Package ‘gRain’

July 30, 2020

Version 1.3-6
Title Graphical Independence Networks
Author Søren Højsgaard <sorenh@math.aau.dk>
Maintainer Søren Højsgaard <sorenh@math.aau.dk>
Description Probability propagation in graphical independence networks, also
known as Bayesian networks or probabilistic expert systems. Documentation
of the package is provided in vignettes included in the package and in
See ‘citation(“gRain”)’ for details.
License GPL (>= 2)
Depends R (>= 3.6.0), methods, gRbase (>= 1.8.6.6)
Suggests microbenchmark, knitr, testthat (>= 2.1.0)
Imports graph, Rgraphviz, igraph, stats4, magrittr, Rcpp (>= 0.11.1)
URL http://people.math.aau.dk/~sorenh/software/gR/
Encoding UTF-8
VignetteBuilder knitr
LinkingTo Rcpp (>= 0.11.1), RcppArmadillo, RcppEigen, gRbase
ByteCompile Yes
biocViews
RoxygenNote 7.1.1
NeedsCompilation yes
Repository CRAN
Date/Publication 2020-07-30 05:00:11 UTC

R topics documented:

chest ................................................................. 2
components_extract ............................................. 3
components_gather .............................................. 5
Chest clinic example

Description

Conditional probability tables for the chest clinic example.

Usage

data(chest_cpt)

Format

An object of class list of length 8.

Examples

## 'data' generated with the following code fragment
yn <- c("yes", "no")
a <- cptable(~asia, values=c(1,99), levels=yn)
t.a <- cptable(~tub|asia, values=c(5,95,1,99), levels=yn)
s <- cptable(~smoke, values=c(5,5), levels=yn)
l.s <- cptable(~lung|smoke, values=c(1,9,1,99), levels=yn)
b.s <- cptable(~bronc|smoke, values=c(6,4,3,7), levels=yn)
e.lt <- cptable(~either|lung:tub, values=c(1,0,1,0,1,0,0,1), levels=yn)
x.e <- cptable(~xray|either, values=c(98,2,5,95), levels=yn)
d.be <- cptable(~dysp|bronc:either, values=c(9,1,7,3,8,2,1,9), levels=yn)
components_extract

grain(compileCPT(a, t.a, s, l.s, b.s, e.lt, x.e, d.be))

# 'data' generated from
# chest_cpt <- list(a, t.a, s, l.s, b.s, e.lt, x.e, d.be)

data(chest_cpt)

---

components_extract  Extract conditional probabilities and clique potentials from data.

Description

Extract list of conditional probability tables and list of clique potentials from data.

Usage

extractCPT(data_, graph, smooth = 0)
extractPOT(data_, graph, smooth = 0)
extractMARG(data_, graph, smooth = 0)
marg2pot(mg)
pot2marg(pt)

Arguments

data_  A named array or a dataframe.

graph  A graphNEL object or a list or formula which can be turned into a graphNEL object by calling ug or dag. For extractCPT, graph must be/define a DAG while for extractPOT, graph must be/define undirected triangulated graph.

smooth  See 'details' below.

mg  An object of class marg_rep

pt  An object of class pot_rep

Details

If smooth is non-zero then smooth is added to all cell counts before normalization takes place.

Value

- extractCPT: A list of conditional probability tables.
- extractPOT: A list of clique potentials.
- extractMARG: A list of clique marginals.
Author(s)
Søren Højsgaard, <sorenh@math.aau.dk>

References

See Also
compileCPT, compilePOT, grain

Examples

```r
## Extract cpts / clique potentials from data and graph
# specification and create network. There are different ways:

data(lizard, package="gRbase")

# DAG: height <- species -> diam
daG <- dag(~species + height:species + diam:species)

# UG : [height:species][diam:species]
uG <- ug(~height:species + diam:species)

pt <- extractPOT(lizard, ~height:species + diam:species)
cp <- extractCPT(lizard, ~species + height:species + diam:species)

pt
cp

# Both specify the same probability distribution
tabListMult(pt) %>% as.data.frame.table
tabListMult(cp) %>% as.data.frame.table

## Not run:
# Bayesian networks can be created as
bn.uG <- grain(pt)
bn.daG <- grain(cp)

# The steps above are wrapped into a convenience method which
# builds a network from at graph and data.
bn.uG <- grain(uG, data=lizard)
bn.daG <- grain(daG, data=lizard)

## End(Not run)
```
Compile conditional probability tables / cliques potentials.

Description

Compile conditional probability tables / cliques potentials as a preprocessing step for creating a graphical independence network.

Usage

```r
compileCPT(x, ..., forceCheck = TRUE)
compilePOT(x, ..., forceCheck = TRUE)
parse_cpt(xi)
```

Arguments

- `x`: To `compileCPT` `x` is a list of conditional probability tables; to `compilePOT` `x` is a list of clique potentials.
- `...`: Additional arguments; currently not used.
- `forceCheck`: Controls if consistency checks of the probability tables should be made.
- `xi`: cpt in some representation

Details

* 'compileCPT' is relevant for turning a collection of `cptable`'s into an object from which a network can be built. For example, when specification of a `cpt` is made with `cptable` then the levels of the node is given but not the levels of the parents. 'compileCPT' checks that the levels of variables in the `cpt`'s are consistent and also that the specifications define a dag.

* 'compilePOT' is not of direct relevance for the user for the moment. However, the elements of the input should be arrays which define a chordal undirected graph and the arrays should, if multiplied, form a valid probability density.

Value

A list with a class attribute.

Author(s)

Søren Højsgaard, <sorenh@math.aau.dk>
References


See Also

extractCPT, extractPOT, extractMARG

Examples

data(chest_cpt)
x <- compileCPT(chest_cpt)
class(x)
grain(x)

## FIXME: compileCPT/compilePOT examples missing.

cpt-update Update components of Bayesian network

Description

Update components of Bayesian network.

Usage

setCPT(object, value)

## S3 method for class 'cpt_grain'
setCPT(object, value)

Arguments

object A grain object.
value A named list, see examples below.

Author(s)

Søren Højsgaard, <sorenh@math.aau.dk>

References


See Also

grain, propagate, triangulate, rip, junctionTree
Examples

### See the wet grass example at
### https://en.wikipedia.org/wiki/Bayesian_network

```r
yn <- c("yes", "no")
p.R <- cptable(~R, values=c(.2, .8), levels=yn)
p.S_R <- cptable(~S:R, values=c(.01, .99, .4, .6), levels=yn)
p.G_SR <- cptable(~G:S:R, values=c(.99, .01, .8, .2, .9, .1, 0, 1), levels=yn)

x
wet.bn <- grain(x)

getgrain(wet.bn, "cpt")
getgrain(wet.bn, "cpt")$R
getgrain(wet.bn, "cpt")$S

# Now update some cpt's
wet.bn2 <- setCPT(wet.bn, list(R=c(.3, .7), S=c(.1, .9, .7, .3)))

getgrain(wet.bn2, "cpt")$R
getgrain(wet.bn2, "cpt")$S
```

cptable

Create conditional probability tables (CPTs)

Description

Creates conditional probability tables of the form p(v|pa(v)).

Usage

cptable(vpar, levels = NULL, values = NULL, normalize = TRUE, smooth = 0)

Arguments

- **vpar**: Specifications of the names in P(v|pa1,...pakan). See section 'details' for information about the form of the argument.
- **levels**: See 'details' below.
- **values**: Probabilities; recycled if necessary. Regarding the order, please see section 'details' and the examples.
- **normalize**: See 'details' below.
- **smooth**: See 'details' below.
Details

If `normalize=TRUE` then the probabilities are normalized to sum to one for each configuration of the parents.

If `smooth` is non-zero then zero entries of `values` are replaced with `smooth` before normalization takes place.

Regarding the form of the argument `vpar`: To specify \( P(a|b,c) \) one may write \(~a|b::c\), \(~a:b::c\), \(~a|b::c\), \(~a+b+c\) or \(c("a","b","c")\). Internally, the last form is used. Notice that the + and : operator is used as a separator only. The order of the variables IS important so the operators DO NOT commute.

If `a` has levels `a1,a2` and likewise for `b` and `c` then the order of values corresponds to the configurations \((a1,b1,c1)\), \((a2,b1,c1)\), \((a1,b2,c1)\), \((a2,b2,c1)\) etc. That is, the first variable varies fastest. Hence the first two elements in `values` will be the conditional probabilities of `a` given `b=b1,c=c1`.

Value

A `cptable` object (a numeric vector with various attributes).

Author(s)

Søren Højsgaard, <sorenh@math.aau.dk>

References


See Also

`andtable`, `ortable`, `extractCPT`, `compileCPT`, `extractPOT`, `compilePOT`, `grain`, `parray`

Examples

```r
## See the wet grass example at
## https://en.wikipedia.org/wiki/Bayesian_network

yn <- c("yes", "no")
p.R <- cptable(~R, values=c(.2, .8), levels=yn)
p.S_R <- cptable(~S:R, values=c(.01, .99, .4, .6), levels=yn)
p.G_SR <- cptable(~G:S:R, values=c(.99, .01, .8, .2, .9, .1, 0, 1), levels=yn)
# or
ssp <- list(R=yn, S=yn, G=yn) # state space
p.R <- cptable(~R, values=c(.2, .8), levels=ssp)
p.S_R <- cptable(~S:R, values=c(.01, .99, .4, .6), levels=ssp)
p.G_SR <- cptable(~G:S:R, values=c(.99, .01, .8, .2, .9, .1, 0, 1), levels=ssp)
```
evidence_object

# components above are "intermediate representations" and are turned into arrays with
wet.cpt
wet.cpt$S # etc

# A Bayesian network is created with:
wet.bn <- grain(wet.cpt)

# Can also create arrays directly
## Not run:
ssp <- list(R=yn, S=yn, G=yn) # state space
p.R <- c(.2, .8)
p.S_R <- c(.01, .99, .4, .6)
p.G_SR <- c(.99, .01, .8, .9, .1, 0, 1)
dim(p.R) <- 2
dimnames(p.R) <- ssp["R"]
dim(p.S_R) <- c(2, 2)
dimnames(p.S_R) <- ssp[c("S", "R")]
dim(p.G_SR) <- c(2, 2, 2)
dimnames(p.G_SR) <- ssp[c("G", "S", "R")]

# Arrays can be created (easier?) with parray() from gRbase
p.R <- parray("R", levels=ssp, values=c(.2, .8))
p.S_R <- parray(c("S", "R"), levels = ssp, values=c(.01, .99, .4, .6))
p.G_SR <- parray(~ G:S:R, levels = ssp, values=c(.99, .01, .8, .9, .1, 0, 1))

## End(Not run)

---

### evidence_object

**Evidence objects**

**Description**

Functions for defining and manipulating evidence.

**Usage**

new_ev(evi.list = NULL, levels)

is.null_ev(object)

## S3 method for class 'grain_ev'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)

setdiff_ev(ev1, ev2)

union_ev(ev1, ev2)
Arguments

evi.list A named list with evidence; see 'examples' below.
levels A named list with the levels of all variables.
object Some R object.
x An evidence object.
row.names Not used.
optional Not used.
... Not used.
ev1, ev2 Evidence.

Details

Evidence is specified as a list. Internally, evidence is represented as a grain evidence object which is a list with 4 elements.

Author(s)

Søren Højsgaard, <sorenh@math.aau.dk>

Examples

## Define the universe

uni <- list(asia = c("yes", "no"), tub = c("yes", "no"), smoke = c("yes", "no"),
            lung = c("yes", "no"), bronc = c("yes", "no"), either = c("yes", "no"),
            xray = c("yes", "no"), dysp = c("yes", "no"))

e1 <- list(dysp="no", xray="no")
eo1 <- new_ev( e1, levels=uni )
eo1
as.data.frame( eo1 )
eo1 %>% str

e1.2 <- list(dysp="no", xray=c(0, 1))
eo1.2 <- new_ev( e1.2, levels=uni )
eo1.2

# Notice that in eo1.2, xray is not regarded as hard
# evidence but as a weight on each level. Other than that, eo1.2
# and eo1 are equivalent here. This is used in connection
# with specifying likelihood evidence.

e2 <- list(dysp="yes", asia="yes")
eo2 <- new_ev(e2, uni)

# If evidence 'e1' is already set in the network and new evidence
# 'e2' emerges, the evidence in the network must be updated. But
# there is a conflict in that dysp="yes" in 'e1' and
finding

# dyssp="no" in 'e2'. The (arbitrary) convention is that
# existing evidence overrides new evidence so that the only new
# evidence in 'e2' is really asia="yes".

# To subtract existing evidence from new evidence we can do:
setdiff_ev( eo2, eo1 )

# Likewise the 'union' is
union_ev( eo2, eo1 )

finding

Set, retrieve, and retract finding in Bayesian network.

Description

Set, retrieve, and retract finding in Bayesian network. NOTICE: The functions described here are kept only for backward compatibility; please use the corresponding evidence-functions in the future.

Usage

setFinding(object, nodes = NULL, states = NULL, flist = NULL, propagate = TRUE)

Arguments

object A "grain" object
nodes A vector of nodes
states A vector of states (of the nodes given by 'nodes')
flist An alternative way of specifying findings, see examples below.
propagate Should the network be propagated?

Note

NOTICE: The functions described here are kept only for backward compatibility; please use the corresponding evidence-functions in the future:

setEvidence() is an improvement of setFinding() (and as such setFinding is obsolete). Users are recommended to use setEvidence() in the future.

setEvidence() allows to specification of "hard evidence" (specific values for variables) and likelihood evidence (also known as virtual evidence) for variables.

The syntax of setEvidence() may change in the future.

Author(s)

Søren Højsgaard, <sorenh@math.aau.dk>
References


See Also

setEvidence, getEvidence, retractEvidence, pEvidence, querygrain

Examples

```r
## setFindings
yn <- c("yes", "no")
a <- cptable(~asia, values=c(1,99), levels=yn)
t.a <- cptable(~tub+asia, values=c(5,95,1,99), levels=yn)
s <- cptable(~smoke, values=c(5,5), levels=yn)
l.s <- cptable(~lung+smoke, values=c(1,9,1,99), levels=yn)
b.s <- cptable(~bronc+smoke, values=c(6,4,3,7), levels=yn)
e.lt <- cptable(~either+lung+tub, values=c(1,0,1,0,1,0,0,1), levels=yn)
x.e <- cptable(~xray+either, values=c(98,2,95,98,2,95), levels=yn)
d.be <- cptable(~dysp+bronc+either, values=c(9,1,7,3,8,2,1,9), levels=yn)
chest.cpt <- compileCPT(a, t.a, s, l.s, b.s, e.lt, x.e, d.be)
chest.bn <- grain(chest.cpt)

## These two forms are equivalent
bn1 <- setFinding(chest.bn, nodes=c("chest", "xray"), states=c("yes", "yes"))
bn2 <- setFinding(chest.bn, flist=list(c("chest", "yes"), c("xray", "yes")))

gFinding(bn1)
gFinding(bn2)

pFinding(bn1)
pFinding(bn2)

bn1 <- retractFinding(bn1, nodes="asia")
bn2 <- retractFinding(bn2, nodes="asia")

gFinding(bn1)
gFinding(bn2)

pFinding(bn1)
pFinding(bn2)
```
Description

Generic functions etc for the gRain package

Usage

nodeNames(x)

## S3 method for class 'grain'
nodeNames(x)

nodeStates(x, nodes = nodeNames(x))

## S3 method for class 'grain'
nodeStates(x, nodes = nodeNames(x))

universe(object, ...)

## S3 method for class 'grain'
universe(object, ...)

## S3 method for class 'grainEvidence_'
varNames(x)

## S3 method for class 'grain'
rip(object, ...)

## S3 method for class 'cpt_spec'
vpar(object, ...)

## S3 method for class 'cpt_grain'
vpar(object, ...)

Arguments

x, object A relevant object.
nodes Some nodes of the object.
... Additional arguments; currently not used.
Usage

grain(x, ...)

## S3 method for class 'cpt_spec'
grain(x, control = list(), smooth = 0, compile = TRUE, details = 0, ...)

## S3 method for class 'CPTspec'
grain(x, control = list(), smooth = 0, compile = TRUE, details = 0, ...)

## S3 method for class 'pot_spec'
grain(x, control = list(), smooth = 0, compile = TRUE, details = 0, ...)

## S3 method for class 'graphNEL'
grain(
  x,
  control = list(),
  smooth = 0,
  compile = TRUE,
  details = 0,
  data = NULL,
  ...
)

## S3 method for class 'dModel'
grain(
  x,
  control = list(),
  smooth = 0,
  compile = TRUE,
  details = 0,
  data = NULL,
  ...
)

Arguments

x An argument to build an independence network from. Typically a list of conditional probability tables, a DAG or an undirected graph. In the two latter cases, data must also be provided.

... Additional arguments, currently not used.

control A list defining controls, see 'details' below.

smooth A (usually small) number to add to the counts of a table if the grain is built from a graph plus a dataset.

compile Should network be compiled.

details Debugging information.

data An optional data set (currently must be an array/table)
Details

If `smooth` is non-zero then entries of `values` which a zero are replaced by the value of `smooth` - BEFORE any normalization takes place.

Value

An object of class "grain"

Note

A change from earlier versions of this package is that grain objects are now compiled upon creation.

Author(s)

Søren Højsgaard, <sorenh@math.aau.dk>

References


See Also

cptable, compile.grain, propagate.grain, setFinding, setEvidence, getFinding, pFinding, retractFinding, extractCPT, extractPOT, compileCPT, compilePOT

Examples

```r
## Asia (chest clinic) network created from conditional probability tables
yn <- c("yes", "no")
a <- cptable(~asia, values=c(1,99), levels=yn) t.a <- cptable(~tub+asia, values=c(5,95,1,99), levels=yn) s <- cptable(~smoke, values=c(5,5), levels=yn) l.s <- cptable(~lung+smoke, values=c(1,9,1,99), levels=yn) b.s <- cptable(~bronc+smoke, values=c(6,4,3,7), levels=yn) e.lt <- cptable(~either+lung+tub, values=c(1,0,1,0,1,0,1), levels=yn) x.e <- cptable(~xray+either, values=c(98,2,5,95), levels=yn) d.be <- cptable(~dysp+bronc+either, values=c(9,1,7,3,8,2,1,9), levels=yn) chest.cpt <- compileCPT(a, t.a, s, l.s, b.s, e.lt, x.e, d.be) chest.bn <- grain(chest.cpt)

## Create network from data and graph specification.
## There are different ways; see documentation in the "See all"
## links.

data(lizard, package="gRbase")
# DAG: height <- species -> diam
daG <- dag(~species + height:species + diam:species)
```
# UG : [height:species][diam:species]
uG <- ug(~height:species + diam:species)

bn.uG <- grain(uG, data=lizard)
bn.daG <- grain(daG, data=lizard)

---

grain-simulate  Simulate from an independence network

**Description**

Simulate data from an independence network.

**Usage**

```r
## S3 method for class 'grain'
simulate(object, nsim = 1, seed = NULL, ...)
```

**Arguments**

- `object` An independence network.
- `nsim` Number of cases to simulate.
- `seed` An optional integer controlling the random number generation.
- `...` Not used.

**Value**

A data frame

**Author(s)**

Søren Højsgaard, <sorenh@math.aau.dk>

**References**


**Examples**

```r
tf <- system.file("huginex", "chest_clinic.net", package = "gRain")
chest <- loadHuginNet(tf, details=1)
simulate(chest, n=10)

chest2 <- setFinding(chest, c("VisitToAsia", "Dyspnoea"),
                     c("yes", "yes"))
simulate(chest2, n=10)
```
**grain_compile**

Compile a graphical independence network (a Bayesian network)

**Description**

Compiles a Bayesian network. This means creating a junction tree and establishing clique potentials; refer to the reference below for details.

**Usage**

```r
## S3 method for class 'grain'
compile(
  object,
  propagate = FALSE,
  root = NULL,
  control = object$control,
  details = 0,
  ...
)
```

**Arguments**

- `object`: A grain object.
- `propagate`: If TRUE the network is also propagated meaning that the cliques of the junction tree are calibrated to each other.
- `root`: A set of variables which must be in the root of the junction tree
- `control`: Controlling the compilation process.
- `details`: For debugging info. Do not use.
- `...`: Currently not used.

**Value**

A compiled Bayesian network; an object of class grain.

**Author(s)**

Søren Højsgaard, <sorenh@math.aau.dk>

**References**


**See Also**

grain, propagate, propagate.grain, triangulate, rip, junctionTree
grain_evidence

**Set, update and remove evidence.**

**Description**

Set, update and remove evidence.

**Usage**

```r
setEvidence(
  object,
  nodes = NULL,
  states = NULL,
  evidence = NULL,
  nslist = NULL,
  propagate = TRUE,
  details = 0
)
```

```r
retractEvidence(object, nodes = NULL, propagate = TRUE)
```

```r
absorbEvidence(object, propagate = TRUE)
```

```r
pEvidence(object)
```

```r
getEvidence(object)
```

```r
evidence(object)
```

```r
## S3 method for class 'grain'
evidence(object)
```

```r
evidence(object) <- value
```

```r
## S3 replacement method for class 'grain'
evidence(object) <- value
```

**Arguments**

- **object**: A "grain" object
- **nodes**: A vector of nodes; those nodes for which the (conditional) distribution is requested.
- **states**: A vector of states (of the nodes given by 'nodes')
- **evidence**: An alternative way of specifying findings (evidence), see examples below.
- **nslist**: deprecated
propagate Should the network be propagated?
details Debugging information
value The evidence in the form of a named list or an evidence-object.

Value

A list of tables with potentials.

Note

setEvidence() is an improvement of setFinding() (and as such setFinding is obsolete). Users are recommended to use setEvidence() in the future.

setEvidence() allows to specification of "hard evidence" (specific values for variables) and likelihood evidence (also known as virtual evidence) for variables.

The syntax of setEvidence() may change in the future.

Author(s)

Søren Højsgaard, <sorenh@math.aau.dk>

References


See Also

setFinding, getFinding, retractFinding, pFinding

Examples

data(chest_cpt)
chest.bn <- grain(compileCPT(chest_cpt))
chest.bn <- compile(chest.bn)

## 1) These two forms are identical
setEvidence(chest.bn, c("asia", "xray"), c("yes", "yes"))
setFinding(chest.bn, c("asia", "xray"), c("yes", "yes"))

## 2) Suppose we do not know with certainty whether a patient has
## recently been to Asia. We can then introduce a new variable
## "guess.asia" with "asia" as its only parent. Suppose
## p(guess.asia=yes|asia=yes) = .8 and p(guess.asia=yes|asia=no) = .1
## If the patient is e.g. unusually tanned we may set
## guess.asia=yes and propagate.
##
## This corresponds to modifying the model by the likelihood (0.8, 0.1) as
setEvidence(chest.bn, c("asia", "xray"), list(c(0.8, 0.1), "yes"))

## 3) Hence, the same result as in 1) can be obtained with
setEvidence(chest.bn, c("asia", "xray"), list(c(1, 0), "yes"))

## 4) An alternative specification using evidence is
setEvidence(chest.bn, evidence=list(asia=c(1, 0), xray="yes"))

---

grain_jevidence

Set joint evidence in grain objects

Description

Setting and removing joint evidence in grain objects.

Usage

setJEvidence(object, evidence = NULL, propagate = TRUE, details = 0)

retractJEvidence(object, items = NULL, propagate = TRUE, details = 0)

new_jev(ev, levels)

Arguments

- **object**: A "grain" object.
- **evidence**: A list of evidence. Each element is a named array.
- **propagate**: Should evidence be absorbed once entered; defaults to TRUE.
- **details**: Amount of printing; for debugging.
- **items**: Items in the evidence list to be removed. Here, NULL means remove everything, 0 means nothing is removed. Otherwise items is a numeric vector.
- **ev**: A named list.
- **levels**: A named list.

Note

All the joint evidence functionality should be used with great care.

Author(s)

Søren Højsgaard, <sorenh@math.aau.dk>
Examples

data(chest_cpt)
chest.bn <- grain(compileCPT(chest_cpt))
chest.bn <- compile(chest.bn)

uni <- list(asia = c("yes", "no"),
            tub = c("yes", "no"),
            smoke = c("yes", "no"),
            lung = c("yes", "no"),
            bronc = c("yes", "no"),
            either = c("yes", "no"),
            xray = c("yes", "no"),
            dysp = c("yes", "no"))

ev <- list(tabNew("asia", levels=uni, values=c(1,0)),
           tabNew("dysp", levels=uni, values=c(1,0)),
           tabNew(c("dysp","bronc"), levels=uni, values=c(.1, .2, .9, .8)) )

chest.bn
chest.bn2 <- setJEvidence(chest.bn, evidence=ev)

# Notice: The evidence is defined on (subsets of) cliques of the junction tree
# and therefore evidence can readily be absorbed:
getgrain(chest.bn, "rip")$cliques %>% str

# On the other hand, below is evidence which is not defined cliques
# of the junction tree and therefore evidence can not easily be
# absorbed. Hence this will fail:

## Not run:
ev.fail <- list(tab(c("dysp","smoke"), levels=uni, values=c(.1, .2, .9, .8)) )
setJEvidence(chest.bn, evidence=ev.fail)

## End(Not run)

## Evidence can be removed with
retractJEvidence(chest.bn2)       ## All evidence removed.
retractJEvidence(chest.bn2, 0)    ## No evidence removed.
retractJEvidence(chest.bn2, 1:2)  ## Evidence items 1 and 2 are removed.

# Setting additional joint evidence to an object where joint
# evidence already is set will cause an error. Hence this will fail:

## Not run:
ev2 <- list(smoke="yes")
setJEvidence(chest.bn2, evidence=ev2)

## End(Not run)

## Instead we can do
new.ev <- c(getEvidence(chest.bn2), list(smoke="yes"))
chest.bn
setJEvidence(chest.bn, evidence=new.ev)
```r
## Create joint evidence object:
yn <- c("yes", "no")
db <- parray(c("dysp", "bronc"), list(yn, yn), values=c(.1, .2, .9, .8))
db
ev <- list(asia=c(1, 0), dysp="yes", db)
jevi <- new_jev(ev, levels=uni)
jevi
chest.bn3 <- setJEvidence(chest.bn, evidence=jevi)
evidence(chest.bn3)
```

grain_predict

*Make predictions from a probabilistic network*

**Description**

Makes predictions (either as the most likely state or as the conditional distributions) of variables conditional on finding (evidence) on other variables in an independence network.

**Usage**

```r
## S3 method for class 'grain'
predict(
  object,
  response,
  predictors = setdiff(names(newdata), response),
  newdata,
  type = "class",
  ...
)
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>A grain object</td>
</tr>
<tr>
<td>response</td>
<td>A vector of response variables to make predictions on</td>
</tr>
<tr>
<td>predictors</td>
<td>A vector of predictor variables to make predictions from. Defaults to all variables that are note responses.</td>
</tr>
<tr>
<td>newdata</td>
<td>A data frame</td>
</tr>
<tr>
<td>type</td>
<td>If &quot;class&quot;, the most probable class is returned; if &quot;distribution&quot; the conditional distribution is returned.</td>
</tr>
<tr>
<td>...</td>
<td>Not used</td>
</tr>
</tbody>
</table>
**grain_propagate**

**Value**

A list with components

- **pred** A list with the predictions
- **pFinding** A vector with the probability of the finding (evidence) on which the prediction is based

**Author(s)**

Søren Højsgaard, <sorenh@math.aau.dk>

**References**


**See Also**

grain

**Examples**

```r
data(chest_cpt)
data(chestSim500)

chest.bn <- grain(compileCPT(chest_cpt))
nd <- chestSim500[1:4]

predict(chest.bn, response="bronc", newdata=nd)
predict(chest.bn, response="bronc", newdata=nd, type="distribution")
```

---

**Description**

Propagation refers to calibrating the cliques of the junction tree so that the clique potentials are consistent on their intersections; refer to the reference below for details.

**Usage**

```r
## S3 method for class 'grain'
propagate(object, details = object$details, engine = "cpp", ...)

propagateLS(cqpotList, rip, initialize = TRUE, details = 0)
```
Arguments

- **object**: A grain object
- **details**: For debugging info
- **engine**: Either "R" or "cpp"; "cpp" is the default and the fastest.
- **cqpotList**: Clique potential list
- **rip**: A rip ordering
- **initialize**: Always true

Details

The `propagate` method invokes `propagateLS` which is a pure R implementation of the Lauritzen-Spiegelhalter algorithm. The c++ based version is several times faster than the purely R based version.

Value

A compiled and propagated grain object.

Author(s)

Søren Højsgaard, <sorenh@math.aau.dk>

References


See Also

`grain`, `compile`

Examples

```r
yn <- c("yes","no")
a <- cptable(~asia, values=c(1,99), levels=yn)
t.a <- cptable(~tub+asia, values=c(5,95,1,99), levels=yn)
s <- cptable(~smoke, values=c(5,5), levels=yn)
l.s <- cptable(~lung+smoke, values=c(1,9,1,99), levels=yn)
b.s <- cptable(~bronc+smoke, values=c(6,4,3,7), levels=yn)
e.lt <- cptable(~either+lung+tub, values=c(1,0,1,0,1,0,0,1), levels=yn)
x.e <- cptable(~xray+either, values=c(98,2,5,95), levels=yn)
d.be <- cptable(~dysp+bronc+either, values=c(9,1,7,3,8,2,1,9), levels=yn)
chest.cpt <- compileCPT(list(a, t.a, s, l.s, b.s, e.lt, x.e, d.be))
chest.bn <- grain(chest.cpt)
bn1 <- compile(chest.bn, propagate=FALSE)
bn2 <- propagate(bn1)
bn3 <- compile(chest.bn, propagate=TRUE)
```
**grass**

**Wet grass example**

**Description**

Conditional probability tables for the wet grass example.

**Usage**

```r
data(grass_cpt)
```

**Format**

An object of class list of length 3.

**Examples**

```r
## 'data' generated with the following code fragment
yn <- c("yes", "no")
p.R <- cptable(~R, values=c(0.2, 0.8), levels=yn)
p.S_R <- cptable(~S:R, values=c(0.01, 0.99, 0.4, 0.6), levels=yn)
p.G_SR <- cptable(~G:S:R, values=c(0.99, 0.01, 0.8, 0.2, 0.9, 0.1, 0, 1), levels=yn)


# 'data' generated from

data(grass_cpt)
```

---

**load-save-hugin**

**Load and save Hugin net files**

**Description**

These functions can load a net file saved in the 'Hugin format' into R and save a network in R as a file in the 'Hugin format'.

**Usage**

```r
loadHuginNet(file, description = NULL, details = 0)

saveHuginNet(gin, file, details = 0)
```
Arguments

- **file**: Name of HUGIN net file. Convenient to give the file the extension '.net'
- **description**: A text describing the network, defaults to file
- **details**: Debugging information
- **gin**: An independence network

Value

An object of class `grain`.

Author(s)

Søren Højsgaard, <sorenh@math.aau.dk>

References


See Also

`grain`

Examples

```r
## Load HUGIN net file
tf <- system.file("huginex", "chest_clinic.net", package = "gRain")
chest <- loadHuginNet(tf, details=1)
chest

## Save a copy
td <- tempdir()
saveHuginNet(chest, paste(td,"/chest.net",sep=''))

## Load the copy
chest2 <- loadHuginNet(paste(td,"/chest.net",sep=''))

tf <- system.file("huginex", "golf.net", package = "gRain")
golf <- loadHuginNet(tf, details=1)

saveHuginNet(golf, paste(td,"/golf.net",sep=''))
golf2 <- loadHuginNet(paste(td,"/golf.net",sep=''))
```
**Description**

Generate conditional probability tables based on the logical expressions AND and OR.

**Usage**

```r
booltab(vpa, levels = c(TRUE, FALSE), op = `&`)

andtab(vpa, levels = c(TRUE, FALSE))

ortab(vpa, levels = c(TRUE, FALSE))

andtable(vpa, levels = c(TRUE, FALSE))

ortable(vpa, levels = c(TRUE, FALSE))
```

**Arguments**

- `vpa` Node and two parents; as a formula or a character vector.
- `levels` The levels (or rather labels) of v, see 'examples' below.
- `op` A logical operator.

**Details**

Regarding the form of the argument `vpa`: To specify \( P(a|b, c) \) one may write \( \neg a|b+c \) or \( \neg a+b+c \) or \( \neg a|b:c \) or \( \neg a:b:c \) or \( c(\"a\", \"b\", \"c\") \). Internally, the last form is used. Notice that the + and : operator are used as separators only. The order of the variables is important so + and : DO NOT commute.

**Value**

An array.

**Note**

`andtable` and `ortable` are aliases for `andtab` and `ortab` and are kept for backward compatibility.

**Author(s)**

Søren Højsgaard, <sorenh@math.aau.dk>

**References**

## Examples

### Logical OR:

```r
## A variable v is TRUE if either of its parents pa1 and pa2 are TRUE:
ortab(c("v", "pa1", "pa2")) %>% ftable(row.vars="v")
## TRUE and FALSE can be recoded to e.g. yes and no:
ortab(c("v", "pa1", "pa2"), levels=c("yes", "no")) %>% ftable(row.vars="v")
```

### Logical AND:

```r
## Same story here:
andtab(c("v", "pa1", "pa2")) %>% ftable(row.vars="v")
andtab(c("v", "pa1", "pa2"), levels=c("yes", "no")) %>% ftable(row.vars="v")
```

### Combined approach

```r
booltab(c("v", "pa1", "pa2"), op="\&") %>% ftable(row.vars="v") ## AND
booltab(c("v", "pa1", "pa2"), op="|") %>% ftable(row.vars="v") ## OR

booltab(~v + pa1 + pa2, op="\&") %>% ftable(row.vars="v") ## AND
booltab(~v + pa1 + pa2, op="|") %>% ftable(row.vars="v") ## OR
```

---

**Description**

Generate conditional probability table for mendelian segregation.

**Usage**

```r
mendel(allele, names = c("child", "father", "mother"))
```

**Arguments**

- `allele` A character vector.
- `names` Names of columns in dataframe.

**Note**

No error checking at all on the input.
Examples

```r
## Inheritance of the alleles "y" and "g"

men <- mendel(c("y","g"), names=c("ch", "fa", "mo"))
men
```

querygrain

Query a network

Description

Query an independence network, i.e. obtain the conditional distribution of a set of variables - possibly (and typically) given finding (evidence) on other variables.

Usage

```r
querygrain(
  object,
  nodes = nodeNames(object),
  type = "marginal",
  evidence = NULL,
  exclude = TRUE,
  normalize = TRUE,
  result = "array",
  details = 0
)
```

Arguments

- **object**: A grain object.
- **nodes**: A vector of nodes; those nodes for which the (conditional) distribution is requested.
- **type**: Valid choices are "marginal" which gives the marginal distribution for each node in nodes; "joint" which gives the joint distribution for nodes and "conditional" which gives the conditional distribution for the first variable in nodes given the other variables in nodes.
- **evidence**: An alternative way of specifying findings (evidence), see examples below.
- **exclude**: If TRUE then nodes on which evidence is given will be excluded from nodes (see above).
- **normalize**: Should the results be normalized to sum to one.
- **result**: If "data.frame" the result is returned as a data frame (or possibly as a list of dataframes).
- **details**: Debugging information
Value
A list of tables with potentials.

Note
setEvidence() is an improvement of setFinding() (and as such setFinding is obsolete). Users are recommended to use setEvidence() in the future.
setEvidence() allows to specification of "hard evidence" (specific values for variables) and likelihood evidence (also known as virtual evidence) for variables.
The syntax of setEvidence() may change in the future.

Author(s)
Søren Højsgaard, <sorenh@math.aau.dk>

References

See Also
setEvidence, getEvidence, retractEvidence, pEvidence

Examples

testfile <- system.file("huginex", "chest_clinic.net", package = "gRain")
chest <- loadHuginNet(testfile, details=0)
qb <- querygrain(chest)
qb

lapply(qb, as.numeric) # Safe
sapply(qb, as.numeric) # Risky

repeatPattern(plist, instances, unlist = TRUE)
Arguments

**plist** A list of conditional probability tables. The variable names must have the form `name[i]` and the `i` will be substituted by the values given in `instances` below.

**instances** A vector of distinct integers

**unlist** If FALSE the result is a list in which each element is a copy of `plist` in which `name[i]` are substituted. If TRUE the result is the result of applying `unlist()`.

Author(s)

Søren Højsgaard, <sorenh@math.aau.dk>

References


See Also

grain, compileCPT

Examples

```r
## Example: Markov chain
yn <- c("yes", "no")

## Specify p(x0)
x.0 <- cptable(~x0, values=c(1, 9), levels=yn)

## Specify transition density
x.x <- cptable(~x[i]|x[i-1], values=c(1, 99, 2, 98), levels=yn)

## Pattern to be repeated
pat <- list(x.x)
rep.pat <- repeatPattern(pat, instances=1:5)
apt <- compileCPT(c(list(x.0), rep.pat))
mch <- grain(cpt)
if (interactive()) iplot(mch)

## Example: Hidden markov model: The x[i]'s are unobserved, the
## y[i]'s can be observed.
yn <- c("yes", "no")

## Specify p(x0)
x.0 <- cptable(~x0, values=c(1, 9), levels=yn)

## Specify transition density
x.x <- cptable(~x[i]|x[i-1], values=c(1, 99, 2, 98), levels=yn)
```

```
## Specify emission density

```r
y.x <- cptable(~y[i]|x[i], values=c(10, 90, 20, 80), levels=yn)
```

## The pattern to be repeated

```r
pat <- list(x.x, y.x)
```

## Repeat pattern and create network

```r
rep.pat <- repeatPattern(pat, instances=1:5)
cpt <- compileCPT(c(list(x.0), rep.pat))
hmm <- grain(cpt)
hmm
```

```r
if (interactive()) iplot(hmm)
```
Index

* datasets
  chest, 2
  grass, 25

* models
  cpt-update, 6
  cptable, 7
  finding, 11
  grain-main, 13
  grain-simulate, 16
  grain_compile, 17
  grain_evidence, 18
  grain_predict, 22
  grain_propagate, 23
  querygrain, 29
  repeatPattern, 30

* utilities
  components_extract, 3
  components_gather, 5
  cpt-update, 6
  finding, 11
  grain_compile, 17
  grain_evidence, 18
  grain_propagate, 23
  load-save-hugin, 25
  logical, 27
  querygrain, 29
  absorbEvidence (grain_evidence), 18
  andtab (logical), 27
  andtable, 8
  andtable (logical), 27
  as.data.frame.grain_ev (evidence_object), 9
  booltab (logical), 27
  chest, 2
  chest_cpt (chest), 2
  compile, 24
  compile.cpt_grain (grain_compile), 17
  compile_grain, 15
  compile.grain (grain_compile), 17
  compile.pot_grain (grain_compile), 17
  compileCPT, 4, 8, 15, 31
  compileCPT (components_gather), 5
  compilePOT, 4, 8, 15
  compilePOT (components_gather), 5
  components_extract, 3
  components_gather, 5
  cpt-update, 6
  cptable, 7, 15, 28
  evidence (grain_evidence), 18
  evidence<- (grain_evidence), 18
  evidence_object, 9
  extractCPT, 6, 8, 15
  extractCPT (components_extract), 3
  extractMARG, 6
  extractMARG (components_extract), 3
  extractPOT, 6, 8, 15
  extractPOT (components_extract), 3
  finding, 11
  generics, 12
  getEvidence, 12, 30
  getEvidence (grain_evidence), 18
  getFinding, 15, 19
  getFinding (finding), 11
  grain, 4, 6, 8, 17, 23, 24, 26, 31
  grain (grain-main), 13
  grain-main, 13
  grain-simulate, 16
  grain.cpt_spec (grain-main), 13
  grain.CPTspec (grain-main), 13
  grain.dModel (grain-main), 13
  grain.graphNEL (grain-main), 13
  grain.pot_spec (grain-main), 13
  grain_compile, 17
  grain_evidence, 18