Package ‘gRain’

November 17, 2022

Version 1.3.12
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, broom, magrittr, Rcpp (>= 0.11.1)
URL https://people.math.aau.dk/~sorenh/software/gR/
Encoding UTF-8
VignetteBuilder knitr
LinkingTo Rcpp (>= 0.11.1), RcppArmadillo, RcppEigen, gRbase
ByteCompile Yes
RoxygenNote 7.2.2
NeedsCompilation yes
Repository CRAN
Date/Publication 2022-11-17 18:40:08 UTC

R topics documented:

chest ................................................................. 2
components_extract ............................................ 3
components_gather ............................................. 5
cpt ................................................................. 6
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

```R
## '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(1,0,1,0,1,0,1), 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)

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_representation

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)
```
components_gather

Compile conditional probability tables / cliques potentials.

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

Usage
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 ctptable's into an object from which a network can be built. For example, when specification of a cpt is made with ctable 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

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

---

cpt

Create conditional probability table CPT.

Description
Create conditional probability table CPT.

Usage
cpt(names, levels, values, smooth = 0)

Arguments
names Names of variables defining table; either a character vector or a right hand sided formula.
levels 1. a list with specification of the levels of the factors in names or 2) a vector with number of levels of the factors in names. See 'examples' below.
values values to go into the array.
smooth Should values be smoothed, see 'Details' below.

Value
An array.
Examples

```r
universe <- list(gender=c('male', 'female'),
                answer=c('yes', 'no'),
                rain=c('yes', 'no'))
t1 <- cpt(c("gender", "answer"), levels=universe, values=1:4)
t1
t2 <- cpt(~gender:answer, levels=universe, values=1:4)
t2
t3 <- cpt(~gender:answer, c(2, 2), values=1:4)
t3
```

cptable

Create conditional probability tables (CPTs)

Description

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

Usage

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

Arguments

- **vpar**: Specifications of the names in \( P(v|pa_1,...,pa_k) \). 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 \( \text{normalize}=\text{TRUE} \) then the probabilities are normalized to sum to one for each configuration of the parents.

If \( \text{smooth} \) is non-zero then zero entries of \( \text{values} \) are replaced with \( \text{smooth} \) before normalization takes place.

Regarding the form of the argument \( \text{vpar} \): To specify \( P(a|b,c) \) one may write \( \sim a|b:c, \sim a:b:c, \sim a|b+c, \sim a+b+c \) 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 \( a_1,a_2 \) and likewise for \( b \) and \( c \) then the order of \( \text{values} \) corresponds to the configurations \( (a_1,b_1,c_1), (a_2,b_1,c_1) (a_1,b_2,c_1), (a_2,b_2,c_1) \) etc. That is, the first variable varies fastest. Hence the first two elements in \( \text{values} \) will be the conditional probabilities of a given \( b=b_1, c=c_1 \).
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)

# 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, .2, .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["S", "R"]
```
```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, .2, .9, .1, 0, 1))
```

---

**Description**

Functions for defining and manipulating evidence.

**Usage**

```r
new_evi(evi_list = NULL, levels)

is.null_evi(object)

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

setdiff_evi(ev1, ev2)

union_evi(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

```r
## Define the universe
yn <- c("yes", "no")
uni <- list(asia = yn, tub = yn, smoke = yn, lung = yn, 
  bronc = yn, either = yn, xray = yn, dysp = yn)

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

e2 <- list(dysp="no", xray=c(0, 1))
eo2 <- new_evi(e2, levels=uni)
eo2 |> as.data.frame()

# Above e1 and e2 specifies the same evidence but information about 
# whether the state has been set definite or as a weight is 
# maintained.

e3 <- list(dysp="yes", asia="yes")
eo3 <- new_evi(e3, uni)
eo3 |> as.data.frame()

# If evidence 'e1' is already set in the network and new evidence 
# 'e3' emerges, then evidence in the network must be updated. But 
# there is a conflict in that dysp="yes" in 'e1' and 
# dysp="no" in 'e3'. The (arbitrary) convention is that 
# existing evidence overrides new evidence so that the only new 
# evidence in 'e3' is really asia="yes".

# To subtract existing evidence from new evidence we can do:
setdiff_evi(eo3, eo1) |> as.data.frame()

# Likewise the 'union' is
union_evi(eo3, eo1) |> as.data.frame()
```

---

**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

## setFindings

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

getFinding(bn1)
getFinding(bn2)

pFinding(bn1)
pFinding(bn2)

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

getFinding(bn1)
getFinding(bn2)

pFinding(bn1)
pFinding(bn2)

---

generics gRain generics

descriptions

Generic functions etc for the gRain package

Usage

nodeNames(object)

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

nodeStates(object, nodes = nodeNames(object))

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

universe(object, ...)

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

isCompiled(object)
isPropagated(object)
isCompiled(object) <- value
isPropagated(object) <- value

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

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

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

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

Arguments

<table>
<thead>
<tr>
<th>nodes</th>
<th>Some nodes of the object.</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>Additional arguments; currently not used.</td>
</tr>
<tr>
<td>value</td>
<td>Value to be set for slot in object.</td>
</tr>
<tr>
<td>x, object</td>
<td>A relevant object.</td>
</tr>
</tbody>
</table>

get_superset_list  

Get superset for each element in a list

Description

For each element (vector) x in x_set, find the first element (vector) y in y_set such that x is contained in y

Usage

get_superset_list(x_set, y_set, warn = FALSE)

Arguments

<table>
<thead>
<tr>
<th>x_set</th>
<th>Vector or list of vectors.</th>
</tr>
</thead>
<tbody>
<tr>
<td>y_set</td>
<td>Vector or list of vectors.</td>
</tr>
<tr>
<td>warn</td>
<td>Should a warning be made if an element is found.</td>
</tr>
</tbody>
</table>
Examples

```r
x_set <- list(c("a", "b"), "e", c("b", "a"))
y_set <- list(c("f","u", "e"), c("a", "b", "c", "a"), c("b", "c", "a"))
get_superset_list(x_set, y_set)
get_superset_list(letters[1:4], y_set)
get_superset_list(letters[1:4], letters[1:10])
x_set <- list(c("a", "b"), "e", c("b", "a"), "o")
y_set <- list(c("f","u", "e"), c("a", "b", "c", "a"), c("b", "c", "a"))
get_superset_list(x_set, y_set, warn=TRUE)
get_superset_list(x_set, y_set, warn=FALSE)
```

---

grain-main  

Graphical Independence Network

Description

Creating grain objects (graphical independence network).

Usage

```r
grain(x, ...)
```

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

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

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

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

## S3 method for class 'dModel'
```r
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.

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

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
## Create network from conditional probability tables CPTs:
yn <- c("yes", "no")
a <- cpt(~asia, values=c(1,99), levels=yn)
t.a <- cpt(~tub+asia, values=c(5,95,1,99), levels=yn)
s <- cpt(~smoke, values=c(5,5), levels=yn)
l.s <- cpt(~lung+smoke, values=c(1,9,1,99), levels=yn)
b.s <- cpt(~bronc+smoke, values=c(6,4,3,7), levels=yn)
e.lt <- cpt(~either+lung+tub, values=c(1,0,1,0,1,0,1), levels=yn)
x.e <- cpt(~xray+either, values=c(98,2,5,95), levels=yn)
d.be <- cpt(~dysp+bronc+either, values=c(9,1,7,3,8,2,1,9), levels=yn)
cpt_list <- list(a, t.a, s, l.s, b.s, e.lt, x.e, d.be)
chest_cpt <- compileCPT(cpt_list)
## Alternative: 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.
data(lizard, package="gRbase")
## From a DAG: height <- species -> diam
daG <- dag(~species + height:species + diam:species)
## From an undirected graph UG : [height:species][diam:species]
uG <- ug(~height:species + diam:species)
liz.ug <- grain(uG, data=lizard)
liz.dag <- grain(daG, data=lizard)
```

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.
grain_compile

Value

A data frame

Author(s)

Søren Højsgaard, <sorenh@math.aau.dk>

References


Examples

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.

Usage

## S3 method for class 'grain'
compile(
  object,
  propagate = FALSE,
  tug = NULL,
  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.
- **tug**: A triangulated undirected graph.
- **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,
  propagate = TRUE,
  details = 0
)
```
```
retractEvidence(object, nodes = NULL, propagate = TRUE)

absorbEvidence(object, propagate = TRUE)
pEvidence(object, evidence = NULL)
getEvidence(object, short = TRUE)
evidence(object, short = TRUE)

## S3 method for class 'grain'
evidence(object, short = TRUE)
evidence(object) <- value

## 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.
- **propagate**
  - Should the network be propagated?
- **details**
  - Debugging information
- **short**
  - If TRUE a dataframe with a summary is returned; otherwise a list with all details.
- **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.au.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"))
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)
chest.bn2
gEvidence(chest.bn2)

# 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(tabNew(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)

## 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")

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

- `object`: A grain object
- `response`: A vector of response variables to make predictions on
- `predictors`: A vector of predictor variables to make predictions from. Defaults to all variables that are not responses.
- `newdata`: A data frame
- `type`: If "class", the most probable class is returned; if "distribution" the conditional distribution is returned.
- `...`: Not used

## 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")
```

---

grain_propagate

Propagate a graphical independence network (a Bayesian network)

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$details, engine = "cpp", ...)
propagateLS(cq_pot_list, rip, initialize = TRUE, details = 0)
compute_p_evidence(object, details = object$details, engine = "cpp", ...)
```

Arguments

- `object` A grain object
- `details` For debugging info
- `engine` Either "R" or "cpp"; "cpp" is the default and the fastest.
- `...` Currently not used
- `cq_pot_list` List of clique potentials
- `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.


**grass**  

**Wet grass example**  

**Description**  
Conditional probability tables for the wet grass example.

**Usage**  
`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(.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)


# '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`.

Note

- In Hugin, it is possible to specify a potential of a node as a functional relation between other nodes. In a .net file, such a specification will appear as 'function' rather than as 'node'. Such a specification is not recognized by `loadHuginNet`.
- It is recommended to avoid the text node as part of the name of a node.
Description

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

Usage

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 \( a:b:c \) 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


See Also
cptable

Examples

## Logical OR:

## 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:
## Same story here:
```r
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
```

---

**mendel**

*Mendelian segregation*

**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,
  simplify = FALSE,
  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.
- `simplify`: Should the result be simplified (to a dataframe) if possible.
- `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

Create repeated patterns in Bayesian networks

Description

Repeated patterns is a useful model specification short cut for Bayesian networks

Usage

repeatPattern(plist, instances, unlist = TRUE, data = NULL)
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. See also the data argument.

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().

data A two column matrix. The first column is the index / name of a node; the second column is the index / name of the node’s parent.

Author(s)

Søren Højsgaard, <sorenh@math.aau.dk>

References


See Also

grain, compileCPT

Examples

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

x.0 <- cptable(~x0, values=c(1, 9), levels=yn) ## p(x0)
x.x <- cptable(~x[i]|x[i-1], values=c(1, 99, 2, 98), levels=yn) ## p(x[i]|x[i-1])

x.0 <- cpt(~x0, values=c(1, 9), levels=yn) ## p(x0)
x.x <- cpt(~x[i]|x[i-1], values=c(1, 99, 2, 98), levels=yn) ## p(x[i]|x[i-1])

pat <- list(x.x) ## Pattern to be repeated

n <- 5
rep.pat <- repeatPattern(pat, instances=1:n)
mc <- compileCPT(c(list(x.0), rep.pat)) |> grain()
if (interactive()) iplot(mc)

## Example: Hidden markov model: The x[i]'s are unobserved, the
## y[i]'s can be observed.

x.0 <- cptable(~x0, values=c(1, 9), levels=yn) ## p(x0)
x.x <- cptable(~x[i]|x[i-1], values=c(1, 99, 2, 98), levels=yn) ## p(x[i]|x[i-1])
y.x <- cptable(~y[i]|x[i], values=c(10, 90, 20, 80), levels=yn) ## p(y[i]|x[i])
x.0 <- cpt(~x0, values=c(1, 9), levels=yn) ## p(x0)
```
x.x <- cpt(~x[i]|x[i-1], values=c(1, 99, 2, 98), levels=yn)  ## p(x[i]|x[i-1])
y.x <- cpt(~y[i]|x[i], values=c(10, 90, 20, 80), levels=yn)  ## p(y[i]|x[i])

pat <- list(x.x, y.x)  ## Pattern to be repeated

rep.pat <- repeatPattern(pat, instances=1:n)
hmm <- compileCPT(c(list(x.0), rep.pat)) |> grain()
hmm

if (interactive()) iplot(hmm)

## Data-driven variable names
dep <- data.frame(i=c(1, 2, 3, 4, 5, 6, 7, 8),
                 p=c(0, 1, 2, 2, 3, 3, 4, 4))

z0 <- cptable(~z0, values=c(0.5, 0.5), levels=yn)
z.a <- cptable(~z[i] | z[data[i, "p"]], values=c(0.5, 0.5), levels=yn)
z.b <- repeatPattern(list(z.a), instances=1:nrow(dep), data=dep)
tree <- compileCPT(c(list(z0), z.b)) |> grain()
tree

if (interactive()) iplot(tree)

---

**replace-cpt**

**Replace CPTs in Bayesian network**

**Description**

Replace CPTs of Bayesian network.

**Usage**

```r
replaceCPT(object, value)
```

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

**Arguments**

- **object**
  - A grain object.

- **value**
  - A named list, see examples below.
Details

When a Bayesian network (BN) is constructed from a list of conditional probability tables (CPTs) (e.g. using the function `grain()`), various actions are taken:

1. It is checked that the list of CPTs define a directed acyclic graph (DAG).
2. The DAG is moralized and triangulated.
3. A list of clique potentials (one for each clique in the triangulated graph) is created from the list of CPTs.
4. The clique potentials are, by default, calibrated to each other so that the potentials contain marginal distributions.

The function described here bypass the first two steps which can provide an important gain in speed compared to constructing a new BN with a new set of CPTs with the same DAG.

Author(s)

Søren Højsgaard, <sorenh@math.aau.dk>

References


See Also

`grain`, `propagate`, `triangulate`, `rip`, `junctionTree`

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)

getgrain(wet.bn, "cpt")[c("R","S")]

# Update some CPTs
wet.bn <- replaceCPT(wet.bn, list(R=c(.3, .7), S=c(.1, .9, .7, .3)))
getgrain(wet.bn, "cpt")[c("R","S")]
```
simplify_query

Simplify output query to a Bayesian network

Description
Simplify output query to a Bayesian network to a dataframe provided that each node has the same levels.

Usage
simplify_query(b)

Arguments
b Result from running querygrain.
Index

* datasets
  chest, 2
  grass, 25

* models
  cptable, 7
  finding, 10
  grain-main, 14
  grain-simulate, 16
  grain_evidence, 18
  grain_predict, 22
  grain_propagate, 24
  querygrain, 30
  repeatPattern, 31
  replace-cpt, 33

* utilities
  components_extract, 3
  components_gather, 5
  cpt, 6
  finding, 10
  grain_compile, 17
  grain_evidence, 18
  grain_propagate, 24
  load-save-hugin, 26
  logical, 27
  querygrain, 30
  replace-cpt, 33

absorbEvidence (grain_evidence), 18
andtab (logical), 27
andtable, 8
andtable (logical), 27
as.data.frame.grain_evidence (evidence_object), 9

booltab (logical), 27

chest, 2
chest_cpt (chest), 2
compile, 25

compile.cpt_grain (grain_compile), 17
compile.grain, 15
compile.grain (grain Compile), 17
compile.pot_grain (grain_compile), 17
compileCPT, 4, 8, 15, 32
compileCPT (components_gather), 5
compilePOT, 4, 8, 15
compilePOT (components_gather), 5
components_extract, 3
components_gather, 5
compute_p_evidence (grain_propagate), 24
cpt, 6
cptable, 7, 15, 28
evidence (grain_evidence), 18
evidences<- (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, 10
generics, 12
get_superset_list, 13
getEvidence, 11, 31
getEvidence (grain_evidence), 18
getFinding, 15, 20
getFinding (finding), 10
grain, 4, 8, 18, 23, 25, 27, 32, 34
grain (grain-main), 14
grain-main, 14
grain-simulate, 16
grain.cpt_spec (grain-main), 14
grain.CPTspec (grain-main), 14
grain.dModel (grain-main), 14
grain.graphNEL (grain-main), 14

36
grain.pot_spec (grain-main), 14
grain_compile, 17
grain_evidence, 18
grain_joint_evidence, 20
grain_predict, 22
grain_propagate, 24
generated, 25
grass_cpt (grass), 25
is.null_evi (evidence_object), 9
isCompiled (generics), 12
isCompiled<- (generics), 12
isPropagated (generics), 12
isPropagated<- (generics), 12
junctionTree, 18, 34
load-save-hugin, 26
loadHuginNet (load-save-hugin), 26
marg2pot (components_extract), 3
mendel, 29
new_evi (evidence_object), 9
new_jev (grain_joint_evidence), 20
nodeNames (generics), 12
nodeStates (generics), 12
ortable (logical), 27
parray, 8
parse_cpt (components_gather), 5
parse_cpt, (components_gather), 5
parse_cpt.default (components_gather), 5
parse_cpt.xtabs, parse_cpt cptable_class, (components_gather), 5
pEvidence, 11, 31
pEvidence (grain-evidence), 18
pFinding, 15, 20
pFinding (finding), 10
pot2marg (components_extract), 3
predict.grain (grain_predict), 22
print.grain_evidence (evidence_object), 9
print.grain_joint_evidence (grain_joint_evidence), 20
propagate, 18, 34
propagate.grain, 15, 18
propagate.grain (grain_propagate), 24
propagateLS (grain_propagate), 24
propagateLS__ (grain_propagate), 24
qgrain (querygrain), 30
querygrain, 11, 30
repeatPattern, 31
replace-cpt, 33
replaceCPT (replace-cpt), 33
retractEvidence, 11, 31
retractEvidence (grain_evidence), 18
retractFinding, 15, 20
retractFinding (finding), 10
retractJEvidence
 (grain_joint_evidence), 20
rip, 18, 34
rip.grain (generics), 12
saveHuginNet (load-save-hugin), 26
setdiff_evi (evidence_object), 9
setEvidence, 11, 15, 31
setEvidence (grain_evidence), 18
setFinding, 15, 20
setFinding (finding), 10
setJEvidence (grain_joint_evidence), 20
simplify_query, 35
simulate.grain (grain_simulate), 16
subset.grain_evidence (evidence_object), 9
triangulate, 18, 34
union_evi (evidence_object), 9
universe (generics), 12
varNames.grain_evidence (evidence_object), 9
varNames.grainEvidence_ (generics), 12
vpar.cpt_grain (generics), 12
vpar.cpt_spec (generics), 12