 or a 0 indicating that the gene of that column is mutated or not. Column g+ 1 contains the fitness values. This is, for instance, the output you will get from `rfitness`.<br>• A two column data frame. The second column is fitness, and the first column are genotypes, given as a character vector. For instance, a row "A, B" would mean the genotype with both A and B mutated.<br>• The output from a call to `evalAllGenotypes`. Make sure you use `order = FALSE` in that call. |

- The output from a call to evalAllGenotypesMut. Make sure you use order = FALSE.
- The output from a call to allFitnessEffects (with no order effects in the specification).

The first two are the same as the format for the genotFitness component in allFitnessEffects.

file The name of the output file. If NULL, a name will be created using tempfile.

max_num_genotypes
Maximum allowed number of genotypes. For some types of input, we make a call to evalAllGenotypes, and use this as the maximum.

## Value

A file is written to disk. You can then plot and/or show summary statistics using MAGELLAN.

## Note

If you try to pass a fitness specification with order effects you will receive an error, since that cannot be plotted with MAGELLAN.

## Author(s)

Ramon Diaz-Uriarte

## References

MAGELLAN web site: http://wwwabi.snv.jussieu.fr/public/Magellan/

Brouillet, S. et al. (2015). MAGELLAN: a tool to explore small fitness landscapes. *bioRxiv*, **31583**. http://doi.org/10.1101/031583

## See Also

allFitnessEffects, evalAllGenotypes, allFitnessEffects, rfitness

## Examples

```
## Generate random fitness for four-genes genotype
## and export landscape.

r1 <- rfitness(4)
to_Magellan(r1, NULL)

## Specify fitness using a DAG and export it
cs <-  data.frame(parent = c(rep("Root", 3), "a", "d", "c"),
                      child = c("a", "b", "d", "e", "c", "f"),
                      s = 0.1,
                      sh = -0.9,
                      typeDep = "MN")

to_Magellan(allFitnessEffects(cs), NULL)
```

# Index


∗Topic **datagen**
rfitness, 44
simOGraph, 48
∗Topic **datasets**
example-missing-drivers, 12
examplePosets, 13
examplesFitnessEffects, 14
mcfLs, 14
∗Topic **graphs**
simOGraph, 48
∗Topic **hplot**
plot.fitnessEffects, 28
plot.oncosimul, 30
plotClonePhylog, 36
plotFitnessLandscape, 37
plotPoset, 41
∗Topic **iteration**
oncoSimulIndiv, 15
∗Topic **list**
allFitnessEffects, 2
∗Topic **manip**
allFitnessEffects, 2
OncoSimulWide2Long, 27
poset, 42
samplePop, 46
to_Magellan, 49
∗Topic **misc**
evalAllGenotypes, 8
oncoSimulIndiv, 15

allFitnessEffects, 2, *9, 10*, *14*, *17, 18*, *24*, *28, 29*, *39, 40*, *43*, *46*, *50*
allMutatorEffects, *9*, *17*
allMutatorEffects *(*allFitnessEffects*)*, 2

brewer.pal, *32*

colorRampPalette, *32*

evalAllGenotypes, 8, *39, 40*, *46*, *49, 50*
evalAllGenotypesFitAndMut *(*evalAllGenotypes*)*, 8
evalAllGenotypesMut, *39*, *50*
evalAllGenotypesMut *(*evalAllGenotypes*)*, 8
evalGenotype, *5*, *20*
evalGenotype *(*evalAllGenotypes*)*, 8
evalGenotypeFitAndMut, *5*
evalGenotypeFitAndMut *(*evalAllGenotypes*)*, 8
evalGenotypeMut *(*evalAllGenotypes*)*, 8
ex_missing_drivers_b11 *(*example-missing-drivers*)*, 12
ex_missing_drivers_b12 *(*example-missing-drivers*)*, 12
example-missing-drivers, 12
examplePosets, 13, *24*, *41*, *43*
examplesFitnessEffects, 14

get.adjacency, *36*
ggrepel, *39*

mcfLs, 14
mclapply, *21*

oncoSimulIndiv, *3–5*, 15, *27*, *34*, *36, 37*, *43*, *46*
oncoSimulPop, *47*
oncoSimulPop *(*oncoSimulIndiv*)*, 15
oncoSimulSample, *47*
oncoSimulSample *(*oncoSimulIndiv*)*, 15
OncoSimulWide2Long, 27

par, *32*
plot.default, *33*
plot.evalAllGenotypes *(*plotFitnessLandscape*)*, 37
plot.evalAllGenotypesMut *(*plotFitnessLandscape*)*, 37
plot.fitnessEffects, *5*, 28, *38*, *40*
plot.genotype_fitness_matrix, *46*
plot.genotype_fitness_matrix *(*plotFitnessLandscape*)*, 37
plot.oncosimul, *12*, *15*, *24*, 30
plot.oncosimulpop *(*plot.oncosimul*)*, 30
plotClonePhylog, *19*, 36
plotFitnessLandscape, *5*, *28, 29*, 37, *46*
plotPoset, 41, *43*
polygon, *34*

poset, *3*, *13*, *17*, *41*, 42
print.oncosimul *(*oncoSimulIndiv*)*, 15
print.oncosimulpop *(*oncoSimulIndiv*)*, 15

rfitness, *4, 5*, *39, 40*, 44, *49, 50*

samplePop, *20*, *24*, 46
simOGraph, 48
summary.oncosimul *(*oncoSimulIndiv*)*, 15
summary.oncosimulpop *(*oncoSimulIndiv*)*, 15

tempfile, *50*
to_Magellan, 49