Package ‘CausalQueries’

February 22, 2025

Type Package

Title Make, Update, and Query Binary Causal Models

Version 1.3.3

Description Users can declare causal models over binary nodes, update beliefs about causal types given data, and calculate arbitrary queries. Updating is implemented in 'stan'. See Humphreys and Jacobs, 2023, Integrated Inferences (<DOI:10.1017/9781316718636>) and Pearl, 2009 Causality (<DOI:10.1017/CBO9780511803161>).

BugReports `https://github.com/integrated-inferences/CausalQueries/issues`

License MIT + file LICENSE

Encoding UTF-8

LazyData true

RoxygenNote 7.3.2

Depends methods, R (>= 4.2.0)

Imports dplyr, dirmult (>= 0.1.3-4), stats (>= 4.1.1), rlang (>= 0.2.0), rstan (>= 2.26.0), rstantools (>= 2.0.0), stringr (>= 1.4.0), latex2exp (>= 0.9.4), knitr (>= 1.45), ggplot2 (>= 3.3.5), lifecycle (>= 1.0.1), ggraph (>= 2.2.0), Rcpp (>= 0.12.0)

LinkingTo Rcpp (>= 0.12.0), BH (>= 1.66.0), RcppArmadillo, RcppEigen (>= 0.3.3.3.0), rstan (>= 2.26.0), StanHeaders (>= 2.26.0)

Suggests testthat, rmarkdown, DeclareDesign, fabricatr, estimatr, bayesplot, covr, curl

SystemRequirements GNU make

Biarch true

VignetteBuilder knitr

URL `https://integrated-inferences.github.io/CausalQueries/`

NeedsCompilation yes

Author Clara Bicalho [ctb],
Jasper Cooper [ctb],
Macartan Humphreys [aut] (<https://orcid.org/0000-0001-7029-2326>),
Till Tietz [aut, cre] (<https://orcid.org/0000-0002-2916-9059>),
Alan Jacobs [aut],
Merlin Heidemanns [ctb],
Lily Medina [aut] (<https://orcid.org/0009-0004-2423-524X>),
Julio Solis [ctb],
Georgiy Syunyaev [aut] (<https://orcid.org/0000-0002-4391-6313>)

Maintainer Till Tietz <ttietz2014@gmail.com>

Repository CRAN

Date/Publication 2025-02-22 11:20:02 UTC

Contents

`CausalQueries` *'CausalQueries'*

Description

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

Author(s)

Maintainer: Till Tietz <`ttietz2014@gmail.com`> (ORCID)

Authors:

- Macartan Humphreys <`macartan@gmail.com`> (ORCID)
- Alan Jacobs <`alan.jacobs@ubc.ca`>
- Lily Medina <`lilymiru@gmail.com`> (ORCID)
- Georgiy Syunyaev <`georgiy.syunyaev@vanderbilt.edu`> (ORCID)

Other contributors:

- Clara Bicalho <`clarabmcorreia@gmail.com`> [contributor]
- Jasper Cooper <`jjc2247@columbia.edu`> [contributor]
- Merlin Heidemanns <`mnh2123@columbia.edu`> [contributor]
- Julio Solis <`juliosolisar@gmail.com`> [contributor]

See Also

Useful links:

- `https://integrated-inferences.github.io/CausalQueries/`
- Report bugs at `https://github.com/integrated-inferences/CausalQueries/issues`

`data_helpers` *Data helpers*

Description

Various helpers to simulate data and to manipulate data types between compact and long forms.

`collapse_data` can be used to convert long form data to compact form data,

`expand_data` can be used to convert compact form data (one row per data type) to long form data (one row per observation).

`make_data` generates a dataset with one row per observation.

`make_events` generates a dataset with one row for each data type. Draws full data only. To generate various types of incomplete data see `make_data`.

Usage

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

expand_data(data_events = NULL, model)

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 = FALSE,
  ...
)

make_events(
  model,
  n = 1,
  w = NULL,
  P = NULL,
  A = NULL,
  parameters = NULL,
  param_type = NULL,
  include_strategy = 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‘.
`data_events`	A ’compact’ `data.frame` with one row per data type. Must be compatible with nodes in `model`. The default columns are `event`, `strategy` and `count`.
`n`	An integer. Number of observations.
`parameters`	A vector of real numbers in [0,1]. Values of parameters to specify (optional). By default, parameters is drawn from the parameters dataframe. See `inspect(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.
`...`	Arguments to be passed to make_priors if param_type == `define`
`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. See `inspect(model, "parameter_matrix")`.
`A`	A `data.frame`. Ambiguities matrix. Not required but may be provided to avoid repeated computation for simulations. `inspect(model, "ambiguities_matrix")`
`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.

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 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

A `data.frame` with rows as data observation

A `data.frame` with simulated data.

A `data.frame` of events

See Also

Other data_generation: `get_all_data_types()`, `make_data_single()`, `observe_data()`

Other data_generation: `get_all_data_types()`, `make_data_single()`, `observe_data()`

Examples

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

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

df |> collapse_data(model)

# Illustrating options

df |> collapse_data(model, drop_NA = FALSE)

df |> collapse_data(model, drop_family = TRUE)

df |> collapse_data(model, summary = TRUE)

# Appropriate behavior given restricted models

model <- make_model('X -> Y') |>
  set_restrictions('X[]==1')
df <- make_data(model, n = 10)
df[1,1] <- ''
df |> collapse_data(model)

df <- data.frame(X = 0:1)
df |> collapse_data(model)




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

```
# 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)

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

`democracy_data` *Development and Democratization: Data for replication of analysis in *Integrated Inferences**

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 data frame 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)
```

`get_all_data_types` *Get all data types*

Description

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

Usage

```
get_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.

See Also

Other data_generation: `data_helpers`, `make_data_single()`, `observe_data()`

Examples

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

---

get_event_probabilities

Draw event probabilities

Description

‘get_event_probabilities‘ draws event probability vector ‘w‘ given a single realization of parameters

Usage

```
get_event_probabilities(
  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 the parameters dataframe. See `inspect(model, "parameters_df")`.
`A`	A `data.frame`. Ambiguities matrix. Not required but may be provided to avoid repeated computation for simulations. `inspect(model, "ambiguities_matrix")`
`P`	A `data.frame`. Parameter matrix. Not required but may be provided to avoid repeated computation for simulations. See `inspect(model, "parameter_matrix")`.
`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_probabilities(model = model)
get_event_probabilities(model = model, given = "X==1")
get_event_probabilities(model = model, parameters = rep(1, 6))
get_event_probabilities(model = model, parameters = 1:6)
```

`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. An expression of the form "Y[X=1]" asks for the value of Y when X is set to 1
`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")

# Nested do operations
get_query_types(
 model = make_model('A -> B -> C -> D'),
 query = '(D[C=C[B=B[A=1]], A=0] > D[C=C[B=B[A=0]], A=0])')

# 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")
```

`inspection` *Helpers for inspecting causal models*

Description

Various helpers to inspect or access internal objects generated or used by Causal Models

Returns specified elements from a `causal_model` and prints summary. Users can use `inspect` to extract model's components or objects implied by the model structure including nodal types, causal types, parameter priors, parameter posteriors, type priors, type posteriors, and other relevant elements. See argument `what` for other options.

Returns specified elements from a `causal_model`. Users can use `inspect` to extract model's components or objects implied by the model structure including nodal types, causal types, parameter priors, parameter posteriors, type priors, type posteriors, and other relevant elements. See argument `what` for other options.

Usage

```
inspect(model, what = NULL, ...)

grab(model, what = NULL, ...)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.
`what`	A character string specifying the component to retrieve. Available options are: • `"statement"` a character string describing causal relations using dagitty syntax, • `"nodes"` A list containing the nodes in the model, • `"parents_df"` A table listing nodes, whether they are root nodes or not, and the number and names of parents they have,

- "`parameters`" A vector of ’true’ parameters,
- "`parameter_names`" A vector of names of parameters,
- "`parameter_mapping`" A matrix mapping from parameters into data types,
- "`parameter_matrix`" A matrix mapping from parameters into causal types,
- "`parameters_df`" A data frame containing parameter information,
- "`causal_types`" A data frame listing causal types and the nodal types that produce them,
- "`nodal_types`" A list with the nodal types of the model,
- "`data_types`" A list with all data types consistent with the model; for options see ?`get_all_data_types`,
- "`ambiguities_matrix`" A matrix mapping from causal types into data types,
- "`prior_hyperparameters`" A vector of alpha values used to parameterize Dirichlet prior distributions; optionally provide node names to reduce output, e.g., `inspect(prior_hyperparameters, nodes = c('M', 'Y'))`,
- "`prior_distribution`" A data frame of the parameter prior distribution,
- "`posterior_distribution`" A data frame of the parameter posterior distribution,
- "`type_prior`" A matrix of type probabilities using priors,
- "`type_posterior`" A matrix of type probabilities using posteriors,
- "`prior_event_probabilities`" A vector of data (event) probabilities given a single realization of parameters; for options see ?`get_event_probabilities`,
- "`posterior_event_probabilities`" A sample of data (event) probabilities from the posterior,
- "`data`" A data frame with data that was provided to update the model,
- `stan_summary`" A ‘stanfit‘ summary with processed parameter names,
- "`stanfit`" An (unprocessed) `stanfit` object as generated by Stan, with raw parameter names,
- "`stan_warnings`" Messages generated during the generation of a `stanfit` object.

`...`	Other arguments passed to helper "`get_*`" functions: get_all_data_types, get_event_probabilities, get_priors, Any such additional arguments must be named.

Value

Objects that can be derived from a `causal_model`, with summary.

Quiet return of objects that can be derived from a `causal_model`.

Examples

```
model <- make_model("X -> Y")
data <- make_data(model, n = 4)

inspect(model, what = "statement")
inspect(model, what = "parameters")
```

```
inspect(model, what = "nodes")
inspect(model, what = "parents_df")
inspect(model, what = "parameters_df")
inspect(model, what = "causal_types")
inspect(model, what = "prior_distribution")
inspect(model, what = "prior_hyperparameters", nodes = "Y")
inspect(model, what = "prior_event_probabilities", parameters = c(.1, .9, .25, .25, 0, .5))
inspect(model, what = "prior_event_probabilities", given = "Y==1")
inspect(model, what = "data_types", complete_data = TRUE)
inspect(model, what = "data_types", complete_data = FALSE)


model <- update_model(model,
  data = data,
  keep_fit = TRUE,
  keep_event_probabilities = TRUE)

inspect(model, what = "posterior_distribution")
inspect(model, what = "posterior_event_probabilities")
inspect(model, what = "type_posterior")
inspect(model, what = "data")
inspect(model, what = "stan_warnings")
inspect(model, what = "stanfit")


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

x <- grab(model, what = "statement")
x
```

`institutions_data` *Institutions and growth: Data for replication of analysis in *Integrated Inferences**

Description

A dataset containing dichotomized versions of variables in Rodrik, Subramanian, and Trebbi (2004).

Usage

```
institutions_data
```

Format

A data frame with 79 rows and 5 columns:

Y Income (GDP PPP 1995), dichotomized

R Institutions, (based on Kaufmann, Kraay, and Zoido-Lobaton (2002)) dichotomized

D Distance from the equator (in degrees), dichotomized

M Settler mortality (from Acemoglu, Johnson, and Robinson), dichotomized

country Country

Source

https://drodrik.scholar.harvard.edu/publications/institutions-rule-primacy-institutions-over-geogra

---

`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, nodes = 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'.
`nodes`	A vector of names of nodes. Can be used to limit interpretation to selected nodes.

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

`lipids_data` *Lipids: Data for Chickering and Pearl replication*

Description

A compact dataset containing information on an encouragement, (Z, cholestyramine prescription), a treatment (X, usage), and an outcome (Y, cholesterol). From David Maxwell Chickering and Judea Pearl: "A Clinician's Tool for Analyzing Non-compliance", AAAI-96 Proceedings. Chickering and Pearl in turn draw the data from Efron, Bradley, and David Feldman. "Compliance as an explanatory variable in clinical trials." Journal of the American Statistical Association 86.413 (1991): 9-17.

Usage

```
lipids_data
```

Format

A data frame with 8 rows and 3 columns:

event The data type

strategy For which nodes is data available

count Number of units with this data type

Source

https://cdn.aaai.org/AAAI/1996/AAAI96-188.pdf

`make_model` *Make a model*

Description

`make_model` uses causal statements encoded as strings to specify the 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 = "X -> Y", add_causal_types = TRUE, nodal_types = NULL)
```

Arguments

`statement`	character string. Statement describing causal relations between nodes. 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‘.
`nodal_types`	List of nodal types associated with model nodes

Value

An object of class `causal_model`.

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

`statement`	A character vector of the statement that defines the model
`dag`	A `data.frame` with columns ‘parent‘and ‘children‘ indicating how nodes relate to each other.
`nodes`	A named `list` with the nodes in the model
`parents_df`	A `data.frame` listing nodes, whether they are root nodes or not, and the number of parents they have
`nodal_types`	Optional: A named `list` with the nodal types in the model. List should be ordered according to the causal ordering of nodes. If NULL nodal types are generated. If FALSE, a parameters data frame is not generated.
`parameters_df`	A `data.frame` with descriptive information of the parameters in the model
`causal_types`	A `data.frame` listing causal types and the nodal types that produce them

See Also

`summary.causal_model` provides summary method for output objects of class `causal_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")
inspect(model, "parameter_matrix")
model <- make_model("Y2 <- X -> Y1; X <-> Y1; X <-> Y2")
dim(inspect(model, "parameter_matrix"))
inspect(model, "parameter_matrix")
model <- make_model("X1 -> Y <- X2; X1 <-> Y; X2 <-> Y")
dim(inspect(model, "parameter_matrix"))
inspect(model, "parameters_df")

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

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

## End(Not run)

nodal_types <-
  list(
    A = c("0","1"),
    B = c("0","1"),
    C = c("0","1"),
    D = c("0","1"),
    E = c("0","1"),
    Y = c(
      "00000000000000000000000000000000",
      "01010101010101010101010101010101",
      "00110011001100110011001100110011",
      "00001111000011110000111100001111",
      "00000000111111110000000011111111",
      "00000000000000001111111111111111",
      "11111111111111111111111111111111" ))

make_model("A -> Y; B ->Y; C->Y; D->Y; E->Y",
          nodal_types = nodal_types) |>
 inspect("parameters_df")

nodal_types = list(Y = c("01", "10"), Z = c("0", "1"))
make_model("Z -> Y", nodal_types = nodal_types) |>
 inspect("parameters_df")
```

`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 the parameters dataframe. See `inspect(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  <- make_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) |>  inspect("parameters")

# Simple examples
model <- make_model('X -> Y')
data  <- make_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"))
```

`print.causal_model` *Print a short summary for a causal model*

Description

print method for class "`causal_model`".

Usage

```
## S3 method for class 'causal_model'
print(x, ...)
```

Arguments

`x`	An object of `causal_model` class, usually a result of a call to `make_model` or `update_model`.
`...`	Further arguments passed to or from other methods.

Details

The information regarding the causal model includes the statement describing causal relations using `dagitty` syntax, number of nodal types per parent in a DAG, and number of causal types.

`print.model_query` *Print a tightened summary of model queries*

Description

print method for class `model_query`.

Usage

```
## S3 method for class 'model_query'
print(x, ...)
```

Arguments

`x`	An object of `model_query` class.
`...`	Further arguments passed to or from other methods.

`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, nodes = NULL)
```

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`	vector of strings. The name of specific parameter in the form of, for example, 'X.1', 'Y.01'
`nodes`	a vector of nodes

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 produce 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 parameters of the prior distribution of the nodal types ("hyperparameters").

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")
```

`query_distribution` *Calculate query distribution*

Description

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

Usage

```
query_distribution(
  model,
  queries = NULL,
  given = NULL,
  using = "parameters",
  parameters = NULL,
  n_draws = 4000,
  join_by = "|",
  case_level = FALSE,
  query = NULL
)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.
`queries`	A vector of strings or list of strings specifying queries on potential outcomes such as "Y[X=1] - Y[X=0]". Queries can also indicate conditioning sets by placing second queries after a colon: "Y[X=1] - Y[X=0] :|: X == 1 & Y == 1". Note a ':|:' is used rather than the traditional conditioning marker '|' to avoid confusion with logical operators.
`given`	A character vector specifying given conditions for each query. A 'given' is a quoted expression that evaluates to logical statement. `given` allows the query to be conditioned on either observed or counterfactural distributions. A value of TRUE is interpreted as no conditioning. A given statement can alternatively be provided after a colon in the query statement.
`using`	A character. Whether to use priors, posteriors or parameters
`parameters`	A vector or list of vectors 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`
`n_draws`	An integer. Number of draws.rm
`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.
`query`	alias for queries

Value

A data frame where columns contain draws from the distribution of the potential outcomes specified in `query`

Examples

```
model <- make_model("X -> Y") |>
         set_parameters(c(.5, .5, .1, .2, .3, .4))

 # simple  queries
 query_distribution(model, query = "(Y[X=1] > Y[X=0])", using = "priors") |>
   head()

 # multiple  queries
 query_distribution(model,
     query = list(PE = "(Y[X=1] > Y[X=0])", NE = "(Y[X=1] < Y[X=0])"),
     using = "priors")|>
   head()

 # multiple queries and givens, with ':' to identify conditioning distributions
 query_distribution(model,
   query = list(POC = "(Y[X=1] > Y[X=0]) :|: X == 1 & Y == 1",
                Q = "(Y[X=1] < Y[X=0]) :|: (Y[X=1] <= Y[X=0])"),
   using = "priors")|>
   head()

 # multiple queries and givens, using 'given' argument
 query_distribution(model,
   query = list("(Y[X=1] > Y[X=0])", "(Y[X=1] < Y[X=0])"),
   given = list("Y==1", "(Y[X=1] <= Y[X=0])"),
   using = "priors")|>
   head()

 # linear queries
 query_distribution(model, query = "(Y[X=1] - Y[X=0])")


 # Linear query conditional on potential outcomes
 query_distribution(model, query = "(Y[X=1] - Y[X=0]) :|: Y[X=1]==0")

 # Use join_by to amend query interpretation
 query_distribution(model, query = "(Y[X=.] == 1)", join_by = "&")

 # Probability of causation query
 query_distribution(model,
    query = "(Y[X=1] > Y[X=0])",
    given = "X==1 & Y==1",
    using = "priors")  |> head()

 # Case level probability of causation query
 query_distribution(model,
    query = "(Y[X=1] > Y[X=0])",
```

```
     given = "X==1 & Y==1",
     case_level = TRUE,
     using = "priors")

# Query posterior
update_model(model, make_data(model, n = 3)) |>
query_distribution(query = "(Y[X=1] - Y[X=0])", using = "posteriors") |>
head()

# Case level queries provide the inference for a case, which is a scalar
# The case level query *updates* on the given information
# For instance, here we have a model for which we are quite sure that X
# causes Y but we do not know whether it works through two positive effects
# or two negative effects. Thus we do not know if M=0 would suggest an
# effect or no effect

set.seed(1)
model <-
  make_model("X -> M -> Y") |>
  update_model(data.frame(X = rep(0:1, 8), Y = rep(0:1, 8)), iter = 10000)

Q <- "Y[X=1] > Y[X=0]"
G <- "X==1 & Y==1 & M==1"
QG <- "(Y[X=1] > Y[X=0]) & (X==1 & Y==1 & M==1)"

# In this case these are very different:
query_distribution(model, Q, given = G, using = "posteriors")[[1]] |> mean()
query_distribution(model, Q, given = G, using = "posteriors",
  case_level = TRUE)

# These are equivalent:
# 1. Case level query via function
query_distribution(model, Q, given = G,
   using = "posteriors", case_level = TRUE)

# 2. Case level query by hand using Bayes' rule
query_distribution(
    model,
    list(QG = QG, G = G),
    using = "posteriors") |>
   dplyr::summarize(mean(QG)/mean(G))
```

`query_helpers` *Query helpers*

Description

Various helpers to describe queries or parts of queries in natural language.

Generate a statement for Y monotonic (increasing) in X

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

Generate a statement for Y monotonic (decreasing) in X

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

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

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

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

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

```
increasing(X, Y)

non_decreasing(X, Y)

decreasing(X, Y)

non_increasing(X, Y)

interacts(X1, X2, Y)

complements(X1, X2, Y)

substitutes(X1, X2, Y)

te(X, Y)
```

Arguments

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

Value

A character statement of class `statement`

A character statement of class `statement`

A character statement of class `statement`

A character statement of class `statement`

A character statement of class `statement`

A character statement of class `statement`

A character statement of class `statement`

A character statement of class `statement`

Examples

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


non_decreasing('A', 'B')


decreasing('A', 'B')


non_increasing('A', 'B')


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


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


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('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)
```

`query_model` *Generate data frame for batches of causal queries*

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 = NULL,
  case_level = FALSE,
  query = NULL,
  cred = 95,
  labels = NULL
)
```

Arguments

`model`	A `causal_model`. A model object generated by `make_model`.
`queries`	A vector of strings or list of strings specifying queries on potential outcomes such as "Y[X=1] - Y[X=0]". Queries can also indicate conditioning sets by placing second queries after a colon: "Y[X=1] - Y[X=0] :|: X == 1 & Y == 1". Note a colon, ’:|:’ is used rather than the traditional conditioning marker ’|’ to avoid confusion with logical operators.
`given`	A character vector specifying given conditions for each query. A ’given’ is a quoted expression that evaluates to logical statement. `given` allows the query to be conditioned on either observed or counterfactural distributions. A value of TRUE is interpreted as no conditioning. A given statement can alternatively be provided after a colon in the query statement.
`using`	A vector or list of strings. 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 the parameters dataframe. See `inspect(model, "parameters_df")`.
`stats`	Functions to be applied to the query distribution. If NULL, defaults to mean, standard deviation, and 95% confidence interval. Functions should return a single numeric value.
`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

`cred`	size of the credible interval ranging between 0 and 100
`labels`	labels for queries: if provided labels should have the length of the combinations of requests

Details

Queries can condition on observed or counterfactual quantities. Nested or "complex" counterfactual queries of the form `Y[X=1, M[X=0]]` are allowed.

Value

An object of class `model_query`. A data frame with possible columns: model, query, given, using, case_level, mean, sd, cred.low, cred.high. Further columns are generated as specified in `stats`.

Examples

```
model <- make_model("X -> Y")
query_model(model, "Y[X=1] - Y[X = 0]", using = "priors")
query_model(model, "Y[X=1] - Y[X = 0] :|: X==1 & Y==1", using = "priors")
query_model(model,
  list("Y[X=1] - Y[X = 0]",
       "Y[X=1] - Y[X = 0] :|: X==1 & Y==1"),
  using = "priors")
query_model(model, "Y[X=1] > Y[X = 0]", using = "parameters")
query_model(model, "Y[X=1] > Y[X = 0]", using = c("priors", "parameters"))


# `expand_grid= TRUE` requests the Cartesian product of arguments

models <- list(
 M1 = make_model("X -> Y"),
 M2 = make_model("X -> Y") |>
   set_restrictions("Y[X=1] < Y[X=0]")
 )

# No expansion: lists should be equal length
query_model(
  models,
  query = list(ATE = "Y[X=1] - Y[X=0]",
               Share_positive = "Y[X=1] > Y[X=0]"),
  given = c(TRUE,  "Y==1 & X==1"),
  using = c("parameters", "priors"),
  expand_grid = FALSE)

# Expansion when query and given arguments coupled
query_model(
  models,
  query = list(ATE = "Y[X=1] - Y[X=0]",
               Share_positive = "Y[X=1] > Y[X=0] :|: Y==1 & X==1"),
  using = c("parameters", "priors"),
  expand_grid = TRUE)
```

```
# Expands over query and given argument when these are not coupled
query_model(
  models,
  query = list(ATE = "Y[X=1] - Y[X=0]",
               Share_positive = "Y[X=1] > Y[X=0]"),
  given = c(TRUE,  "Y==1 & X==1"),
  using = c("parameters", "priors"),
  expand_grid = TRUE)

# An example of a custom statistic: uncertainty of token causation
f <- function(x) mean(x)*(1-mean(x))

query_model(
  model,
  using = list( "parameters", "priors"),
  query = "Y[X=1] > Y[X=0]",
  stats = c(mean = mean, sd = sd, token_variance = f))
```

`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, add_rownames = TRUE)
```

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`.
`add_rownames`	logical indicating whether to add causal types as rownames to the output

Details

If a node is not specified all outcomes are realised for all possible causal types consistent with the model. If a node is specified then outcomes of Y are returned conditional on different values of parents, whether or not these values of the parents obtain given restrictions under the model.

`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

```
make_model("X -> Y") |>
  realise_outcomes()

make_model("X -> Y <- W") |>
set_restrictions(labels = list(X = "1", Y="0010"),
                 keep = TRUE) |>
 realise_outcomes()

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

# With node specified
make_model("X->M->Y") |>
realise_outcomes(node = "Y")

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

`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_confound` 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 degrees of freedom.

Value

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

See Also

Other set: `set_prior_distribution()`, `set_restrictions()`

Examples

```
make_model('X -> Y; X <-> Y') |>
inspect("parameters")

make_model('X -> M -> Y; X <-> Y') |>
inspect("parameters")

model <- make_model('X -> M -> Y; X <-> Y; M <-> Y')
inspect(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") |>
inspect("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"))
inspect(model, "statement")
# plot(model)
```

`set_prior_distribution` *Add prior distribution draws*

Description

Add ‘n_param x n_draws‘ database of possible parameter 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` with the ‘prior_distribution‘ attached to it.

See Also

Other set: `set_confound()`, `set_restrictions()`

Examples

```
make_model('X -> Y') |>
  set_prior_distribution(n_draws = 5) |>
  inspect("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,
  param_names = NULL,
  given = NULL,
  keep = 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.
`param_names`	A character vector of names of parameters to restrict on.
`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`.

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 set: `set_confound()`, `set_prior_distribution()`

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')))

inspect(model, "parameters_df")

model <- make_model('X-> Y <- W') |>
  set_restrictions(statement = decreasing('X', 'Y'))
inspect(model, "parameters_df")

model <- make_model('X->Y') |>
  set_restrictions(decreasing('X', 'Y'))
inspect(model, "parameters_df")

model <- make_model('X->Y') |>
  set_restrictions(c(increasing('X', 'Y'), decreasing('X', 'Y')))
inspect(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]'))
inspect(model, "parameter_matrix")

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

#  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(inspect(model ,"parameter_matrix"))


# 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(inspect(model, "parameter_matrix"))

# 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)
inspect(model, "parameter_matrix")

# 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'))
inspect(model, "parameter_matrix")

# 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'))
inspect(model, "parameter_matrix")

# 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)
inspect(model, "parameter_matrix")
```

```
# 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'))
inspect(model, "parameter_matrix")
```

`summary.causal_model` *Summarizing causal models*

Description

summary method for class "`causal_model`".

Usage

```
## S3 method for class 'causal_model'
summary(object, include = NULL, ...)

## S3 method for class 'summary.causal_model'
print(x, what = NULL, ...)
```

Arguments

`object`	An object of `causal_model` class produced using `make_model` or `update_model`.
`include`	A character string specifying the additional objects to include in summary. Defaults to `NULL`. See details for full list of available values.
`...`	Further arguments passed to or from other methods.
`x`	An object of `summary.causal_model` class, produced using `summary.causal_model`.
`what`	A character string specifying the objects summaries to print. Defaults to `NULL` printing causal statement, specification of nodal types and summary of model restrictions. See details for full list of available values.

Details

In addition to the default objects included in ‘summary.causal_model‘ users can request additional objects via ‘include‘ argument. Note that these additional objects can be large for complex models and can increase computing time. The ‘include‘ argument can be a vector of any of the following additional objects:

- `"parameter_matrix"` A matrix mapping from parameters into causal types,
- `"parameter_mapping"` a matrix mapping from parameters into data types,
- `"causal_types"` A data frame listing causal types and the nodal types that produce them,
- `"prior_distribution"` A data frame of the parameter prior distribution,
- `"ambiguities_matrix"` A matrix mapping from causal types into data types,

- `"type_prior"` A matrix of type probabilities using priors.

`print.summary.causal_model` reports causal statement, full specification of nodal types and summary of model restrictions. By specifying ‘what‘ argument users can instead print a custom summary of any set of the following objects contained in the ‘summary.causal_model‘:

- `"statement"` A character string giving the causal statement,
- `"nodes"` A list containing the nodes in the model,
- `"parents"` A list of parents of all nodes in a model,
- `"parents_df"` A data frame listing nodes, whether they are root nodes or not, and the number and names of parents they have,
- `"parameters"` A vector of ’true’ parameters,
- `"parameters_df"` A data frame containing parameter information,
- `"parameter_names"` A vector of names of parameters,
- `"parameter_mapping"` A matrix mapping from parameters into data types,
- `"parameter_matrix"` A matrix mapping from parameters into causal types,
- `"causal_types"` A data frame listing causal types and the nodal types that produce them,
- `"nodal_types"` A list with the nodal types of the model,
- `"data_types"` A list with the all data types consistent with the model; for options see ‘"?get_all_data_types"‘,
- `"prior_hyperparameters"` A vector of alpha values used to parameterize Dirichlet prior distributions; optionally provide node names to reduce output ‘inspect(prior_hyperparameters, c(’M’, ’Y’))‘
- `"prior_distribution"` A data frame of the parameter prior distribution,
- `"prior_event_probabilities"` A vector of data (event) probabilities given a single (sepcified) parameter vector; for options see ‘"?get_event_probabilities"‘,
- `"ambiguities_matrix"` A matrix mapping from causal types into data types,
- `"type_prior"` A matrix of type probabilities using priors,
- `"type_posterior"` A matrix of type probabilities using posteriors,
- `"posterior_distribution"` A data frame of the parameter posterior distribution,
- `"posterior_event_probabilities"` A sample of data (event) probabilities from the posterior,
- `"data"` A data frame with data that was used to update model,
- `"stanfit"` A ‘stanfit‘ object generated by Stan,
- `"stan_summary"` A ‘stanfit‘ summary with updated parameter names.

Value

Returns the object of class `summary.causal_model` that preserves the list structure of `causal_model` class and adds the following additional objects:

- `"parents"` a list of parents of all nodes in a model,
- `"parameters"` a vector of ’true’ parameters,

- `"parameter_names"` a vector of names of parameters,
- `"data_types"` a list with the all data types consistent with the model; for options see `"?get_all_data_types"`,
- `"prior_event_probabilities"` a vector of prior data (event) probabilities given a parameter vector; for options see `"?get_event_probabilities"`,
- `"prior_hyperparameters"` a vector of alpha values used to parameterize Dirichlet prior distributions; optionally provide node names to reduce output `"inspect(prior_hyperparameters, c('M', 'Y'))"`

Examples

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

model |>
  update_model(
    keep_event_probabilities = TRUE,
    keep_fit = TRUE,
    data = make_data(model, n = 100)
  ) |>
  summary()



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

model <-
  model |>
  update_model(
    keep_event_probabilities = TRUE,
    keep_fit = TRUE,
    data = make_data(model, n = 100)
  )

print(summary(model), what = "type_posterior")
print(summary(model), what = "posterior_distribution")
print(summary(model), what = "posterior_event_probabilities")
print(summary(model), what = "data_types")
print(summary(model), what = "prior_hyperparameters")
print(summary(model), what = c("statement", "nodes"))
print(summary(model), what = "parameters_df")
print(summary(model), what = "posterior_event_probabilities")
print(summary(model), what = "posterior_distribution")
print(summary(model), what = "data")
print(summary(model), what = "stanfit")
print(summary(model), what = "type_posterior")

# Large objects have to be added to the summary before printing
print(summary(model, include = "ambiguities_matrix"),
  what = "ambiguities_matrix")
```

`summary.model_query` *Summarizing model queries*

Description

summary method for class "`model_query`".

Usage

```
## S3 method for class 'model_query'
summary(object, ...)

## S3 method for class 'summary.model_query'
print(x, ...)
```

Arguments

`object`	An object of `model_query` class produced using `query_model`
`...`	Further arguments passed to or from other methods.
`x`	an object of `model_query` class produced using `query_model`

Value

Returns the object of class `summary.model_query`

Examples

```
model <-
  make_model("X -> Y") |>
  query_model("Y[X=1] > Y[X=1]")  |>
  summary()
```

`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 = NULL,
  keep_type_distribution = TRUE,
  keep_event_probabilities = FALSE,
  keep_fit = FALSE,
  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 `make_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. When long form data is provided with missingness, missing data is assumed to be missing at random.
`keep_type_distribution`	Logical. Whether to keep the (transformed) distribution of the causal types. Defaults to 'TRUE'
`keep_event_probabilities`	Logical. Whether to keep the (transformed) distribution of event probabilities. Defaults to 'FALSE'
`keep_fit`	Logical. Whether to keep the `stanfit` object produced by sampling for further inspection. See `?stanfit` for more details. Defaults to 'FALSE'. Note the `stanfit` object has internal names for parameters (lambda), event probabilities (w), and the type distribution (types)
`censored_types`	vector of data types that are selected out of the data, e.g. `c("X0Y0")`
`...`	Options passed onto sampling call. For details see `?rstan::sampling`

Value

An object of class `causal_model` with posterior distribution on parameters and other elements generated by updating; all elements accessible via `get` and `inspect`.

See Also

`make_model` to create a new model, `summary.causal_model` provides a summary method for output objects of class `causal_model`

Examples

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

  # It is possible to implement updating without data, in which
  # case the posterior is a stan object that reflects the prior

  update_model(model)

## Not run:

  # Censored data types illustrations
  # Here we update less than we might because we are aware of filtered data

  data <- data.frame(X=rep(0:1, 10), Y=rep(0:1,10))
  uncensored <-
    make_model("X->Y") |>
    update_model(data) |>
    query_model(te("X", "Y"), using = "posteriors")

  censored <-
    make_model("X->Y") |>
    update_model(
      data,
      censored_types = c("X1Y0")) |>
    query_model(te("X", "Y"), using = "posteriors")


  # Censored data: We learn nothing because the data
  # we see is the only data we could ever see
  make_model("X->Y") |>
    update_model(
      data,
      censored_types = c("X1Y0", "X0Y0", "X0Y1")) |>
    query_model(te("X", "Y"), using = "posteriors")

## End(Not run)
```

Index