Package 'CausalQueries'

June 28, 2022

Type Package

Title Make, Update, and Query Binary Causal Models

Version 0.1.0

Description
Users can declare binary causal models, update beliefs about causal types given data and calculate arbitrary estimands. Model definition makes use of 'dagitty' functionality. Updating is implemented in 'stan'. The approach used in 'CausalQueries' is a generalization of the 'biqq' models described in ``Mixing Methods: A Bayesian Approach'' (Humphreys and Jacobs, 2015, <DOI:10.1017/S0003055415000453>). The conceptual extension makes use of work on probabilistic causal models described in Pearl's Causality (Pearl, 2009, <DOI:10.1017/CBO9780511803161>).

License MIT + file LICENSE

Encoding UTF-8

LazyData true

RoxygenNote 7.2.0

Depends dplyr, methods, R (>= 3.4.0), Rcpp (>= 0.12.0)

Imports dagitty (>= 0.3-1), dirmult (>= 0.1.3-4), stats (>= 4.1.1), rlang (>= 0.2.0), rstan (>= 2.18.1), rstantools (>= 2.0.0), stringr (>= 1.4.0), ggdag (>= 0.2.4), latex2exp (>= 0.9.4), repr (>= 1.1.4), ggplot2 (>= 3.3.5), lifecycle (>= 1.0.1)

LinkingTo BH (>= 1.66.0), Rcpp (>= 0.12.0), RcppEigen (>= 0.3.3.3.0), rstan (>= 2.18.1), StanHeaders (>= 2.18.0)

Suggests testthat, rmarkdown, knitr, DeclareDesign, covr

SystemRequirements GNU make

Biarch true

NeedsCompilation yes

Author Clara Bicalho [ctb],
Jasper Cooper [ctb],
Macartan Humphreys [aut],
Till Tietz [aut, cre],
Alan Jacobs [aut],

Merlin Heidemanns [ctb],
Lily Medina [aut],
Julio Solis [ctb],
Georgiy Syunyaev [ctb]

Maintainer Till Tietz <ttietz2014@gmail.com>

Repository CRAN

Date/Publication 2022-06-27 22:20:13 UTC

R topics documented:

`CausalQueries-package` *'CausalQueries'*

Description

'CausalQueries' is a package that lets you generate binary causal models, update over models given data and calculate arbitrary causal queries. Model definition makes use of dagitty syntax. Updating is implemented in 'stan'.

`all_data_types` *All data types*

Description

Creates dataframe with all data types (including NA types) that are possible from a model.

Usage

```
all_data_types(
  model,
  complete_data = FALSE,
  possible_data = FALSE,
  given = NULL
)
```

Arguments

model	A causal_model. A model object generated by make_model.
complete_data	Logical. If 'TRUE' returns only complete data types (no NAs). Defaults to 'FALSE'.
possible_data	Logical. If 'TRUE' returns only complete data types (no NAs) that are *possible* given model restrictions. Note that in principle an intervention could make observationally impossible data types arise. Defaults to 'FALSE'.
given	A character. A quoted statement that evaluates to logical. Data conditional on specific values.

Value

A data.frame with all data types (including NA types) that are possible from a model.

Examples

```
all_data_types(make_model('X -> Y'))
model <- make_model('X -> Y') %>% set_restrictions(labels = list(Y = '00'), keep = TRUE)
  all_data_types(model)
  all_data_types(model, complete_data = TRUE)
  all_data_types(model, possible_data = TRUE)
  all_data_types(model, given  = 'X==1')
  all_data_types(model, given  = 'X==1 & Y==1')
```

---

collapse_data *Make compact data with data strategies*

Description

Take a 'data.frame' and return compact 'data.frame' of event types and strategies.

Usage

```
collapse_data(
  data,
  model,
  drop_NA = TRUE,
  drop_family = FALSE,
  summary = FALSE
)
```

Arguments

data	A data.frame. Data of nodes that can take three values: 0, 1, and NA. In long form as generated by make_events
model	A causal_model. A model object generated by make_model.
drop_NA	Logical. Whether to exclude strategy families that contain no observed data. Exceptionally if no data is provided, minimal data on data on first node is returned. Defaults to 'TRUE'
drop_family	Logical. Whether to remove column strategy from the output. Defaults to 'FALSE'.
summary	Logical. Whether to return summary of the data. See details. Defaults to 'FALSE'.

Value

A vector of data events

If summary = TRUE 'collapse_data' returns a list containing the following components:

data_events	A compact data.frame of event types and strategies.
observed_events	A vector of character strings specifying the events observed in the data
unobserved_events	A vector of character strings specifying the events not observed in the data

Examples

```
model <- make_model('X -> Y')

df <- data.frame(X = c(0,1,NA), Y = c(0,0,1))

df %>% collapse_data(model)


collapse_data(df, model, drop_NA = FALSE)

collapse_data(df, model, drop_family = TRUE)

collapse_data(df, model, summary = TRUE)

data <- make_data(model, n = 0)
collapse_data(data, model)

model <- make_model('X -> Y') %>% set_restrictions('X[]==1')
df <- simulate_data(model, n = 10)
df[1,1] <- ''
collapse_data(df, model)
data <- data.frame(X= 0:1)
collapse_data(data, model)
```

`complements` *Make statement for complements*

Description

Generate a statement for X1, X1 complement each other in the production of Y

Usage

```
complements(X1, X2, Y)
```

Arguments

`X1`	A character. The quoted name of the input node 1.
`X2`	A character. The quoted name of the input node 2.
`Y`	A character. The quoted name of the outcome node.

Value

A character statement of class `statement`

See Also

Other statements: `decreasing()`, `increasing()`, `interacts()`, `non_decreasing()`, `non_increasing()`, `substitutes()`, `te()`

Examples

```
complements('A', 'B', 'W')
```

data_type_names *Data type names*

Description

Provides names to data types

Usage

```
data_type_names(model, data)
```

Arguments

model	A causal_model. A model object generated by make_model.
data	A data.frame. Data of nodes that can take three values: 0, 1, and NA. In long form as generated by make_events

Value

A vector of strings of data types

Examples

```
model <- make_model('X -> Y')
data <- simulate_data(model, n = 2)
data_type_names(model, data)
```

decreasing *Make monotonicity statement (negative)*

Description

Generate a statement for Y monotonic (decreasing) in X

Usage

```
decreasing(X, Y)
```

Arguments

X	A character. The quoted name of the input node
Y	A character. The quoted name of the outcome node

Value

A character statement of class statement

See Also

Other statements: `complements()`, `increasing()`, `interacts()`, `non_decreasing()`, `non_increasing()`, `substitutes()`, `te()`

Examples

```
decreasing('A', 'B')
```

`democracy_data` *Democracy Data*

Description

A dataset containing information on inequality, democracy, mobilization, and international pressure. Made by `devtools::use_data(democracy_data, CausalQueries)`

Usage

```
democracy_data
```

Format

A data frame with 84 rows and 5 nodes:

Case Case

D Democracy

I Inequality

P International Pressure

M Mobilization

Source

https://www.cambridge.org/core/journals/american-political-science-review/article/inequality-and-regime-change-democratic-transitions-and-the-stability-of-democratic-rule/C39AAF4CF274445555FF41F7CC896AE3#fndtn-supplementary-materials/

draw_causal_type *Draw a single causal type given a parameter vector*

Description

Output is a parameter dataframe recording both parameters (case level priors) and the case level causal type.

Usage

```
draw_causal_type(model, ...)
```

Arguments

model	A causal_model. A model object generated by make_model.
...	Arguments passed to 'set_parameters'

Examples

```
# Simple draw using model's parameter vector
make_model("X -> M -> Y") %>%
draw_causal_type(.)

# Draw parameters from priors and draw type from parameters
make_model("X -> M -> Y") %>%
draw_causal_type(., param_type = "prior_draw")

# Draw type given specified parameters
make_model("X -> M -> Y") %>%
draw_causal_type(., parameters = 1:10)

# Define a causal type and reveal data
model <- make_model("X -> Y; X <-> Y")
type <- model %>% draw_causal_type()
make_data(model, parameters = type$causal_type)
```

expand_data *Expand compact data object to data frame*

Description

Expand compact data object to data frame

Usage

```
expand_data(data_events = NULL, model)
```

Arguments

data_events	A data.frame. It must be compatible with nodes in model. The default columns are event, strategy and count.
model	A causal_model. A model object generated by make_model.

Value

A data.frame with rows as data observation

Examples

```
model <- make_model('X->M->Y')
make_events(model, n = 5) %>%
  expand_data(model)
make_events(model, n = 0) %>%
  expand_data(model)
```

---

expand_wildcard *Expand wildcard*

Description

Expand statement containing wildcard

Usage

```
expand_wildcard(to_expand, join_by = "|", verbose = TRUE)
```

Arguments

to_expand	A character vector of length 1L.
join_by	A logical operator. Used to connect causal statements: *AND* ('&') or *OR* ('|'). Defaults to '|'.
verbose	Logical. Whether to print expanded query on the console.

Value

A character string with the expanded expression. Wildcard '.' is replaced by 0 and 1.

Examples

```
# Position of parentheses matters for type of expansion
# In the "global expansion" versions of the entire statement are joined
expand_wildcard('(Y[X=1, M=.] > Y[X=1, M=.])')
# In the "local expansion" versions of indicated parts are joined
expand_wildcard('(Y[X=1, M=.]) > (Y[X=1, M=.])')

# If parentheses are missing global expansion used.
expand_wildcard('Y[X=1, M=.] > Y[X=1, M=.]')

# Expressions not requiring expansion are allowed
expand_wildcard('(Y[X=1])')
```

`get_ambiguities_matrix` *Get ambiguities matrix*

Description

Return ambiguities matrix if it exists; otherwise calculate it assuming no confounding. The ambiguities matrix maps from causal types into data types.

Usage

```
get_ambiguities_matrix(model)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.

Value

A `data.frame`. Causal types (rows) corresponding to possible data realizations (columns).

Examples

```
model <- make_model('X -> Y')
get_ambiguities_matrix(model = model)
```

get_causal_types *Get causal types*

Description

Return data frame with types produced from all combinations of possible data produced by a DAG.

Usage

```
get_causal_types(model)
```

Arguments

model	A causal_model. A model object generated by make_model.

Value

A data.frame indicating causal types of a model

Examples

```
get_causal_types(make_model('X -> Y'))
```

get_event_prob *Draw event probabilities*

Description

Draw event probabilities

Usage

```
get_event_prob(model, parameters = NULL, A = NULL, P = NULL, given = NULL)
```

Arguments

model	A causal_model. A model object generated by make_model.
parameters	A vector of real numbers in [0,1]. Values of parameters to specify (optional). By default, parameters is drawn from model$parameters_df.
A	A data.frame. Ambiguity matrix. Not required but may be provided to avoid repeated computation for simulations.
P	A data.frame. Parameter matrix. Not required but may be provided to avoid repeated computation for simulations.
given	A string specifying known values on nodes, e.g. "X==1 & Y==1"

Value

An array of event probabilities

Examples

```
model <- make_model('X -> Y')
get_event_prob(model = model)
get_event_prob(model = model, parameters = rep(1, 6))
get_event_prob(model = model, parameters = 1:6)
```

`get_nodal_types` *Get list of types for nodes in a DAG*

Description

As type labels are hard to interpret for large models, the type list includes an attribute to help interpret them. See `attr(types, interpret)`

Usage

```
get_nodal_types(model, collapse = TRUE)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.
`collapse`	Logical. If 'TRUE', shows unique nodal types for each node. If 'FALSE', shows for each node a matrix with nodal types as rows and parent types as columns, if applicable. Defaults to 'TRUE'.

Value

A named `list` of nodal types for each parent in a DAG

Examples

```
model <- make_model('X -> K -> Y')
get_nodal_types(model)

model <- make_model('X -> K -> Y') %>%
   set_restrictions(statement = 'K[X=1]>K[X=0]') %>%
   set_confound(list(K = 'Y[K=1]>Y[K=0]'))
get_nodal_types(model)
```

get_parameter_matrix *Get parameter matrix*

Description

Return parameter matrix if it exists; otherwise calculate it assuming no confounding. The parameter matrix maps from parameters into causal types. In models without confounding parameters correspond to nodal types.

Usage

```
get_parameter_matrix(model)
```

Arguments

model	A model created by make_model()

Value

A data.frame, the parameter matrix, mapping from parameters to causal types

Examples

```
model <- make_model('X -> Y')
get_parameter_matrix(model)
```

get_parameter_names *Get parameter names*

Description

Parameter names taken from P matrix or model if no P matrix provided

Usage

```
get_parameter_names(model, include_paramset = TRUE)
```

Arguments

model	A causal_model. A model object generated by make_model.
include_paramset	Logical. Whether to include the param set prefix as part of the name.

Value

A character vector with the names of the parameters in the model

Examples

```
get_parameter_names(make_model('X->Y'))
```

`get_param_dist` *Get a distribution of model parameters*

Description

Using parameters, priors, or posteriors

Usage

```
get_param_dist(model, using, n_draws = 4000)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.
`using`	A character string. It indicates whether to use 'priors', 'posteriors' or 'parameters'.
`n_draws`	An integer. If no prior distribution is provided, generate prior distribution with `n_draws` number of draws.

Value

A `matrix` with the distribution of the parameters in the model

Examples

```
get_param_dist(model = make_model('X->Y'), using = 'priors', n_draws = 4)
get_param_dist(model = make_model('X->Y'), using = 'parameters')
```

`get_parents` *Get list of parents of all nodes in a model*

Description

Get list of parents of all nodes in a model

Usage

```
get_parents(model)
```

Arguments

model	A causal_model. A model object generated by make_model.

Value

A list of parents in a DAG

Examples

```
model <- make_model('X -> K -> Y')
get_parents(model)
```

---

get_parmap *Get parmap: a matrix mapping from parameters to data types*

Description

Gets parmap from a model, or generates if not available.

Usage

```
get_parmap(model, A = NULL, P = NULL)
```

Arguments

model	A causal_model. A model object generated by make_model.
A	A data.frame. Ambiguity matrix. Not required but may be provided to avoid repeated computation for simulations.
P	A data.frame. Parameter matrix. Not required but may be provided to avoid repeated computation for simulations.

Value

A matrix

Examples

```
get_parmap(model = make_model('X->Y'))
```

get_prior_distribution *Get a prior distribution from priors*

Description

Add to the model a ‘n_draws x n_param‘ matrix of possible parameters.

Usage

```
get_prior_distribution(model, n_draws = 4000)
```

Arguments

model	A causal_model. A model object generated by make_model.
n_draws	A scalar. Number of draws.

Value

A ‘data.frame‘ with dimension ‘n_param‘x ‘n_draws‘ of possible lambda draws

See Also

Other prior_distribution: make_prior_distribution(), set_prior_distribution()

Examples

```
make_model('X -> Y') %>% set_prior_distribution(n_draws = 5) %>% get_prior_distribution()
make_model('X -> Y') %>% get_prior_distribution(3)
```

get_query_types *Look up query types*

Description

Find which nodal or causal types are satisfied by a query.

Usage

```
get_query_types(model, query, map = "causal_type", join_by = "|")
```

Arguments

model	A causal_model. A model object generated by make_model.
query	A character string. An expression defining nodal types to interrogate realise_outcomes
map	Types in query. Either nodal_type or causal_type. Default is causal_type.
join_by	A logical operator. Used to connect causal statements: *AND* ('&') or *OR* ('|'). Defaults to '|'.

Value

A list containing some of the following elements

types	A named vector with logical values indicating whether a nodal_type or a causal_type satisfy 'query'
query	A character string as specified by the user
expanded_query	A character string with the expanded query. Only differs from 'query' if this contains wildcard '.'
evaluated_nodes	Value that the nodes take given a query
node	A character string of the node whose nodal types are being queried
type_list	List of causal types satisfied by a query

Examples

```
model <- make_model('X -> M -> Y; X->Y')
query <- '(Y[X=0] > Y[X=1])'

get_query_types(model, query, map="nodal_type")
get_query_types(model, query, map="causal_type")
get_query_types(model, query)

# Examples with map = "nodal_type"

query <- '(Y[X=0, M = .] > Y[X=1, M = 0])'
get_query_types(model, query, map="nodal_type")

query <- '(Y[] == 1)'
get_query_types(model, query, map="nodal_type")
get_query_types(model, query, map="nodal_type", join_by = '&')

# Root nodes specified with []
get_query_types(model, '(X[] == 1)', map="nodal_type")

query <- '(M[X=1] == M[X=0])'
get_query_types(model, query, map="nodal_type")

# Helpers
model <- make_model('M->Y; X->Y')
query <- complements('X', 'M', 'Y')
get_query_types(model, query, map="nodal_type")
```

```
# Examples with map = "causal_type"

model <- make_model('X -> M -> Y; X->Y')
query <- 'Y[M=M[X=0], X=1]==1'
get_query_types(model, query, map= "causal_type")

query <- '(Y[X=1, M = 1] >  Y[X=0, M = 1]) & (Y[X=1, M = 0] >  Y[X=0, M = 0])'
get_query_types(model, query, "causal_type")

query <- 'Y[X=1] == Y[X=0]'
get_query_types(model, query, "causal_type")

query <- '(X == 1) & (M==1) & (Y ==1) & (Y[X=0] ==1)'
get_query_types(model, query, "causal_type")

query <- '(Y[X = .]==1)'
get_query_types(model, query, "causal_type")
```

`get_type_prob` *Get type probabilities*

Description

Gets probability of vector of causal types given a single realization of parameters, possibly drawn from model priors.

Usage

```
get_type_prob(model, P = NULL, parameters = NULL)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.
`P`	A `data.frame`. Parameter matrix. Not required but may be provided to avoid repeated computation for simulations.
`parameters`	A vector of real numbers in [0,1]. Values of parameters to specify (optional). By default, parameters is drawn from `model$parameters_df`.

Details

By default, parameters is drawn from 'using' argument (either from priors, posteriors, or from model$parameters)

Value

A vector with probabilities of vector of causal types

Examples

```
get_type_prob(model = make_model('X->Y'))
get_type_prob(model = make_model('X->Y'), parameters = 1:6)
```

`get_type_prob_multiple` *Draw matrix of type probabilities, before or after estimation*

Description

Draw matrix of type probabilities, before or after estimation

Usage

```
get_type_prob_multiple(
  model,
  using = "priors",
  parameters = NULL,
  n_draws = 4000,
  param_dist = NULL,
  P = NULL
)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.
`using`	A character. It indicates whether to use 'priors', 'posteriors' or 'parameters'.
`parameters`	A vector of real numbers in [0,1]. Values of parameters to specify (optional). By default, parameters is drawn from `model$parameters_df`.
`n_draws`	An integer. If no prior distribution is provided, generate prior distribution with `n_draws` number of draws.
`param_dist`	A `matrix`. Distribution of parameters. Optional for speed.
`P`	A `data.frame`. Parameter matrix. Not required but may be provided to avoid repeated computation for simulations.

Value

A `matrix` of type probabilities.

Examples

```
model <- make_model('X -> Y')
get_type_prob_multiple(model, using = 'priors', n_draws = 3)
get_type_prob_multiple(model, using = 'parameters', n_draws = 3)
```

increasing *Make monotonicity statement (positive)*

Description

Generate a statement for Y monotonic (increasing) in X

Usage

```
increasing(X, Y)
```

Arguments

X	A character. The quoted name of the input node
Y	A character. The quoted name of the outcome node

Value

A character statement of class `statement`

See Also

Other statements: `complements()`, `decreasing()`, `interacts()`, `non_decreasing()`, `non_increasing()`, `substitutes()`, `te()`

Examples

```
increasing('A', 'B')
```

interacts *Make statement for any interaction*

Description

Generate a statement for X1, X1 interact in the production of Y

Usage

```
interacts(X1, X2, Y)
```

Arguments

X1	A character. The quoted name of the input node 1.
X2	A character. The quoted name of the input node 2.
Y	A character. The quoted name of the outcome node.

Value

A character statement of class `statement`

See Also

Other statements: `complements()`, `decreasing()`, `increasing()`, `non_decreasing()`, `non_increasing()`, `substitutes()`, `te()`

Examples

```
interacts('A', 'B', 'W')
get_query_types(model = make_model('X-> Y <- W'),
         query = interacts('X', 'W', 'Y'), map = "causal_type")
```

---

interpret_type *Interpret or find position in nodal type*

Description

Interprets the position of one or more digits (specified by `position`) in a nodal type. Alternatively returns nodal type digit positions that correspond to one or more given `condition`.

Usage

```
interpret_type(model, condition = NULL, position = NULL)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.
`condition`	A vector of characters. Strings specifying the child node, followed by '|' (given) and the values of its parent nodes in `model`.
`position`	A named list of integers. The name is the name of the child node in `model`, and its value a vector of digit positions in that node's nodal type to be interpreted. See 'Details'.

Details

A node for a child node X with k parents has a nodal type represented by X followed by `2^k` digits. Argument `position` allows user to interpret the meaning of one or more digit positions in any nodal type. For example `position = list(X = 1:3)` will return the interpretation of the first three digits in causal types for X. Argument `condition` allows users to query the digit position in the nodal type by providing instead the values of the parent nodes of a given child. For example, `condition = 'X | Z=0 & R=1'` returns the digit position that corresponds to values X takes when Z = 0 and R = 1.

Value

A named `list` with interpretation of positions of the digits in a nodal type

Examples

```
model <- make_model('R -> X; Z -> X; X -> Y')
#Example using digit position
interpret_type(model, position = list(X = c(3,4), Y = 1))
#Example using condition
interpret_type(model, condition = c('X | Z=0 & R=1', 'X | Z=0 & R=0'))
#Return interpretation of all digit positions of all nodes
interpret_type(model)
```

`make_data` *Make data*

Description

Make data

Usage

```
make_data(
  model,
  n = NULL,
  parameters = NULL,
  param_type = NULL,
  nodes = NULL,
  n_steps = NULL,
  probs = NULL,
  subsets = TRUE,
  complete_data = NULL,
  given = NULL,
  verbose = TRUE,
  ...
)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.
`n`	Non negative integer. Number of observations. If not provided it is inferred from the largest n_step.
`parameters`	A vector of real numbers in [0,1]. Values of parameters to specify (optional). By default, parameters is drawn from `model$parameters_df`.

param_type	A character. String specifying type of parameters to make ("flat", "prior_mean", "posterior_mean", "prior_draw", "posterior_draw", "define"). With param_type set to `define` use arguments to be passed to `make_priors`; otherwise `flat` sets equal probabilities on each nodal type in each parameter set; `prior_mean`, `prior_draw`, `posterior_mean`, `posterior_draw` take parameters as the means or as draws from the prior or posterior.
nodes	A `list`. Which nodes to be observed at each step. If NULL all nodes are observed.
n_steps	A `list`. Number of observations to be observed at each step
probs	A `list`. Observation probabilities at each step
subsets	A `list`. Strata within which observations are to be observed at each step. TRUE for all, otherwise an expression that evaluates to a logical condition.
complete_data	A `data.frame`. Dataset with complete observations. Optional.
given	A string specifying known values on nodes, e.g. "X==1 & Y==1"
verbose	Logical. If TRUE prints step schedule.
...	additional arguments that can be passed to `link{make_parameters}`

Details

Note that default behavior is not to take account of whether a node has already been observed when determining whether to select or not. One can however specifically request observation of nodes that have not been previously observed.

Value

A `data.frame` with simulated data.

Examples

```
# Simple draws
model <- make_model("X -> M -> Y")
make_data(model)
make_data(model, n = 3, nodes = c("X","Y"))
make_data(model, n = 3, param_type = "prior_draw")
make_data(model, n = 10, param_type = "define", parameters =  0:9)

# Data Strategies
# A strategy in which X, Y are observed for sure and M is observed
# with 50% probability for X=1, Y=0 cases

model <- make_model("X -> M -> Y")
make_data(
  model,
  n = 8,
  nodes = list(c("X", "Y"), "M"),
  probs = list(1, .5),
  subsets = list(TRUE, "X==1 & Y==0"))
```

```
# n not provided but inferred from largest n_step (not from sum of n_steps)
make_data(
  model,
  nodes = list(c("X", "Y"), "M"),
  n_steps = list(5, 2))

# Wide then deep
  make_data(
  model,
  n = 8,
  nodes = list(c("X", "Y"), "M"),
  subsets = list(TRUE, "!is.na(X) & !is.na(Y)"),
  n_steps = list(6, 2))


make_data(
  model,
  n = 8,
  nodes = list(c("X", "Y"), c("X", "M")),
  subsets = list(TRUE, "is.na(X)"),
  n_steps = list(3, 2))

# Example with probabilities at each step

make_data(
  model,
  n = 8,
  nodes = list(c("X", "Y"), c("X", "M")),
  subsets = list(TRUE, "is.na(X)"),
  probs = list(.5, .2))

# Example with given data
make_data(model, given = "X==1 & Y==1", n = 5)
```

`make_events` *Make data in compact form*

Description

Draw n events given event probabilities. Draws full data only. For incomplete data see `make_data`.

Usage

```
make_events(
  model,
  n = 1,
  w = NULL,
  P = NULL,
  A = NULL,
```

```
  parameters = NULL,
  param_type = NULL,
  include_strategy = FALSE,
  ...
)
```

Arguments

model	A causal_model. A model object generated by make_model.
n	An integer. Number of observations.
w	A numeric matrix. A 'n_parameters x 1' matrix of event probabilities with named rows.
P	A data.frame. Parameter matrix. Not required but may be provided to avoid repeated computation for simulations.
A	A data.frame. Ambiguity matrix. Not required but may be provided to avoid repeated computation for simulations.
parameters	A vector of real numbers in [0,1]. Values of parameters to specify (optional). By default, parameters is drawn from model$parameters_df.
param_type	A character. String specifying type of parameters to make ('flat', 'prior_mean', 'posterior_mean', 'prior_draw', 'posterior_draw', 'define). With param_type set to define use arguments to be passed to make_priors; otherwise flat sets equal probabilities on each nodal type in each parameter set; prior_mean, prior_draw, posterior_mean, posterior_draw take parameters as the means or as draws from the prior or posterior.
include_strategy	Logical. Whether to include a 'strategy' vector. Defaults to FALSE. Strategy vector does not vary with full data but expected by some functions.
...	Arguments to be passed to make_priors if param_type == define

Value

A data.frame of events

Examples

```
model <- make_model('X -> Y')
make_events(model = model)
make_events(model = model, param_type = 'prior_draw')
make_events(model = model, include_strategy = TRUE)
```

make_model *Make a model*

Description

make_model uses dagitty syntax and functionality to specify nodes and edges of a graph. Implied causal types are calculated and default priors are provided under the assumption of no confounding. Models can be updated with specification of a parameter matrix, P, by providing restrictions on causal types, and/or by providing informative priors on parameters. The default setting for a causal model have flat (uniform) priors and parameters putting equal weight on each parameter within each parameter set. These can be adjust with set_priors and set_parameters

Usage

```
make_model(statement, add_causal_types = TRUE)
```

Arguments

statement	A character. Statement describing causal relations using dagitty syntax. Only directed relations are permitted. For instance "X -> Y" or "X1 -> Y <- X2; X1 -> X2".
add_causal_types	Logical. Whether to create and attach causal types to model. Defaults to 'TRUE'.

Value

An object of class causal_model.

An object of class "causal_model" is a list containing at least the following components:

dag	A data.frame with columns 'parent'and 'children' indicating how nodes relate to each other.
node	A named list with the nodes in the model
statement	A character vector of the statement that defines the model
nodal_types	A named list with the nodal types in the model
parameters_df	A data.frame with descriptive information of the parameters in the model

Examples

```
make_model(statement = "X -> Y")
modelXKY <- make_model("X -> K -> Y; X -> Y")

# Example where cyclicaly dag attempted
## Not run:
 modelXKX <- make_model("X -> K -> X")

## End(Not run)
```

```
# Examples with confounding
model <- make_model("X->Y; X <-> Y")
model$P
model <- make_model("Y2 <- X -> Y1; X <-> Y1; X <-> Y2")
dim(model$P)
model$P
model <- make_model("X1 -> Y <- X2; X1 <-> Y; X2 <-> Y")
dim(model$P)
model$parameters_df

# A single node graph is also possible
model <- make_model("X")

# Unconnected nodes cannot
## Not run:
 model <- make_model("X <-> Y")

## End(Not run)
```

`make_parameter_matrix` *Make parameter matrix*

Description

Calculate parameter matrix assuming no confounding. The parameter matrix maps from parameters into causal types. In models without confounding parameters correspond to nodal types.

Usage

```
make_parameter_matrix(model)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.

Value

A `data.frame`, the parameter matrix, mapping from parameters to causal types

Examples

```
model <- make_model('X -> Y')
make_parameter_matrix(model)
```

make_parmap *Make parmap: a matrix mapping from parameters to data types*

Description

Generates a matrix with a row per parameter and a column per data type.

Usage

```
make_parmap(model, A = NULL, P = NULL)
```

Arguments

model	A causal_model. A model object generated by make_model.
A	A data.frame. Ambiguity matrix. Not required but may be provided to avoid repeated computation for simulations.
P	A data.frame. Parameter matrix. Not required but may be provided to avoid repeated computation for simulations.

Value

A matrix

Examples

```
make_parmap(model = make_model('X->Y'))
make_parmap(model = make_model('X->Y; X<->Y'))
make_parmap(model = make_model('X->Y; X<->Y')) %>% attr("map")
make_parmap(model = make_model('X -> M -> Y; X <-> Y'))
make_parmap(model = make_model('X -> M -> Y; M <-> Y'))
model <- make_model('X -> M -> Y; M <-> Y; X <-> M')
make_parmap(model)
make_parmap(model) %>% attr("map")
# Any ways (without paths splits)
make_parmap(model) %*% (make_parmap(model) %>% attr("map"))

## Not run:
# X1 and X2 are confounded and jointly determine Y1, Y2.
# For instance for models in which X and Y take on four values rather than 2.
model <- make_model("Y2 <- X1 -> Y1; Y2 <- X2 ->Y1; X1 <-> X2; Y1 <-> Y2")
data <- CausalQueries:::minimal_event_data(model)
check <- CausalQueries:::prep_stan_data(model, data, keep_transformed = TRUE)
check$n_params
a <- update_model(model)
make_parmap(model) %>% dim

## End(Not run)
```

make_prior_distribution *Make a prior distribution from priors*

Description

Create a ‘n_param‘x ‘n_draws‘ database of possible lambda draws to be attached to the model.

Usage

```
make_prior_distribution(model, n_draws = 4000)
```

Arguments

model	A causal_model. A model object generated by make_model.
n_draws	A scalar. Number of draws.

Value

A ‘data.frame‘ with dimension ‘n_param‘x ‘n_draws‘ of possible lambda draws

See Also

Other prior_distribution: get_prior_distribution(), set_prior_distribution()

Examples

```
make_model('X -> Y') %>% make_prior_distribution(n_draws = 5)
```

non_decreasing *Make monotonicity statement (non negative)*

Description

Generate a statement for Y weakly monotonic (increasing) in X

Usage

```
non_decreasing(X, Y)
```

Arguments

X	A character. The quoted name of the input node
Y	A character. The quoted name of the outcome node

Value

A character statement of class `statement`

See Also

Other statements: `complements()`, `decreasing()`, `increasing()`, `interacts()`, `non_increasing()`, `substitutes()`, `te()`

Examples

```
non_decreasing('A', 'B')
```

---

non_increasing *Make monotonicity statement (non positive)*

Description

Generate a statement for Y weakly monotonic (not increasing) in X

Usage

```
non_increasing(X, Y)
```

Arguments

`X`	A character. The quoted name of the input node
`Y`	A character. The quoted name of the outcome node

Value

A character statement of class `statement`

See Also

Other statements: `complements()`, `decreasing()`, `increasing()`, `interacts()`, `non_decreasing()`, `substitutes()`, `te()`

Examples

```
non_increasing('A', 'B')
```

observe_data *Observe data, given a strategy*

Description

Observe data, given a strategy

Usage

```
observe_data(
  complete_data,
  observed = NULL,
  nodes_to_observe = NULL,
  prob = 1,
  m = NULL,
  subset = TRUE
)
```

Arguments

complete_data	A data.frame. Data observed and unobserved.
observed	A data.frame. Data observed.
nodes_to_observe	A list. Nodes to observe.
prob	A scalar. Observation probability.
m	A integer. Number of units to observe; if specified, m overrides prob.
subset	A character. Logical statement that can be applied to rows of complete data. For instance observation for some nodes might depend on observed values of other nodes; or observation may only be sought if data not already observed!

Value

A data.frame with logical values indicating which nodes to observe in each row of 'complete_data'.

Examples

```
model <- make_model("X -> Y")
df <- simulate_data(model, n = 8)
# Observe X values only
observe_data(complete_data = df, nodes_to_observe = "X")
# Observe half the Y values for cases with observed X = 1
observe_data(complete_data = df,
     observed = observe_data(complete_data = df, nodes_to_observe = "X"),
     nodes_to_observe = "Y", prob = .5,
     subset = "X==1")
```

parameter_setting *Setting parameters*

Description

Functionality for altering parameters:

A vector of 'true' parameters; possibly drawn from prior or posterior.

Add a true parameter vector to a model. Parameters can be created using arguments passed to make_parameters and make_priors.

Extracts parameters as a named vector

Usage

```
make_parameters(
  model,
  parameters = NULL,
  param_type = NULL,
  warning = TRUE,
  normalize = TRUE,
  ...
)

set_parameters(
  model,
  parameters = NULL,
  param_type = NULL,
  warning = FALSE,
  ...
)

get_parameters(model, param_type = NULL)
```

Arguments

model	A causal_model. A model object generated by make_model.
parameters	A vector of real numbers in [0,1]. Values of parameters to specify (optional). By default, parameters is drawn from model$parameters_df.
param_type	A character. String specifying type of parameters to make ("flat", "prior_mean", "posterior_mean", "prior_draw", "posterior_draw", "define). With param_type set to define use arguments to be passed to make_priors; otherwise flat sets equal probabilities on each nodal type in each parameter set; prior_mean, prior_draw, posterior_mean, posterior_draw take parameters as the means or as draws from the prior or posterior.
warning	Logical. Whether to warn about parameter renormalization.

normalize	Logical. If parameter given for a subset of a family the residual elements are normalized so that parameters in param_set sum to 1 and provided params are unaltered.
...	Options passed onto make_priors.

Value

A vector of draws from the prior or distribution of parameters

An object of class causal_model. It essentially returns a list containing the elements comprising a model (e.g. 'statement', 'nodal_types' and 'DAG') with true vector of parameters attached to it.

A vector of draws from the prior or distribution of parameters

Examples

```
# make_parameters examples:

# Simple examples
model <- make_model('X -> Y')
data  <- simulate_data(model, n = 2)
model <- update_model(model, data)
make_parameters(model, parameters = c(.25, .75, 1.25,.25, .25, .25))
make_parameters(model, param_type = 'flat')
make_parameters(model, param_type = 'prior_draw')
make_parameters(model, param_type = 'prior_mean')
make_parameters(model, param_type = 'posterior_draw')
make_parameters(model, param_type = 'posterior_mean')




#altering values using \code{alter_at}
make_model("X -> Y") %>% make_parameters(parameters = c(0.5,0.25),
alter_at = "node == 'Y' & nodal_type %in% c('00','01')")

#altering values using \code{param_names}
make_model("X -> Y") %>% make_parameters(parameters = c(0.5,0.25),
param_names = c("Y.10","Y.01"))

#altering values using \code{statement}
make_model("X -> Y") %>% make_parameters(parameters = c(0.5),
statement = "Y[X=1] > Y[X=0]")

#altering values using a combination of other arguments
make_model("X -> Y") %>% make_parameters(parameters = c(0.5,0.25),
node = "Y", nodal_type = c("00","01"))

# Normalize renormalizes values not set so that value set is not renomalized
make_parameters(make_model('X -> Y'),
                statement = 'Y[X=1]>Y[X=0]', parameters = .5)
make_parameters(make_model('X -> Y'),
```

```
                statement = 'Y[X=1]>Y[X=0]', parameters = .5, normalize = FALSE)


# set_parameters examples:

make_model('X->Y') %>% set_parameters(1:6) %>% get_parameters()

# Simple examples
model <- make_model('X -> Y')
data  <- simulate_data(model, n = 2)
model <- update_model(model, data)
set_parameters(model, parameters = c(.25, .75, 1.25,.25, .25, .25))
set_parameters(model, param_type = 'flat')
set_parameters(model, param_type = 'prior_draw')
set_parameters(model, param_type = 'prior_mean')
set_parameters(model, param_type = 'posterior_draw')
set_parameters(model, param_type = 'posterior_mean')




#altering values using \code{alter_at}
make_model("X -> Y") %>% set_parameters(parameters = c(0.5,0.25),
alter_at = "node == 'Y' & nodal_type %in% c('00','01')")

#altering values using \code{param_names}
make_model("X -> Y") %>% set_parameters(parameters = c(0.5,0.25),
param_names = c("Y.10","Y.01"))

#altering values using \code{statement}
make_model("X -> Y") %>% set_parameters(parameters = c(0.5),
statement = "Y[X=1] > Y[X=0]")

#altering values using a combination of other arguments
make_model("X -> Y") %>% set_parameters(parameters = c(0.5,0.25),
node = "Y", nodal_type = c("00","01"))




# get_parameters examples:

get_parameters(make_model('X -> Y'))
```

`prior_setting` *Setting priors*

Description

Functionality for altering priors:

`make_priors` Generates priors for a model.

`set_priors` Adds priors to a model.

Extracts priors as a named vector

Usage

```
make_priors(
  model,
  alphas = NA,
  distribution = NA,
  alter_at = NA,
  node = NA,
  nodal_type = NA,
  label = NA,
  param_set = NA,
  given = NA,
  statement = NA,
  join_by = "|",
  param_names = NA
)

set_priors(
  model,
  alphas = NA,
  distribution = NA,
  alter_at = NA,
  node = NA,
  nodal_type = NA,
  label = NA,
  param_set = NA,
  given = NA,
  statement = NA,
  join_by = "|",
  param_names = NA
)

get_priors(model)
```

Arguments

`model`	A model object generated by make_model().
`alphas`	Real positive numbers giving hyperparameters of the Dirichlet distribution
`distribution`	string indicating a common prior distribution (uniform, jeffreys or certainty)
`alter_at`	string specifying filtering operations to be applied to parameters_df, yielding a logical vector indicating parameters for which values should be altered. (see examples)
`node`	string indicating nodes which are to be altered

nodal_type	string. Label for nodal type indicating nodal types for which values are to be altered
label	string. Label for nodal type indicating nodal types for which values are to be altered. Equivalent to nodal_type.
param_set	string indicating the name of the set of parameters to be altered
given	string indicates the node on which the parameter to be altered depends
statement	causal query that determines nodal types for which values are to be altered
join_by	string specifying the logical operator joining expanded types when statement contains wildcards. Can take values '&' (logical AND) or '|' (logical OR).
param_names	string. The name of specific parameter in the form of, for example, 'X.1', 'Y.01'

Details

Seven arguments govern which parameters should be altered. The default is 'all' but this can be reduced by specifying

* alter_at String specifying filtering operations to be applied to parameters_df, yielding a logical vector indicating parameters for which values should be altered. "node == 'X' & nodal_type

* node, which restricts for example to parameters associated with node 'X'

* label or nodal_type The label of a particular nodal type, written either in the form Y0000 or Y.Y0000

* param_set The param_set of a parameter.

* given Given parameter set of a parameter.

* statement, which restricts for example to nodal types that satisfy the statement 'Y[X=1] > Y[X=0]'

* param_set, given, which are useful when setting confound statements that produces several sets of parameters

Two arguments govern what values to apply:

* alphas is one or more non negative numbers and

* distribution indicates one of a common class: uniform, jeffreys, or 'certain'

Forbidden statements include:

- Setting distribution and values at the same time.
- Setting a distribution other than uniform, jeffreys or certainty.
- Setting negative values.
- specifying alter_at with any of node, nodal_type,param_set,given,statement or param_names
- specifying param_names with any of node,nodal_type,param_set,given,statement or alter_at
- specifying statement with any of node or nodal_type

Value

A vector indicating the hyperparameters of the prior distribution of the nodal types.

An object of class causal_model. It essentially returns a list containing the elements comprising a model (e.g. 'statement', 'nodal_types' and 'DAG') with the 'priors' attached to it.

A vector indicating the hyperparameters of the prior distribution of the nodal types.

Examples

```
# make_priors examples:

# Pass all nodal types
model <- make_model("Y <- X")
make_priors(model, alphas = .4)
make_priors(model, distribution = "jeffreys")

model <- CausalQueries::make_model("X -> M -> Y; X <-> Y")

#altering values using \code{alter_at}
make_priors(model = model, alphas = c(0.5,0.25),
alter_at = "node == 'Y' & nodal_type %in% c('00','01') & given == 'X.0'")

#altering values using \code{param_names}
make_priors(model = model, alphas = c(0.5,0.25),
param_names = c("Y.10_X.0","Y.10_X.1"))

#altering values using \code{statement}
make_priors(model = model, alphas = c(0.5,0.25),
statement = "Y[M=1] > Y[M=0]")

#altering values using a combination of other arguments
make_priors(model = model, alphas = c(0.5,0.25),
node = "Y", nodal_type = c("00","01"), given = "X.0")

# set_priors examples:

# Pass all nodal types
model <- make_model("Y <- X")
set_priors(model, alphas = .4)
set_priors(model, distribution = "jeffreys")

model <- CausalQueries::make_model("X -> M -> Y; X <-> Y")

#altering values using \code{alter_at}
set_priors(model = model, alphas = c(0.5,0.25),
alter_at = "node == 'Y' & nodal_type %in% c('00','01') & given == 'X.0'")

#altering values using \code{param_names}
set_priors(model = model, alphas = c(0.5,0.25),
param_names = c("Y.10_X.0","Y.10_X.1"))

#altering values using \code{statement}
set_priors(model = model, alphas = c(0.5,0.25),
statement = "Y[M=1] > Y[M=0]")

#altering values using a combination of other arguments
set_priors(model = model, alphas = c(0.5,0.25), node = "Y",
nodal_type = c("00","01"), given = "X.0")
```

```
# get_priors examples:

get_priors(make_model('X -> Y'))
```

`query_distribution` *Calculate query distribution*

Description

Calculated distribution of a query from a prior or posterior distribution of parameters

Usage

```
query_distribution(
  model,
  query,
  given = TRUE,
  using = "parameters",
  parameters = NULL,
  type_distribution = NULL,
  verbose = FALSE,
  join_by = "|",
  case_level = FALSE
)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.
`query`	A character. A query on potential outcomes such as "Y[X=1] - Y[X=0]"
`given`	A character. A quoted expression evaluates to logical statement. `given` allows the query to be conditioned on *observational* distribution. A value of TRUE is interpreted as no conditioning.
`using`	A character. Whether to use priors, posteriors or parameters
`parameters`	A vector of real numbers in [0,1]. A true parameter vector to be used instead of parameters attached to the model in case `using` specifies `parameters`
`type_distribution`	A numeric vector. If provided saves calculation, otherwise calculated from model; may be based on prior or posterior
`verbose`	Logical. Whether to print mean and standard deviation of the estimand on the console.
`join_by`	A character. The logical operator joining expanded types when `query` contains wildcard (`.`). Can take values `"&"` (logical AND) or `"|"` (logical OR). When restriction contains wildcard (`.`) and `join_by` is not specified, it defaults to `"|"`, otherwise it defaults to `NULL`.
`case_level`	Logical. If TRUE estimates the probability of the query for a case.

Value

A vector of draws from the distribution of the potential outcomes specified in query

Examples

```
model <- make_model("X -> Y") %>%
         set_prior_distribution() %>%
         set_parameter_matrix()

 distribution <- query_distribution(model, query = "(Y[X=1] - Y[X=0])")

 distribution <- query_distribution(model, query = "(Y[X=1] - Y[X=0])", given = "X==1")
distribution <- query_distribution(model, query = "(Y[X=1] - Y[X=0])", given = "Y[X=1]==0")
distribution <- query_distribution(model, query = "(Y[X=1] - Y[X=0])", given = "Y[X=1]==1")
 distribution <- query_distribution(model, query = "(Y[X=1] > Y[X=0])")
 distribution <- query_distribution(model,
                                    query = "(Y[X=1] > Y[X=0])",
                                    given = "X==1 & Y==1",
                                    verbose = TRUE)
 distribution <- query_distribution(model,
                                    query = "(Y[X=1] > Y[X=0])",
                                    given = "X==1 & Y==1",
                                    case_level = TRUE,
                                    verbose = TRUE)
 distribution <- query_distribution(model, query = "(Y[X=.] == 1)", join_by = "&")
distribution <- query_distribution(model, query = "(Y[X=1] - Y[X=0])", using = "parameters")
 df    <- simulate_data(model, n = 3)
 updated_model <- update_model(model, df)
 query_distribution( updated_model , query = "(Y[X=1] - Y[X=0])", using = "posteriors")
 updated_model <- update_model(model, df, keep_transformed = TRUE)
 query_distribution( updated_model , query = "(Y[X=1] - Y[X=0])", using = "posteriors")
```

---

query_model *Generate estimands dataframe*

Description

Calculated from a parameter vector, from a prior or from a posterior distribution

Usage

```
query_model(
  model,
  queries = NULL,
  given = NULL,
  using = list("parameters"),
  parameters = NULL,
  stats = NULL,
```

```
  n_draws = 4000,
  expand_grid = FALSE,
  case_level = FALSE,
  query = NULL
)
```

Arguments

model	A causal_model. A model object generated by make_model.
queries	A vector of characters. Query on potential outcomes such as "Y[X=1] - Y[X=0]".
given	A character. A quoted expression that evaluates to a logical statement. Allows estimand to be conditioned on *observational* (or counterfactual) distribution.
using	A character. Whether to use priors, posteriors or parameters.
parameters	A vector of real numbers in [0,1]. Values of parameters to specify (optional). By default, parameters is drawn from model$parameters_df.
stats	Functions to be applied to estimand distribution. If NULL, defaults to mean, standard deviation, and 95% confidence interval.
n_draws	An integer. Number of draws.
expand_grid	Logical. If TRUE then all combinations of provided lists are examined. If not then each list is cycled through separately. Defaults to FALSE.
case_level	Logical. If TRUE estimates the probability of the query for a case.
query	alias for queries

Value

A data.frame with columns Query, Given and Using defined by corresponding input values. Further columns are generated as specified in stats.

Examples

```
model <- make_model("X -> Y") %>% set_prior_distribution(n_draws = 10000)


estimands_df <-query_model(
               model,
              query = list(ATE = "Y[X=1] - Y[X=0]", Share_positive = "Y[X=1] > Y[X=0]"),
               using = c("parameters", "priors"),
               expand_grid = TRUE)

estimands_df <-query_model(
               model,
              query = list(ATE = "Y[X=1] - Y[X=0]", Share_positive = "Y[X=1] > Y[X=0]"),
               using = c("parameters", "priors"),
               expand_grid = FALSE)

estimands_df <- query_model(
                model,
                using = list( "parameters", "priors"),
```

```
                query = list(ATE = "Y[X=1] - Y[X=0]", Is_B = "Y[X=1] > Y[X=0]"),
                given = list(TRUE,  "Y==0 & X==1"),
                expand_grid = TRUE)

# An example: a stat representing uncertainty of token causation
token_var <- function(x) mean(x)*(1-mean(x))
estimands_df <- query_model(
                model,
                using = list( "parameters", "priors"),
                query = "Y[X=1] > Y[X=0]",
                stats = c(mean = mean, sd = sd, token_var = token_var))
```

`realise_outcomes` *Realise outcomes*

Description

Realise outcomes for all causal types. Calculated by sequentially calculating endogenous nodes. If a do operator is applied to any node then it takes the given value and all its descendants are generated accordingly.

Usage

```
realise_outcomes(model, dos = NULL, node = NULL)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.
`dos`	A named `list`. Do actions defining node values, e.g., `list(X = 0, M = 1)`.
`node`	A character. An optional quoted name of the node whose outcome should be revealed. If specified all values of parents need to be specified via `dos`.

Details

`realise_outcomes` starts off by creating types (via `get_nodal_types`). It then takes types of endogenous and reveals their outcome based on the value that their parents took. Exogenous nodes outcomes correspond to their type.

Value

A `data.frame` object of revealed data for each node (columns) given causal / nodal type (rows) .

Examples

```
model <- make_model("X -> Y")
realise_outcomes(model)

model <- make_model("X1->Y;X2->M;M->Y")
realise_outcomes(model, dos = list(X1 = 1, M = 0))

model <- make_model("X->M->Y")
realise_outcomes(model, dos = list(M = 1), node = "Y")
```

---

set_ambiguities_matrix

Set ambiguity matrix

Description

Add an ambiguities matrix to a model

Usage

```
set_ambiguities_matrix(model, A = NULL)
```

Arguments

model	A causal_model. A model object generated by make_model.
A	A data.frame. Ambiguity matrix. Not required but may be provided to avoid repeated computation for simulations.

Value

An object of type causal_model with the ambiguities matrix attached

Examples

```
model <- make_model('X -> Y') %>%
         set_ambiguities_matrix()
model$A
```

`set_confound` *Set confound*

Description

Adjust parameter matrix to allow confounding.

Usage

```
set_confound(model, confound = NULL)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.
`confound`	A `list` of statements indicating pairs of nodes whose types are jointly distributed (e.g. list("A <-> B", "C <-> D")).

Details

Confounding between X and Y arises when the nodal types for X and Y are not independently distributed. In the X -> Y graph, for instance, there are 2 nodal types for X and 4 for Y. There are thus 8 joint nodal types:

```
|          | t^X                |                    |           |
|----------|----|---------------|--------------------|-----------|
|          |    | 0             | 1                  | Sum       |
|----------|----|---------------|--------------------|-----------|
| t^Y | 00 | Pr(t^X=0 & t^Y=00) | Pr(t^X=1 & t^Y=00) | Pr(t^Y=00)|
|     | 10 | .                  | .                  | .         |
|     | 01 | .                  | .                  | .         |
|     | 11 | .                  | .                  | .         |
|----------|----|---------------|--------------------|-----------|
|     |Sum | Pr(t^X=0)          | Pr(t^X=1)          | 1         |
```

This table has 8 interior elements and so an unconstrained joint distribution would have 7 degrees of freedom. A no confounding assumption means that Pr(t^X | t^Y) = Pr(t^X), or Pr(t^X, t^Y) = Pr(t^X)Pr(t^Y). In this case there would be 3 degrees of freedom for Y and 1 for X, totaling 4 rather than 7.

`set_confounds` lets you relax this assumption by increasing the number of parameters characterizing the joint distribution. Using the fact that P(A,B) = P(A)P(B|A) new parameters are introduced to capture P(B|A=a) rather than simply P(B). For instance here two parameters (and one degree of freedom) govern the distribution of types X and four parameters (with 3 degrees of freedom) govern the types for Y given the type of X for a total of 1+3+3 = 7 degress of freedom.

Value

An object of class `causal_model` with updated parameters_df and parameter matrix.

Examples

```
make_model('X -> Y; X <-> Y') %>%
get_parameters()

make_model('X -> M -> Y; X <-> Y') %>%
get_parameters()

model <- make_model('X -> M -> Y; X <-> Y; M <-> Y')
model$parameters_df

# Example where set_confound is implemented after restrictions
make_model("A -> B -> C") %>%
set_restrictions(increasing("A", "B")) %>%
set_confound("B <-> C") %>%
get_parameters()

# Example where two parents are confounded
make_model('A -> B <- C; A <-> C') %>%
  set_parameters(node = "C", c(0.05, .95, .95, 0.05)) %>%
  make_data(n = 50) %>%
  cor()

 # Example with two confounds, added sequentially
model <- make_model('A -> B -> C') %>%
  set_confound(list("A <-> B", "B <-> C"))
model$statement
# plot(model)
```

`set_parameter_matrix` *Set parameter matrix*

Description

Add a parameter matrix to a model

Usage

```
set_parameter_matrix(model, P = NULL)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.
`P`	A `data.frame`. Parameter matrix. Not required but may be provided to avoid repeated computation for simulations.

Value

An object of class `causal_model`. It essentially returns a list containing the elements comprising a model (e.g. 'statement', 'nodal_types' and 'DAG') with the parameter matrix attached to it.

Examples

```
model <- make_model('X -> Y')
P <- diag(8)
colnames(P) <- rownames(model$causal_types)
model <- set_parameter_matrix(model, P = P)
```

`set_parmap` *Set parmap: a matrix mapping from parameters to data types*

Description

Generates and adds parmap to a model

Usage

```
set_parmap(model)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.

Value

A matrix

Examples

```
set_parmap(model = make_model('X->Y'))
```

`set_prior_distribution` *Add prior distribution draws*

Description

Add 'n_param x n_draws' database of possible lambda draws to the model.

Usage

```
set_prior_distribution(model, n_draws = 4000)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.
`n_draws`	A scalar. Number of draws.

Value

An object of class causal_model. It essentially returns a list containing the elements comprising a model (e.g. 'statement', 'nodal_types' and 'DAG') with the 'prior_distribution' attached to it.

See Also

Other prior_distribution: get_prior_distribution(), make_prior_distribution()

Examples

```
make_model('X -> Y') %>% set_prior_distribution(n_draws = 5) %>% get_prior_distribution()
```

---

set_restrictions *Restrict a model*

Description

Restrict a model's parameter space. This reduces the number of nodal types and in consequence the number of unit causal types.

Usage

```
set_restrictions(
  model,
  statement = NULL,
  join_by = "|",
  labels = NULL,
  given = NULL,
  keep = FALSE,
  update_types = TRUE,
  wildcard = FALSE
)
```

Arguments

model	A causal_model. A model object generated by make_model.
statement	A quoted expressions defining the restriction. If values for some parents are not specified, statements should be surrounded by parentheses, for instance (Y[A = 1] > Y[A=0]) will be interpreted for all combinations of other parents of Y set at possible levels they might take.
join_by	A string. The logical operator joining expanded types when statement contains wildcard (.). Can take values '&' (logical AND) or '|' (logical OR). When restriction contains wildcard (.) and join_by is not specified, it defaults to '|', otherwise it defaults to NULL. Note that join_by joins within statements, not across statements.

labels	A list of character vectors specifying nodal types to be kept or removed from the model. Use get_nodal_types to see syntax. Note that labels gets overwritten by statement if statement is not NULL.
given	A character vector or list of character vectors specifying nodes on which the parameter set to be restricted depends. When restricting by statement, given must either be NULL or of the same length as statement. When mixing statements that are further restricted by given and ones that are not, statements without given restrictions should have given specified as one of NULL, NA, "" or " ".
keep	Logical. If 'FALSE', removes and if 'TRUE' keeps only causal types specified by statement or labels.
update_types	Logical. If 'TRUE' the 'causal_types' matrix gets updated after application of restrictions.
wildcard	Logical. If 'TRUE' allows for use of wildcards in restriction string. Default 'FALSE'.

Details

Restrictions are made to nodal types, not to unit causal types. Thus for instance in a model X -> M -> Y, one cannot apply a simple restriction so that Y is nondecreasing in X, however one can restrict so that M is nondecreasing in X and Y nondecreasing in M. To have a restriction that Y be nondecreasing in X would otherwise require restrictions on causal types, not nodal types, which implies a form of undeclared confounding (i.e. that in cases in which M is decreasing in X, Y is decreasing in M).

Since restrictions are to nodal types, all parents of a node are implicitly fixed. Thus for model make_model(`X -> Y <- W`) the request set_restrictions(`(Y[X=1] == 0)`) is interpreted as set_restrictions(`(Y[X=1, W=0] == 0 | Y[X=1, W=1] == 0)`).

Statements with implicitly controlled nodes should be surrounded by parentheses, as in these examples.

Note that prior probabilities are redistributed over remaining types.

Value

An object of class model. The causal types and nodal types in the model are reduced according to the stated restriction.

See Also

Other restrictions: restrict_by_labels(), restrict_by_query()

Examples

```
# 1. Restrict parameter space using statements
model <- make_model('X->Y') %>%
  set_restrictions(statement = c('X[] == 0'))

model <- make_model('X->Y') %>%
  set_restrictions(non_increasing('X', 'Y'))
```

```
model <- make_model('X -> Y <- W') %>%
  set_restrictions(c(decreasing('X', 'Y'), substitutes('X', 'W', 'Y')))

model$parameters_df

model <- make_model('X-> Y <- W') %>%
  set_restrictions(statement = decreasing('X', 'Y'))
model$parameters_df

model <- make_model('X->Y') %>%
  set_restrictions(decreasing('X', 'Y'))
model$parameters_df

model <- make_model('X->Y') %>%
  set_restrictions(c(increasing('X', 'Y'), decreasing('X', 'Y')))
model$parameters_df

# Restrict to define a model with monotonicity
model <- make_model('X->Y') %>%
set_restrictions(statement = c('Y[X=1] < Y[X=0]'))
get_parameter_matrix(model)

# Restrict to a single type in endogenous node
model <- make_model('X->Y') %>%
set_restrictions(statement =  '(Y[X = 1] == 1)', join_by = '&', keep = TRUE)
get_parameter_matrix(model)

#  Use of | and &
# Keep node if *for some value of B* Y[A = 1] == 1
model <- make_model('A->Y<-B') %>%
set_restrictions(statement =  '(Y[A = 1] == 1)', join_by = '|', keep = TRUE)
dim(get_parameter_matrix(model))


# Keep node if *for all values of B* Y[A = 1] == 1
model <- make_model('A->Y<-B') %>%
set_restrictions(statement =  '(Y[A = 1] == 1)', join_by = '&', keep = TRUE)
dim(get_parameter_matrix(model))

# Restrict multiple nodes
model <- make_model('X->Y<-M; X -> M' ) %>%
set_restrictions(statement =  c('(Y[X = 1] == 1)', '(M[X = 1] == 1)'), join_by = '&', keep = TRUE)
get_parameter_matrix(model)

# Restrict using statements and given:
model <- make_model("X -> Y -> Z; X <-> Z") %>%
 set_restrictions(list(decreasing('X','Y'), decreasing('Y','Z')), given = c(NA,'X.0'))
get_parameter_matrix(model)

# Restrictions on levels for endogenous nodes aren't allowed
## Not run:
model <- make_model('X->Y') %>%
```

```
set_restrictions(statement =  '(Y == 1)')

## End(Not run)

# 2. Restrict parameter space Using labels:
model <- make_model('X->Y') %>%
set_restrictions(labels = list(X = '0', Y = '00'))

# Restrictions can be  with wildcards
model <- make_model('X->Y') %>%
set_restrictions(labels = list(Y = '?0'), wildcard = TRUE)
get_parameter_matrix(model)

# Deterministic model
model <- make_model('S -> C -> Y <- R <- X; X -> C -> R') %>%
set_restrictions(labels = list(C = '1000', R = '0001', Y = '0001'), keep = TRUE)
get_parameter_matrix(model)

# Restrict using labels and given:
model <- make_model("X -> Y -> Z; X <-> Z") %>%
 set_restrictions(labels = list(X = '0', Z = '00'), given = c(NA,'X.0'))
get_parameter_matrix(model)
```

`simulate_data` *simulate_data is an alias for make_data*

Description

simulate_data is an alias for make_data

Usage

```
simulate_data(...)
```

Arguments

`...`	arguments for `make_model`

Value

A `data.frame` with simulated data.

Examples

```
simulate_data(make_model("X->Y"))
```

strategy_statements *Generate strategy statements given data*

Description

Helper to generate statements of the form "X = 1 & Y = 0" from realized data on one observation

Usage

```
strategy_statements(data, strategies)
```

Arguments

data	A data frame with one row
strategies	A list of strategies where each strategy is a set of nodes to be observed

Value

A string

Examples

```
data.frame(X = 1, M = 0, Y = NA) %>%
strategy_statements(list(c("X", "M", "Y"), "X", "Y"))
```

substitutes *Make statement for substitutes*

Description

Generate a statement for X1, X1 substitute for each other in the production of Y

Usage

```
substitutes(X1, X2, Y)
```

Arguments

X1	A character. The quoted name of the input node 1.
X2	A character. The quoted name of the input node 2.
Y	A character. The quoted name of the outcome node.

Value

A character statement of class `statement`

See Also

Other statements: complements(), decreasing(), increasing(), interacts(), non_decreasing(), non_increasing(), te()

Examples

```
get_query_types(model = make_model('A -> B <- C'),
         query = substitutes('A', 'C', 'B'),map = "causal_type")

query_model(model = make_model('A -> B <- C'),
         queries = substitutes('A', 'C', 'B'),
         using = 'parameters')
```

---

te *Make treatment effect statement (positive)*

Description

Generate a statement for (Y(1) - Y(0)). This statement when applied to a model returns an element in (1,0,-1) and not a set of cases. This is useful for some purposes such as querying a model, but not for uses that require a list of types, such as `set_restrictions`.

Usage

```
te(X, Y)
```

Arguments

X	A character. The quoted name of the input node
Y	A character. The quoted name of the outcome node

Value

A character statement of class `statement`

See Also

Other statements: complements(), decreasing(), increasing(), interacts(), non_decreasing(), non_increasing(), substitutes()

Examples

```
te('A', 'B')

model <- make_model('X->Y') %>% set_restrictions(increasing('X', 'Y'))
query_model(model, list(ate = te('X', 'Y')),  using = 'parameters')

# set_restrictions  breaks with te because it requires a listing
# of causal types, not numeric output.

## Not run:
model <- make_model('X->Y') %>% set_restrictions(te('X', 'Y'))

## End(Not run)
```

`update_model` *Fit causal model using 'stan'*

Description

Takes a model and data and returns a model object with data attached and a posterior model

Usage

```
update_model(
  model,
  data = NULL,
  data_type = "long",
  keep_fit = FALSE,
  keep_transformed = TRUE,
  censored_types = NULL,
  ...
)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.
`data`	A `data.frame`. Data of nodes that can take three values: 0, 1, and NA. In long form as generated by `make_events`
`data_type`	Either 'long' (as made by `simulate_data`) or 'compact' (as made by `collapse_data`). Compact data must have entries for each member of each strategy family to produce a valid simplex.
`keep_fit`	Logical. Whether to append the `stanfit` object to the model. Defaults to 'FALSE'

keep_transformed
: Logical. Whether to keep transformed parameters, prob_of_types, P_lambdas, w, w_full

censored_types
: vector of data types that are selected out of the data, eg c("X0Y0")

...
: Options passed onto stan call.

Value

An object of class causal_model. It essentially returns a list containing the elements comprising a model (e.g. 'statement', 'nodal_types' and 'DAG') with the 'posterior_distribution' returned by stan attached to it.

Examples

```
model <- make_model('X->Y')
data_long   <- simulate_data(model, n = 4)
data_short  <- collapse_data(data_long, model)

model_1 <- update_model(model, data_long)


model_2 <- update_model(model, data_long, keep_transformed = FALSE)

## Not run:
# Throws error unless compact data indicated:

model_3 <- update_model(model, data_short)

## End(Not run)

model_4 <- update_model(model, data_short, data_type = 'compact')

# It is possible to implement updating without data, in which case the posterior
# is a stan object that reflects the prior
model_5 <- update_model(model)


# Censored data types
make_model("X->Y") %>%
  update_model(data.frame(X=c(1,1), Y=c(1,1)), censored_types = c("X1Y0")) %>%
  query_model(te("X", "Y"), using = "posteriors")

# Censored data: Learning nothing
make_model("X->Y") %>%
 update_model(data.frame(X=c(1,1), Y=c(1,1)), censored_types = c("X1Y0", "X0Y0", "X0Y1")) %>%
  query_model(te("X", "Y"), using = "posteriors")
```

Index