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.

This function can be used also to produce the fitness specification needed to run simulations in a frequency dependent fitness way. In that situation we presume that the effects must be considered as fitness effects and never as mutator effects (see `details` for more info).

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, frequencyDependentFitness = FALSE,
  spPopSizes = NULL)

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 de`details` pend 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).

  This paramenter is not used if `frequencyDependentFitness` is TRUE.

`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.

  This paramenter is not used if `frequencyDependentFitness` is TRUE.

`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.

  This paramenter is not used if `frequencyDependentFitness` is TRUE.

`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.

  This paramenter is not used if `frequencyDependentFitness` is TRUE.

`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. This paramenter is not used if `frequencyDependentFitness` is TRUE.
`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 default 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 (where g > 1). 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. Of course, except for column or row names, all entries in this matrix or data frame must be numeric, except when `frequencyDependentFitness` is TRUE. In this case last column must be character and contains fitness ecuations. • 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. If `frequencyDependentFitness` is TRUE both columns must be character vectors. When `frequencyDependentFitness` is FALSE, fitness must be `>= 0`. If any possible genotype is missing, its fitness is assumed to be 1. By the contrary, if `frequencyDependentFitness` is TRUE, the Fitness column must contain the fitness specification ecuations, like characters, using as variables the frequency of the all possible genotypes. Frequency variables must be f_ for wild type, f_1 or f_A if first gene is mutated, f_2 or f_B if is the case for the second one, f_1_2 or f_A_B, if both are mutated, and so on. Mathematical operations and symbols allowed are described in the documentation of C++ library ExprTkthat is used to parse an evaluate the fitness ecuations (see references for more information).
`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.
`frequencyDependentFitness`	If FALSE, the default value, all downstream work will be realised in a way not related to frequency depedent fitness situations. That implies that fitness specifications are fixed, except death rate in case of McFarland model (see `oncoSimulIndiv` for more details). If TRUE, you are in a frequency dependent fitness situation, where fitness specification ecuations must be passed as characters at `genotFitness`.
`spPopSizes`	`spPopSizes` is a numeric vector that contains the population sizes of the clones, in the same order of getotypes appear in the Genotype column of `genotFitness`. `spPopSizes` is only needed when `frequencyDependentFitness` = TRUE and you want to evaluate fitness with `evalGenotype` or `evalAllGenotypes` (see these functions for more info).

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. `allFitnessEffects` produce the output necessary to pass to the C++ code the fitness/mutator specifications to run simulations. 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. When you run the simulations in `frequencyDependentFitness` = TRUE only fitness effects are allowed, and must be codified in `genotFitness`.

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. This is always the case when `frequencyDependentFitness` = TRUE.

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 in case of `frequencyDependentFitness` = FALSE. By the contrary when `frequencyDependentFitness` = TRUE you are totaly free to determine the fitness as a function of the frequencies of the genotypes (see `genotFitness` and the vignette).

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.
`fitnessLandscape`	A data.frame that contains number of genes + 1 columns, where the first columns are the genes (1 if mutated and 0 if not) and the last one contains the fitnesses.
`fitnessLandscape_df`	A data.frame with the same information of `fitnessLandscape`, but in this case ther are only two columns: Genotype, that has genotypes as vectors codified as characters, and Fitness.
`fitnessLandscape_gene_id`	A data.frame with two columns (Gene and GeneNumID), that map by rows genes as letters (Gene) with genes as numbers (GeneNumID).
`fitnessLandscapeVariables`	A character vector that contains the frequency variables necessary to C++ code.
`frequencyDependentFitness`	TRUE or FALSE as we have explained before.
`spPopSizes`	A numeric vector of population sizes as it had been passsed in the input, if `frequencyDependentFitness` is TRUE, or NULL, if `frequencyDependentFitness` is FALSE.

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.

By the moment the variables you need to specify the frequencies in the fitness ecuations when you are in a frequency dependent fitness situation are fixed as we have explained in `genotFitness`. Using different and strange combinations of "f_" followed by letters and numbers perhaps you could confuse the R parser, but never the C++ one. For a correct performance please be aware of this.

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.

Partow, A. ExprTk: C++ Mathematical Expression Library. (Open Souce MIT License) <http://www.partow.net/programming/exprtk/>

See Also

`evalGenotype`, `evalAllGenotypes`, `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)


##Frequency Dependent Fitness
genofit <- data.frame(A = c(0, 1, 0, 1),
                      B = c(0, 0, 1, 1),
                      Fitness = c("max(3, 2*f_)",
                                  "max(1.5, 3*(f_ + f_1))",
                                  "max(2, 3*(f_ + f_2))",
                                  "max(2, 5*f_ - 0.5*( f_1 + f_2) + 15*f_1_2)"),
                      stringsAsFactors = FALSE)

afe <- allFitnessEffects(genotFitness = genofit,
                         frequencyDependentFitness = TRUE,
                         spPopSizes = c(5000, 2500, 3000, 7500))
##Ploting fitness landscape in case of spPopSizes = c(5000, 2500, 3000, 7500)
plotFitnessLandscape(evalAllGenotypes(afe))

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