Package 'simDAG'

November 2, 2026

Title Simulate Data from a (Time-Dependent) Causal DAG

Version 1.1.0

Maintainer Robin Denz <robin.denz@rub.de>

Description Simulate complex data from a given directed acyclic graph and information about each individual node.
Root nodes are simply sampled from the specified distribution. Child Nodes are simulated according to
one of many implemented regressions, such as logistic regression, linear
regression, poisson regression or any other function. Also includes a comprehensive framework for discrete-time
simulation, discrete-event simulation, and networks-
based simulation which can generate even more complex longitudinal and dependent data.
For more details, see Robin Denz, Nina Timmesfeld (2026) <doi:10.18637/jss.v116.i02>.

License GPL (>= 3)

URL https://github.com/RobinDenz1/simDAG,
https://robindenz1.github.io/simDAG/

BugReports https://github.com/RobinDenz1/siMDAG/issues

Imports data.table (>= 1.15.0), Rfast, rlang, igraph (>= 2.0.0),
dagitty, ggdag

Suggests knitr, rmarkdown, testthat (>= 3.0.0), vdiffr (>= 1.0.0),
ggplot2, ggforce, MASS, covr, foreach, doSNOW, doRNG, parallel,
utils, simr, rsurv, survival, calibrar

VignetteBuilder knitr

Config/testthat/edition 3

Contact <robin.denz@rub.de>

Encoding UTF-8

Config/roxygen2/version 8.1.1

NeedsCompilation no

Author Robin Denz [aut, cre],
Katharina Meiszl [aut]

Repository CRAN

Date/Publication 2026-11-02 15:20:02 UTC

Contents

simDAG-package . . . 3
add_node . . . 5
as.dagitty.DAG . . . 6
as.igraph.DAG . . . 7
as_tidy_dagitty.DAG . . . 9
calibrate_dag . . . 10
dag2matrix . . . 16
dag_from_data . . . 18
do . . . 20
empty_dag . . . 22
lazy_dag . . . 23
long2start_stop . . . 24
matrix2dag . . . 26
net . . . 27
network . . . 29
node . . . 33
node_aalen . . . 37
node_beta . . . 39
node_binomial . . . 41
node_competing_events . . . 44
node_conditional_distr . . . 48
node_conditional_prob . . . 51
node_cox . . . 54
node_gamma . . . 57
node_gaussian . . . 59
node_identity . . . 62
node_mixture . . . 65
node_multinomial . . . 67
node_negative_binomial . . . 69
node_next_time . . . 70
node_poisson . . . 74
node_polr . . . 77
node_rsurv . . . 79
node_time_to_event . . . 82
node_zeroinfl . . . 87
plot.calibrated_DAG . . . 90
plot.DAG . . . 92
plot.simDT . . . 95
rbernoulli . . . 99
rcategorical . . . 101
rconstant . . . 102
rsample . . . 103

rtexp 104
set_lazy_dag 105
sim2data 106
sim_discrete_event 111
sim_discrete_time 117
sim_from_dag 123
sim_n_datasets 125

Index **128**

`simDAG-package` *Simulate Data from a DAG and Associated Node Information*

Description

What is this package about?

This package aims to give a comprehensive framework to simulate static and longitudinal data given a directed acyclic graph and some information about each node. Our goal is to make this package as user-friendly and intuitive as possible, while allowing extreme flexibility and while keeping the underlying code as fast and RAM efficient as possible.

What features are included in this package?

This package includes three main simulation functions: the `sim_from_dag` function, which can be used to simulate data from a previously defined causal DAG and node information, the `sim_discrete_time` function, which implements a framework to conduct discrete-time simulations and the `sim_discrete_event` function for discrete-event simulations. The former is very easy to use, but cannot deal with time-varying variable easily. The latter two are a little more difficult to use (usually requiring the user to write some functions himself), but allow the simulation of arbitrarily complex longitudinal data in discrete and continuous time.

Through a collection of implemented node types, this package allows the user to generate data with a mix of binary, categorical, count and time-to-event data. The `sim_discrete_time` and `sim_discrete_event` functions additionally enable the user to generate time-to-event data with, if desired, a mix of competing events, recurrent events, time-varying variables that influence each other and any types of censoring.

The package also includes a few functions to transform resulting data into multiple formats, to augment existing DAGs, to plot DAGs and to plot a flow-chart of the data generation process.

All of the above mentioned features may also be combined with networks-based simulation, in which user-specified network dependencies among individuals may be used directly when specifying nodes. One or multiple networks (directed or undirected, weighted or unweighted) that may or may not change over time (possibly as a function of other variables) are supported.

What does a typical workflow using this package look like?

Users should start by defining a `DAG` object using the `empty_dag` and `node` functions. This `DAG` can then be passed to one of the two simulation functions included in this package. More information on how to do this can be found in the respective documentation pages and the three vignettes of this package.

When should I use `sim_from_dag` ***and when*** `sim_discrete_time`***?***

If you want to simulate data that is easily described using a standard DAG without time-varying variables, you should use the `sim_from_dag` function. If the DAG includes time-varying variables, but you only want to consider a few points in time and can easily describe the relations between those manually, you can still use the `sim_from_dag` function. If you want more complex data with time-varying variables, particularly with time-to-event outcomes, you should consider using the `sim_discrete_time` or `sim_discrete_event` functions.

What features are missing from this package?

The package currently only implements some possible child nodes. In the future we would like to implement more child node types, such as more complex survival time models and extending the already existing support for multilevel modeling to other node types.

Why should I use this package instead of the* simcausal *package?

The **simcausal** package was a big inspiration for this package. In contrast to it, however, it allows quite a bit more flexibility. A big difference is that this package includes a comprehensive framework for discrete-time and discrete-event simulations and the **simcausal** package does not.

Where can I get more information?

The documentation pages contain a lot of information, relevant examples and some literature references. Additional examples can be found in the vignettes of this package, which can be accessed using:

- `vignette(topic="v_sim_from_dag", package="simDAG")`
- `vignette(topic="v_sim_discrete_time", package="simDAG")`
- `vignette(topic="v_sim_discrete_event", package="simDAG")`
- `vignette(topic="v_covid_example", package="simDAG")`
- `vignette(topic="v_using_formulas", package="simDAG")`
- `vignette(topic="v_custom_nodes", package="simDAG")`
- `vignette(topic="v_cookbook", package="simDAG")`
- `vignette(topic="v_sim_networks", package="simDAG")`
- `vignette(topic="v_calibrate_dag", package="simDAG")`

A separate peer-reviewed article about this package has been published in the *Journal of Statistical Software* (Denz and Timmesfeld 2026). It is also available as a vignette in this package.

I have a problem using the* `sim_discrete_time` *or* `sim_discrete_event` *function

The `sim_discrete_time` and `sim_discrete_event` functions can become difficult to use depending on what kind of data the user wants to generate. For this reason we put in extra effort to make the documentation and examples as clear and helpful as possible. Please consult the relevant documentation pages and the vignettes before contacting the authors directly with programming related questions that are not clearly bugs in the code.

I want to suggest a new feature / I want to report a bug. Where can I do this?

Bug reports, suggestions and feature requests are highly welcome. Please file an issue on the official github page or contact the author directly using the supplied e-mail address.

Author(s)

Robin Denz, <robin.denz@rub.de>

References

Denz, Robin and Nina Timmesfeld (2026). "Simulating Complex Cross-Sectional and Longitudinal Data using the simDAG R Package". Journal of Statistical Software 116 (2), doi: 10.18637/jss.v116.i02.

Banks, Jerry, John S. Carson II, Barry L. Nelson, and David M. Nicol (2014). Discrete-Event System Simulation. Vol. 5. Edinburgh Gate: Pearson Education Limited.

---

add_node *Add a* DAG.node *or a* DAG.network *object to a* DAG *object*

Description

This function allows users to add DAG.node objects created using the node or node_td function and DAG.network objects created using the network or network_td function to DAG objects created using the empty_dag function, which makes it easy to fully specify a DAG to use in the sim_from_dag function and sim_discrete_time.

Usage

```
add_node(dag, node)

## S3 method for class 'DAG'
object_1 + object_2
```

Arguments

dag	A DAG object created using the empty_dag function.
node	Either a DAG.node object created using the node function or node_td function, or a DAG.network object created using the network function or network_td function.
object_1	Either a DAG object, a DAG.node object or a DAG.network object. The order of the objects does not change the result.
object_2	See argument object_1.

Details

The two ways of adding a node or a network to a DAG object are: dag <- add_node(dag, node(...)) and dag <- dag + node(...), which give identical results (note that the ... should be replaced with actual arguments and that the initial dag should be created with a call to empty_dag). See node for more information on how to specify a DAG for use in the sim_from_dag and node_td functions.

Value

Returns an DAG object with the DAG.node object or DAG.network object added to it.

Author(s)

Robin Denz

Examples

```
library(simDAG)

## add nodes to DAG using +
dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=5) +
  node("sex", type="rbernoulli", p=0.5) +
  node("income", type="gaussian", parents=c("age", "sex"), betas=c(1.1, 0.2),
       intercept=-5, error=4)

## add nodes to DAG using add_node()
dag <- empty_dag()
dag <- add_node(dag, node("age", type="rnorm", mean=50, sd=5))
```

---

as.dagitty.DAG *Transform a* DAG *object into a* dagitty *object*

Description

This function extends the as.dagitty function from the dagitty package to allow the input of a DAG object. The result is a dagitty object that includes only the structure of the DAG, without any specifications. May be useful to perform identifiability checks etc. on the DAG.

Usage

```
## S3 method for class 'DAG'
as.dagitty(x, include_root_nodes=TRUE,
           include_td_nodes=TRUE, include_networks=FALSE,
           layout=FALSE, ...)
```

Arguments

x	A DAG object created using the empty_dag function with nodes added to it using the + syntax. See ?empty_dag or ?node for more details. Supports DAGs with time-dependent nodes added using the node_td function. However, including such DAGs may result in cyclic causal structures, because time is not represented in the output matrix.
include_root_nodes	Whether to include root nodes in the output matrix. Should usually be kept at TRUE (default).
include_td_nodes	Whether to include time-dependent nodes added to the dag using the node_td function or not. When including these types of nodes, it is possible for the adjacency matrix to contain cycles, e.g. that it is not a classic DAG anymore, due to the matrix not representing the passage of time.

include_networks
: Whether to include time-fixed networks added to the dag using the network function or not. Usually it does not make sense to include those, because they are not classical nodes.

layout
: Corresponds to the argument of the same name in the dagitty function.

...
: Currently not used.

Value

Returns a dagitty object.

Author(s)

Robin Denz

See Also

empty_dag, node, node_td, as.igraph.DAG

Examples

```
library(simDAG)

# some example DAG
dag <- empty_dag() +
  node("death", type="binomial", parents=c("age", "sex"), betas=c(1, 2),
       intercept=-10) +
  node("age", type="rnorm", mean=10, sd=2) +
  node("sex", parents="", type="rbernoulli", p=0.5) +
  node("smoking", parents=c("sex", "age"), type="binomial",
       betas=c(0.6, 0.2), intercept=-2)

if (requireNamespace("dagitty")) {
  g <- dagitty::as.dagitty(dag)
}
```

---

as.igraph.DAG *Transform a* DAG *object into an* igraph *object*

Description

This function extends the as.igraph function from the igraph package to allow the input of a DAG object. The result is an igraph object that includes only the structure of the DAG, without any specifications. May be useful for plotting purposes.

Usage

```
## S3 method for class 'DAG'
as.igraph(x, include_root_nodes=TRUE,
          include_td_nodes=TRUE, include_networks=FALSE, ...)
```

Arguments

x	A DAG object created using the empty_dag function with nodes added to it using the + syntax. See ?empty_dag or ?node for more details. Supports DAGs with time-dependent nodes added using the node_td function. However, including such DAGs may result in cyclic causal structures, because time is not represented in the output matrix.
include_root_nodes	Whether to include root nodes in the output matrix. Should usually be kept at TRUE (default).
include_td_nodes	Whether to include time-dependent nodes added to the dag using the node_td function or not. When including these types of nodes, it is possible for the adjacency matrix to contain cycles, e.g. that it is not a classic DAG anymore, due to the matrix not representing the passage of time.
include_networks	Whether to include time-fixed networks added to the dag using the network function or not. Usually it does not make sense to include those, because they are not classical nodes.
...	Currently not used.

Value

Returns an igraph object.

Author(s)

Robin Denz

See Also

empty_dag, node, node_td

Examples

```
library(simDAG)

# some example DAG
dag <- empty_dag() +
  node("death", type="binomial", parents=c("age", "sex"), betas=c(1, 2),
       intercept=-10) +
  node("age", type="rnorm", mean=10, sd=2) +
  node("sex", parents="", type="rbernoulli", p=0.5) +
  node("smoking", parents=c("sex", "age"), type="binomial",
       betas=c(0.6, 0.2), intercept=-2)

if (requireNamespace("igraph")) {
  g <- igraph::as.igraph(dag)
  plot(g)
}
```

as_tidy_dagitty.DAG *Transform a* DAG *object into a* tidy_dagitty *object*

Description

This function extends the as_tidy_dagitty function from the **ggdag** package to allow the input of a DAG object. The result is a tidy_dagitty object that includes only the structure of the DAG, without any specifications. May be useful to plot DAGs using the **ggdag** package. Note that completely unconnected nodes (no arrows going in or out) are ignored by this function.

Usage

```
## S3 method for class 'DAG'
as_tidy_dagitty(x, include_root_nodes=TRUE,
                include_td_nodes=TRUE, include_networks=FALSE,
                seed=NULL, layout="nicely", ...)
```

Arguments

x	A DAG object created using the empty_dag function with nodes added to it using the + syntax. See ?empty_dag or ?node for more details. Supports DAGs with time-dependent nodes added using the node_td function. However, including such DAGs may result in cyclic causal structures, because time is not represented in the output matrix.
include_root_nodes	Whether to include root nodes in the output matrix. Should usually be kept at TRUE (default).
include_td_nodes	Whether to include time-dependent nodes added to the dag using the node_td function or not. When including these types of nodes, it is possible for the adjacency matrix to contain cycles, e.g. that it is not a classic DAG anymore, due to the matrix not representing the passage of time.
include_networks	Whether to include time-fixed networks added to the dag using the network function or not. Usually it does not make sense to include those, because they are not classical nodes.
seed	A numeric seed for reproducible layout generation.
layout	A layout available in ggraph. See create_layout for details.
...	Optional arguments passed to ggraph::create_layout().

Value

Returns a tidy_dagitty object.

Author(s)

Robin Denz

See Also

empty_dag, node, node_td, as.igraph.DAG

Examples

```
library(simDAG)

# some example DAG
dag <- empty_dag() +
  node("death", type="binomial", parents=c("age", "sex"), betas=c(1, 2),
       intercept=-10) +
  node("age", type="rnorm", mean=10, sd=2) +
  node("sex", parents="", type="rbernoulli", p=0.5) +
  node("smoking", parents=c("sex", "age"), type="binomial",
       betas=c(0.6, 0.2), intercept=-2)

if (requireNamespace("ggdag")) {
  library(ggdag)
  g <- ggdag::as_tidy_dagitty(dag)
  ggdag(g)
}
```

---

calibrate_dag *Calibrate a partially specified* DAG *to match target values*

Description

This function may be used to calibrate certain parameters in a partially specified DAG, given some global targets. For example, users may be interested in fixing marginal effects (such as the average causal treatment effect), rather than fixing the conditional treatment effects as represented by beta coefficients. See details.

Usage

```
calibrate_dag(lazy_dag, targets, params, n_sim_run=100000,
              n_sim_eval=1000000, n_reps_eval=1,
              seed=stats::runif(1, 0, 100),
              optimizer="auto", optim_method=NULL,
              optim_control=list(), objective_fun=NULL,
              ...)
```

Arguments

lazy_dag	An unevaluated, partially specified lazy_DAG object created using the lazy_dag function. See ?lazy_dag for more information on lazy DAGs and the associated param() syntax. See ?node for general information on how to define DAGs.

`targets` A `list` containing one `list` per desired target. These sub-lists need to contain at least two named elements: `target`, a numeric value specifying the exact desired value of the target and `eval_fun`, a function to evaluate the target value given simulated data with `n_sim_run` individuals or the current version of the `DAG` (if an additional `use_dag=TRUE` element is included). The `eval_fun` should only have one argument, which is the simulated data or DAG, and return exactly one numeric value relating to the target. Additionally, users may include a `scale` element, which should be a single numeric value specifying a scaling factor included during optimization. If not supplied, it defaults to 1. May be useful when targets are on widely different scales. This argument may specify any number of targets. See details, examples and the associated vignette.

`params` A named `list` specifying the free parameters of the partially specified DAG. The names must correspond to the parameter placeholders introduced by `param()` in the `lazy_dag`. Each parameter specification must contain a finite numeric start value and may optionally contain the following elements:

`range`: A numeric vector of length two specifying the lower and upper bounds for the parameter. If omitted, the parameter is treated as unbounded. Bounds are used by optimization methods that support them. For a single parameter and a single target, a finite range is required for automatic root finding using `uniroot`.

`scale`: A positive numeric scalar specifying the parameter scale used by the numerical optimizer. If omitted, the scale is determined automatically from the parameter range or starting value.

Parameter specifications may be supplied as named lists or named numeric vectors when all supplied elements are scalar. For example, `list(par1 = c(start = 1))` and `list(par1 = list(start = 1))` are both valid. If the DAG contains only one free parameter, its specification may be supplied directly, without an additional named list, e.g. `list(start = 0, range = c(-5, 5))`. In this case, the parameter name is inferred from the corresponding `param()` placeholder.

`n_sim_run` A single positive number, specifying the sample size that should be used when evaluating the `targets` during optimization. Higher numbers lead to more stable, reliable and precise estimates, but may take much longer to run. It often makes sense to use a smaller number first, to adjust the starting values with the (perhaps non-converged) results and to then re-run `calibrate_dag()` with a much higher value for more precise results.

`n_sim_eval` A single positive number, specifying the sample size used when evaluating the `targets` after optimization has finished. This is primarily done to estimate how well the optimization worked. To obtain reliable and precise results, it should either be a fairly large number with a single simulation repetition (e.g. `n_reps_eval=1`), or a moderate number with a large number of repetitions. The latter has the advantage that sampling distributions can be plotted and confidence intervals can be estimated, but it comes at the cost of higher computation time.

`n_reps_eval` A single positive number, specifying how many simulation repetitions should be performed when evaluating the final parameter values. See `n_sim_eval`.

`seed` A random number generator seed. If not `NULL`, this seed is used on every single evaluation of the underlying changing `DAG`, eliminating some random sampling variation. The only exception is the final evaluation using a sample of

size n_sim_eval, which instead uses seed + 1 (if possible), to make the evaluation independent of the optimization. Set to NULL to use an independent dataset during each optimization iteration instead.

optimizer — A single character string, specifying which optimizer should be used. Allowed values are: "uniroot" to use uniroot, "optim" to use optim, "optim2" to use the optim2 function from the **calibrar** package, "optimh" to use the optimh function from the **calibrar** package and "auto" (default), which is equal to "uniroot" whenever possible (e.g. whenever there is one target, one free parameter and defined bounds on that parameter) and equal to "optim" otherwise.

optim_method — A single character string, specifying which method should be used in the underlying optimizer function call. Ignored if "uniroot" is used as an optimizer.

optim_control — This argument is directly passed to the control argument of the respective optimiter function. Ignored if "uniroot" is used as an optimizer.

objective_fun — A custom function that should be used in the underlying optimizer function call to evaluate how close the current value of the targets are to the desired results. Any function that takes a numeric vector of (scaled) differences between the desired and estimated value and returns a scalar number that can be minimized may be used. Ignored if "uniroot" is used as an optimizer.

... — Further arguments passed to the optimizer and the sim_n_datasets function, which is used internally to generate one dataset (using n_repeats = 1). All named arguments are first passed through the optimizer.

Details

We recommend consulting the associated vignette for a more detailed explanation, including multiple worked examples, available at vignette(topic="v_calibrate_dag", package="simDAG").

What it does:

This function tries to help find suitable conditional parameters, when the goal is to fix marginal ones. Instead of defining all conditional parameters directly, users may specify arbitrary (potentially marginal) global parameters instead and to leave the conditional parameters partially unspecified. It then tries to *calibrate* the unspecified parameters, so that the global targets are met. The resulting DAG can then be used to simulate the data with the desired global target properties using sim_from_dag, sim_discrete_time or sim_discrete_event.

How it works:

Instead of defining a fully specified DAG, users should define only a partially specified DAG, leaving one or more parameters unspecified. This should be done using the lazy_dag function (see ?lazy_dag for more information). Afterwards, users can define an arbitrary amount of targets through custom targets functions. These functions should simply calculate the current value of the desired property. For example, if the data should have a 50% prevalence in the binary variable Y, a suitable target function would simply be function(data) {mean(data$Y)} with an associated target = 0.5. Additionally, users should give rough ranges in which the unspecified parameters may lie.

Internally, root finding (if 1 target and 1 unspecified parameter, with defined ranges) or numerical optimization (all other cases) is then used to find suitable values for the unspecified parameters that satisfy the target properties defined by the user. In a nutshell, the function simply simulates a large dataset from a given parameter combination, checks the resulting target properties and tries again.

This idea, although applied in a less general setting, is also described by Austin and Stafford (2008) and Austin (2023).

To inspect the found parameter values and how well those produce the desired targets, users may use the associated `summary()` method (see examples). If `n_reps_eval > 1`, the associated `plot.calibrated_DAG` method may also be used. To simulate data directly using the resulting parameters, users may use the `dag` object returned in the output.

Currently supported features:

Users may use custom functions to define the target properties, so there are very few limits to what can be specified. Additionally, any parameter in a valid expression that defines a `DAG` may be left unspecified. Furthermore, there is no limit put on either the number of target properties (denoted m) or the number of unspecified parameters (denoted p), although some care needs to be exercised. If $p > m$, there may be multiple different solutions, which is usually not problematic, but should be noted. If $m > p$, there likely aren't any solutions, unless some `targets` are highly correlated. DAGs of any kind (using discrete-time varying covariates, continuous-time varying covariates, network dependencies, ...) may be used here.

Limitations:

There is unfortunately no guarantee that a solution for any given specification exist. Even worse, there is no guarantee that if a solution exists, that this function will find it. It is, however, still much better than manually searching for a solution. Furthermore, users should keep in mind that even if a solution exists and the numeric optimization is able to converge to it, that the results are only approximate. In most cases, this should be completely fine for Monte-Carlo simulation studies. However, in some edge cases, if exact simulation specifications of marginal parameters are required, this may not be sufficient.

Convergence:

Convergence may be tricky with complicated `DAG`s, many `params` or many `targets` (or both). Ultimately, the same techniques that work for any numerical optimization may be used to obtain convergence, e.g. tuning the `optim_control` arguments, re-running the function with starting values from a previous run etc. The only thing specific to this function that has a large effect on convergence is the `n_sim_run` argument. Better convergence may be obtained with larger values, at the price of a longer computation time.

Value

Returns a `calibrated_DAG` object. This object is basically just a `list`, containing the following objects:

- `inputs`: A list containing all user inputs.
- `dag`: A fully specified `DAG`, using the best sets of parameter values found to satisfy the user targets. This `DAG` may be directly used for simulation purposes.
- `est_params`: The final parameter values after optimization.
- `target_eval`: A final estimate of the `targets`, evaluated on the final `DAG`. If the function call was successful, these should be very close to the desired target parameters. If `n_reps_eval > 1`, these correspond to the average over all simulation repetitions.
- `d_target_eval`: If `n_reps_eval > 1` was used, this object includes three `data.tables`, including the estimated targets on each simulation run, the differences to the desired values and the scaled differences.

- optim_obj: If optim, optim2 or optimh was used, the resulting object from that function call.
- uniroot: If uniroot was used, the resulting object from that function call.

Author(s)

Robin Denz

References

Denz, Robin and Nina Timmesfeld (2026). "Simulating Complex Cross-Sectional and Longitudinal Data using the simDAG R Package". Journal of Statistical Software 116 (2), doi: 10.18637/jss.v116.i02.

Austin, Peter C. 2023. "The Iterative Bisection Procedure: A Useful Tool for Determining Parameter Values in Data-Generating Processes in Monte Carlo Simulations."" BMC Medical Research Methodology 23 (45): 1–10.

Austin, Peter C., and James Stafford. 2008. "The Performance of Two Data-Generation Processes for Data with Specified Marginal Treatment Odds Ratios."" Communications in Statistics: Simulation and Computation 37 (6): 1039–51.

Examples

```
library(simDAG)
library(data.table)

# Example 1: Calibrate a single parameter to match a target prevalence.
# A finite parameter range allows automatic root finding.

ldag <- lazy_dag(
  empty_dag() +
    node("X", type="rnorm") +
    node("A", type="binomial", formula= ~ param(intercept_A) + X*0.5)
)

targets <- list(
  prevalence_A = list(
    target = 0.4,
    eval_fun = function(data) mean(data$A)
  )
)

# For a single parameter, its specification can be supplied directly.
params <- list(
  start = 0,
  range = c(-5, 5)
)

fit <- calibrate_dag(
  lazy_dag = ldag,
  targets = targets,
  params = params,
  n_sim_run = 2000,
```

```
  n_sim_eval = 5000,
  seed = 123
)
summary(fit)

# use final DAG to simulate data
data <- sim_from_dag(fit$dag, n_sim=10)
head(data)

# Example 2: Jointly calibrate treatment and outcome prevalence.
# The DAG contains two free parameters and two targets.
# Parameter ranges are optional.

ldag <- lazy_dag(
  empty_dag() +
    node("X", type="rnorm") +
    node("A", type="binomial",
         formula= ~ param(intercept_A) + X * 0.5) +
    node("Y", type="binomial",
         formula= ~ param(intercept_Y) + A * 0.7 + X * 0.4)
)

targets <- list(
  prevalence_A = list(
    target = 0.4,
    eval_fun = function(data) mean(data$A)
  ),
  prevalence_Y = list(
    target = 0.3,
    eval_fun = function(data) mean(data$Y)
  )
)

params <- list(
  intercept_A = list(start = 0),
  intercept_Y = list(start = 0)
)

fit <- calibrate_dag(
  lazy_dag = ldag,
  targets = targets,
  params = params,
  n_sim_run = 2000,
  n_sim_eval = 5000,
  seed = 123,
  optim_control = list(
    maxit = 200,
    trace = 0,
    ndeps = c(0.05, 0.05),
    factr = 100
  )
)
summary(fit)
```

```
# use final DAG to simulate data
data <- sim_from_dag(fit$dag, n_sim=10)
head(data)
```

---

dag2matrix *Obtain a Adjacency Matrix from a* DAG *object*

Description

The sim_from_dag function requires the user to specify the causal relationships inside a DAG object containing node information. This function takes this object as input and outputs the underlying adjacency matrix. This can be useful to plot the theoretical DAG or to check if the nodes have been specified correctly.

Usage

```
dag2matrix(dag, include_root_nodes=TRUE, include_td_nodes=FALSE,
           include_networks=FALSE)
```

Arguments

dag	A DAG object created using the empty_dag function with nodes added to it using the + syntax. See ?empty_dag or ?node for more details. Supports DAGs with time-dependent nodes added using the node_td function. However, including such DAGs may result in cyclic causal structures, because time is not represented in the output matrix.
include_root_nodes	Whether to include root nodes in the output matrix. Should usually be kept at TRUE (default).
include_td_nodes	Whether to include time-dependent nodes added to the dag using the node_td function or not. When including these types of nodes, it is possible for the adjacency matrix to contain cycles, e.g. that it is not a classic DAG anymore, due to the matrix not representing the passage of time.
include_networks	Whether to include time-fixed networks added to the dag using the network function or not. Usually it does not make sense to include those, because they are not classical nodes. This is mostly used internally to ensure that the generation of nodes and networks is processed in the right order.

Details

An adjacency matrix is simply a square matrix in which each node has one column and one row associated with it. For example, if the node A has a causal effect on node B, the matrix will contain 1 in the spot matrix["A", "B"].

If a time-varying node is also defined as a time-fixed node, the parents of both parts will be pooled when creating the output matrix.

Value

Returns a numeric square matrix with one row and one column per used node in dag.

Author(s)

Robin Denz

See Also

empty_dag, node, node_td

Examples

```
library(simDAG)

# some example DAG
dag <- empty_dag() +
  node("death", type="binomial", parents=c("age", "sex"), betas=c(1, 2),
       intercept=-10) +
  node("age", type="rnorm", mean=10, sd=2) +
  node("sex", parents="", type="rbernoulli", p=0.5) +
  node("smoking", parents=c("sex", "age"), type="binomial",
       betas=c(0.6, 0.2), intercept=-2)

# get adjacency matrix
dag2matrix(dag)

# get adjacency matrix using only the child nodes
dag2matrix(dag, include_root_nodes=FALSE)

## adding time-varying nodes
dag <- dag +
  node_td("disease", type="time_to_event", parents=c("age", "smoking"),
          prob_fun=0.01) +
  node_td("cve", type="time_to_event", parents=c("age", "sex", "smoking",
                                                 "disease"),
          prob_fun=0.001, event_duration=Inf)

# get adjacency matrix including all nodes
dag2matrix(dag, include_td_nodes=TRUE)

# get adjacency matrix including only time-constant nodes
dag2matrix(dag, include_td_nodes=FALSE)

# get adjacency matrix using only the child nodes
dag2matrix(dag, include_root_nodes=FALSE)
```

dag_from_data *Fills a partially specified* DAG *object with parameters estimated from reference data*

Description

Given a partially specified DAG object, where only the name, type and the parents are specified plus a data.frame containing realizations of these nodes, return a fully specified DAG (with beta-coefficients, intercepts, errors, ...). The returned DAG can be used directly to simulate data with the sim_from_dag function.

Usage

```
dag_from_data(dag, data, return_models=FALSE, na.rm=FALSE)
```

Arguments

dag	A partially specified DAG object created using the empty_dag and node functions. See ?node for a more detailed description on how to do this. All nodes need to contain information about their name, type and parents. All other attributes will be added (or overwritten if already in there) when using this function. Currently does not support DAGs with time-dependent nodes added with the node_td function.
data	A data.frame or data.table used to obtain the parameters needed in the DAG object. It needs to contain a column for every node specified in the dag argument.
return_models	Whether to return a list of all models that were fit to estimate the information for all child nodes (elements in dag where the parents argument is not NULL).
na.rm	Whether to remove missing values or not.

Details

How it works:

It can be cumbersome to specify all the node information needed for the simulation, especially when there are a lot of nodes to consider. Additionally, if data is available, it is natural to fit appropriate models to the data to get an empirical estimate of the node information for the simulation. This function automates this process. If the user has a reasonable DAG and knows the node types, this is a very fast way to generate synthetic data that corresponds well to the empirical data.

All the user has to do is create a minimal DAG object including only information on the parents, the name and the node type. For root nodes, the required distribution parameters are extracted from the data. For child nodes, regression models corresponding to the specified type are fit to the data using the parents as independent covariates and the name as dependent variable. All required information is extracted from these models and added to the respective node. The output contains a fully specified DAG object which can then be used directly in the sim_from_dag function. It may also include a list containing the fitted models for further inspection, if return_models=TRUE.

Supported root node types:

Currently, the following root node types are supported:

- "rnorm": Estimates parameters of a normal distribution.
- "rbernoulli": Estimates the p parameter of a Bernoulli distribution.
- "rcategorical": Estimates the class probabilities in a categorical distribution.

Other types need to be implemented by the user.

Supported child node types:

Currently, the following child node types are supported:

- "gaussian": Estimates parameters for a node of type "gaussian".
- "binomial": Estimates parameters for a node of type "binomial".
- "poisson": Estimates parameters for a node of type "poisson".
- "negative_binomial": Estimates parameters for a node of type "negative_binomial".
- "conditional_prob": Estimates parameters for a node of type "conditional_prob".

Other types need to be implemented by the user.

Support for custom nodes:

The sim_from_dag function supports custom node functions, as described in the associated vignette. It is impossible for us to directly support these custom types in this function directly. However, the user can extend this function easily to accommodate any of his/her custom types. Similar to defining a custom node type, the user simply has to write a function that returns a correctly specified node.DAG object, given the named arguments name, parents, data, return_model and na.rm. The first two arguments should simply be added directly to the output. Additionally, the user should include both the node type itself as a string under the name type_str and the full custom node function under type_fun in the output. The data should be used inside your function to fit a model or obtain the required parameters in some other way. The return_model argument should control whether the model should be added to the output (in a named argument called model). The function name should be paste0("gen_node_", YOURTYPE). The na.rm argument should be used to control whether missing values should be removed or not. An examples is given below.

Interactions & cubic terms:

This function currently does not support the usage of interaction effects or non-linear terms (such as using A ~ B + I(B^2) as a formula). Instead, it will be assumed that all values in parents have a linear effect on the respective node. For example, using parents=c("A", "B") for a node named "C" will use the formula C ~ A + B. If other behavior is desired, users need to integrate this into their own custom function as described above.

Value

A list of length two containing the new fully specified DAG object named dag and a list of the fitted models (if return_models=TRUE) in the object named models.

Author(s)

Robin Denz

Examples

```
library(simDAG)

set.seed(457456)

# get some example data from a known DAG
dag <- empty_dag() +
  node("death", type="binomial", parents=c("age", "sex"), betas=c(1, 2),
       intercept=-10) +
  node("age", type="rnorm", mean=10, sd=2) +
  node("sex", parents="", type="rbernoulli", p=0.5) +
  node("smoking", parents=c("sex", "age"), type="binomial",
       betas=c(0.6, 0.2), intercept=-2)

data <- sim_from_dag(dag=dag, n_sim=1000)

# suppose we only know the causal structure and the node type:
dag <- empty_dag() +
  node("death", type="binomial", parents=c("age", "sex")) +
  node("age", type="rnorm") +
  node("sex", type="rbernoulli") +
  node("smoking", type="binomial", parents=c("sex", "age"))

# get parameter estimates from data
dag_full <- dag_from_data(dag=dag, data=data)

# can now be used to simulate data
data2 <- sim_from_dag(dag=dag_full$dag, n_sim=100)
```

---

do *Pearls do-operator for* DAG *objects*

Description

This function can be used to set one or more nodes in a given DAG object to a specific value, which corresponds to an intervention on a DAG as defined by the do-operator introduced by Judea Pearl.

Usage

```
do(dag, names, values)
```

Arguments

dag	A DAG object created using the empty_dag and node functions. See ?node for more information on how to specify a DAG.
names	A character string specifying names of nodes in the dag object. The value of these nodes will be set to the corresponding value specified in the values argument. If the node is not already defined in dag, a new one will be added without warning.

values — A vector or list of any values or node / node_td definitions. These nodes defined with the names argument will be set to those values or the new node definitions.

Details

Internally this function simply removes the old node definition of all nodes in names and replaces it with a new node definition that defines the node as a constant value, irrespective of the original definition, if the corresponding entry in values is a single value. If it is a new node or node_td instead, the new definition replaces the old one. The same effect can be created by directly specifying the DAG in this way from the start (see examples).

This function does not alter the original DAG in place. Instead, it returns a modified version of the DAG. In other words, using only do(dag, names="A", values=3) will not change the dag object.

Value

Returns a DAG object with updated node definitions.

Author(s)

Robin Denz

References

Judea Pearl (2009). Causality: Models, Reasoning and Inference. 2nd ed. Cambridge: Cambridge University Press

Examples

```
library(simDAG)

# define some initial DAG
dag <- empty_dag() +
  node("death", "binomial", c("age", "sex"), betas=c(1, 2), intercept=-10) +
  node("age", type="rnorm", mean=10, sd=2) +
  node("sex", parents="", type="rbernoulli", p=0.5) +
  node("smoking", parents=c("sex", "age"), type="binomial",
       betas=c(0.6, 0.2), intercept=-2)

# return new DAG with do(smoking = TRUE)
dag2 <- do(dag, names="smoking", values=TRUE)

# which is equivalent to
dag2 <- empty_dag() +
  node("death", "binomial", c("age", "sex"), betas=c(1, 2), intercept=-10) +
  node("age", type="rnorm", mean=10, sd=2) +
  node("sex", parents="", type="rbernoulli", p=0.5) +
  node("smoking", type="rconstant", constant=TRUE)

# use do() on multiple variables: do(smoking = TRUE, sex = FALSE)
dag2 <- do(dag, names=c("smoking", "sex"), values=list(TRUE, FALSE))
```

```
## set node in DAG to a completely new definition
dag <- empty_dag() +
  node("death", "binomial", c("age", "sex"), betas=c(1, 2), intercept=-10) +
  node("age", type="rnorm", mean=10, sd=2) +
  node("sex", parents="", type="rbernoulli", p=0.5) +
  node("smoking", parents=c("sex", "age"), type="binomial",
       betas=c(0.6, 0.2), intercept=-2)

dag2 <- do(dag, names="smoking", values=list(
  node(".", type="poisson", formula= ~ -2 + age*0.2 + sex*1.2)
))
```

`empty_dag` *Initialize an empty* DAG *object*

Description

This function should be used in conjunction with multiple calls to `node` or `node_td` to create a DAG object, which can then be used to simulate data using the `sim_from_dag` and `sim_discrete_time` functions.

Usage

```
empty_dag()
```

Details

Note that this function is only used to initialize an empty DAG object. Actual information about the respective nodes have to be added using the `node` function or the `node_td` function. The documentation page of that function contains more information on how to correctly do this.

Value

Returns an empty DAG object.

Author(s)

Robin Denz

Examples

```
library(simDAG)

# just an empty DAG
empty_dag()

# adding a node to it
empty_dag() + node("age", type="rnorm", mean=20, sd=5)
```

lazy_dag *Define a partial* DAG *object without evaluating it*

Description

This function should be used as a wrapper around an expression that aims to generate a DAG object, but with some parameters left unspecified. Those parameters should be defined using the param() syntax, as shown below. The DAG is then not instantly evaluated. Instead, it will only be evaluated later using a set_lazy_dag call, which then sets the still unspecified parameters. This may be useful to create flexible functions that generate DAG objects during simulations, and is also used in calibrate_dag.

Usage

```
lazy_dag(expr)
```

Arguments

expr	An expression that would usually create a DAG object. This should include one empty_dag call and one or more added node or node_td functions. Importantly, at least one parameter of one node should be left unspecified, using the param() syntax. See ?node for more information on how to specify a regular DAG. See details on how to use the param() syntax.

Details

It is sometimes convenient to define a DAG object, but to leave some parameters of it flexible. For example, if we wanted to conduct a Monte-Carlo simulation study with different values of an interaction coefficient in a formula of a node, we need different DAG definitions for each scenario. Instead of copying and pasting nearly the same code multiple times, this function allows users to define a single partially specified DAG that leaves one or multiple parameters of a DAG definition unspecified.

This works the same way a usual DAG construction does. The only two differences are (1) that the whole expression (expr) that creates the DAG should be wrapped in a lazy_dag() function call and (2) that parameters that should remain flexible are wrapped in a param() function. For example, if we wish to leave the intercept in a formula flexible, we could use param(intercept). There is no limit on the amount of flexible parameters that can be specified. They also do not have to be in a formula, but can be passed to any argument of any node call. The only requirements are that the names used inside the param() call are unique, so that they can later be unambiguously identified.

Once a lazy_DAG is created, users may create real DAG objects from it using the set_lazy_dag function, which only requires the lazy_DAG and a vector of params as input. See ?set_lazy_dag for more information. Before running set_lazy_dag(), it is not possible to use these objects to simulate any data, because they are not fully specified or even executed yet. Note also that param() is just a piece of syntax. The param() function itself does not actually exist in the **simDAG** package.

Value

Returns a lazy_DAG object.

Author(s)

Robin Denz

Examples

```
library(simDAG)

# define a lazy DAG with one unspecified parameter
ldag <- lazy_dag(
  empty_dag() +
    node("A", type="rnorm", mean=1, sd=param(SD))
)

# get a real DAG object with SD = 1 and another with SD = 2
dag1 <- set_lazy_dag(ldag, params=c(SD=1))
dag2 <- set_lazy_dag(ldag, params=c(SD=2))

# define some lazy DAG, leaving the intercept of "death" and one beta
# coefficient of "smoking" unspecified
ldag <- lazy_dag(
  empty_dag() +
    node("age", type="rnorm", mean=10, sd=2) +
    node("sex", parents="", type="rbernoulli", p=0.5) +
    node("smoking", type="binomial",
         formula= ~ -2 + age*param(Sbeta) + sex*0.2) +
    node("death", type="binomial",
         formula= ~ param(Dintercept) + age*1 + sex*2)
)

# get a real DAG object from this with set values
dag3 <- set_lazy_dag(ldag, params=c(Sbeta=2.1, Dintercept=-10))
```

---

long2start_stop *Transform a* data.table *in the long-format to a* data.table *in the start-stop format*

Description

This function transforms a data.table in the long-format (one row per person per time point) to a data.table in the start-stop format (one row per person-specific period in which no variables changed).

Usage

```
long2start_stop(data, id, time, varying, overlap=FALSE,
                check_inputs=TRUE)
```

Arguments

data	A data.table or an object that can be coerced to a data.table (such as a data.frame) including data in the long-format.
id	A single character string specifying a unique person identifier included in in data.
time	A single character string specifying a time variable included in in data coded as integers starting from 1.
varying	A character vector specifying names of variables included in in data that may change over time.
overlap	Specifies whether the intervals should overlap or not. If TRUE, the "stop" column is simply increased by one, as compared to the output when overlap=FALSE. This means that changes for a given t are recorded at the start of the next interval, but the previous interval ends on that same day.
check_inputs	Whether to check if the user input is correct or not. Can be turned off by setting it to FALSE to save computation time.

Details

This function relies on data.table syntax to make the data transformation as RAM efficient and fast as possible.

Value

Returns a data.table containing the columns .id (the unique person identifier), .time (an integer variable encoding the time) and all other variables included in the input data in the long format.

Author(s)

Robin Denz

Examples

```
library(simDAG)
library(data.table)

# generate example data in long format
long <- data.table(.id=rep(seq_len(10), each=5),
                   .time=rep(seq_len(5), 10),
                   A=c(rep(FALSE, 43), TRUE, TRUE, rep(FALSE, 3), TRUE,
                           TRUE),
                   B=FALSE)
setkey(long, .id, .time)

# transform to start-stop format
long2start_stop(data=long, id=".id", time=".time", varying=c("A", "B"))
```

matrix2dag *Obtain a* DAG *object from a Adjacency Matrix and a List of Node Types*

Description

The sim_from_dag function requires the user to specify the causal relationships inside a DAG object containing node information. This function creates such an object using a adjacency matrix and a list of node types. The resulting DAG will be only partially specified, which may be useful for the dag_from_data function.

Usage

```
matrix2dag(mat, type)
```

Arguments

mat	A p x p adjacency matrix where p is the number of variables. The matrix should be filled with zeros. Only places where the variable specified by the row has a direct causal effect on the variable specified by the column should be 1. Both the columns and the rows should be named with the corresponding variable names.
type	A named list with one entry for each variable in mat, specifying the type of the corresponding node. See node for available node types.

Details

An adjacency matrix is simply a square matrix in which each node has one column and one row associated with it. For example, if the node A has a causal effect on node B, the matrix will contain 1 in the spot matrix["A", "B"]. This function uses this kind of matrix and additional information about the node type to create a DAG object. The resulting DAG cannot be used in the sim_from_dag function directly, because it will not contain the necessary parameters such as beta-coefficients or intercepts etc. It may, however, be passed directly to the dag_from_data function. This is pretty much it's only valid use-case. If the goal is to to specify a full DAG manually, the user should use the empty_dag function in conjunction with node calls instead, as described in the respective documentation pages and the vignettes.

The output will never contain time-dependent nodes. If this is necessary, the user needs to manually define the DAG.

Value

Returns a partially specified DAG object.

Author(s)

Robin Denz

See Also

empty_dag, node, node_td, dag_from_data

Examples

```
library(simDAG)

# simple example adjacency matrix
mat <- matrix(c(0, 0, 1, 0, 0, 1, 0, 0, 0), ncol=3, byrow=TRUE)
colnames(mat) <- c("age", "sex", "death")
rownames(mat) <- c("age", "sex", "death")

type <- list(age="rnorm", sex="rbernoulli", death="binomial")

matrix2dag(mat=mat, type=type)
```

---

net *Specify Network Dependencies in a* DAG

Description

This function may be used in the `formula` of nodes in which the value of the observation of one individual are dependent on its' neighbors in a defined static `network` or dynamic `network_td`. Given the network and a previously generated variable, `net()` aggregates data of the neighbors according to an arbitrary function under the hood. The resulting variable can then be used directly in a `formula`.

Usage

```
net(expr, net=NULL, mode="all", order=1,
    mindist=0, na=NA)
```

Arguments

`expr`	Any R expression, usually containing one or more previously generated variables, that returns one numeric value given a vector, such as `sum(variable)` or `mean(variable)`.
`net`	A single character string specifying the name of the network that should be used to define the neighbors of an observation. If only one network is present in the `DAG`, this argument can be omitted. The single added network is then used by default. If multiple networks are present and this argument is not defined, an error will be produced.
`mode`	A single character, specifying how to use the direction of the edges if a directed network is supplied (ignored otherwise). If `"all"`, the direction of the edges is ignored and both incoming and outgoing edges are used to define the neighbors of each individual. If `"out"`, only the individuals who i (the observation row) is pointing to are used as neighbors and if `"in"` only the individuals who point to i are being used as neighbors.
`order`	A single integer giving the order of the neighborhood. If `order=1` (default), only the vertices that are directly connected to vertex i are considered its neighbors. If `order=2`, all vertices connected to those neighbors are also considered the neighbors of vertex i and so on.

`mindist` A single integer >= 0, specifying the minimum distance the neighbors needs to have to an observation to be considered neighbors. Only makes sense with `order > 1`.

`na` A single value assigned to the variable if `expr` could not be computed. This can happen due to the nature of the expression (e.g. `NA` being returned directly after evaluating the expression for some reason), or when an observation does not have any neighbors in a network.

Details

How it works:

Internally the following procedure is used whenever a `net()` function call is included in a `formula` of a `node` (regardless of whether time-fixed or time-dependent). First, the associated network (defined using the `net` argument) is used to identify the neighbors of each observation. Every vertex that is directly connected to an observation is considered its' neighbor. The parent variable(s) specified in the `net()` call are then aggregated over these neighbors using the given `expr`. A simple example: consider observation `1` with four neighbors named `2, 5, 8` and `10`. The `formula` contains the following `net()` call: `net(sum(infected))`. The value of the `infected` variable is 0, 0, 1, 1 for persons `2, 5, 8` and `10` respectively. These values are then summed up to result in a value of 2 for person `1`. The same is done for every person in the simulated data. The resulting variable is then used as-is in the simulation.

Supported inputs:

Any function that returns a single (usually numeric) value, given the neighbors' values can be used. It is therefore also possible to make the simulation dependent on specific neighbors only. For example, using `infected[1]` instead of `sum(infected)` would return a value of 0 for observation 1 in the above example, because person `2` is the first neighbor and has a value of 0. Note that the internally used variable named `..neighbor..` includes the ids of the neighbors. The entire `expr` is evaluated in a **data.table** call of the form: `data[, .(variable = eval(expr)), by=id]`, making it also possible to use any **data.table** syntax such as `.N` (which would return the number of neighbors a person has).

Specifying parents:

Whenever a `net()` call is used in a `formula`, we recommend specifying the `parents` argument of the node as well. The reason for this recommendation is, that it is sometimes difficult to identify which variables are used in `net()` calls, depending on the `expr`. This may cause issues if a `DAG` is not specified in a topologically sorted manner and users rely on the `sort_dag` argument of `sim_from_dag` to re-order the variables. Specifying the `parents` ensures that this issue cannot occur.

A small warning:

Note that it never really makes sense to use this function outside of a `formula` argument: if you look at its source code you will realize that it does not actually do anything, except returning its input. It is only a piece of syntax for the `formula` interface. Please consult the `network` documentation page or the associated vignette for more information.

Value

"Returns" a vector of length `n_sim` when used properly in a `sim_from_dag` or `sim_discrete_time` call. Returns a list of its input when used outside `formula`.

Author(s)

Robin Denz

Examples

```
library(igraph)
library(data.table)
library(simDAG)

# define a random network for illustration, with 10 vertices
set.seed(234)
g <- igraph::sample_smallworld(1, 10, 2, 0.5)

# a simple dag containing only two time-constant variables and the network
dag <- empty_dag() +
  node("A", type="rnorm", mean=0, sd=1) +
  node("B", type="rbernoulli", p=0.5) +
  network("Net1", net=g)

# using the mean of A of each observations neighbor in a linear model
dag2 <- dag +
  node("Y", type="gaussian", formula= ~ -2 + net(mean(A))*4, error=1)

# using an indicator of whether any of an observations neighbors has
# a 1 in B in a linear model
dag3 <- dag +
  node("Y", type="gaussian", formula= ~ 1.5 + net(as.numeric(any(B==1)))*3,
       error=1.2)
```

`network` *Create a network object for a* `DAG`

Description

These functions (in conjunction with the `empty_dag` and `node` functions) allow users to create `DAG` objects with one or more, possibly time-varying, network structures linking individual observations to each other. This makes it possible to simulate data with complex network-based dependencies among observations using the `sim_from_dag` function or the `sim_discrete_time` function.

Usage

```
network(name, net, parents=NULL, ...)

network_td(name, net, parents=NULL, create_at_t0=TRUE, ...)
```

Arguments

name	A single character string, specifying the name of the network. Contrary to the node function, multiple values are not allowed, because defining the same network multiple times does not make sense.
net	For network(), two kinds of inputs are allowed. The first is an igraph object containing one vertex per observation (e.g. n_sim vertices) that should be generated when later calling sim_from_dag or sim_discrete_time. The second is a function that generates such an object, given a named argument called n_sim and any number of further named arguments. For network_td(), only the latter kind of input is allowed. The vertices in the network defined by this variable should not be named. Instead, the vertex with index 1 represents row 1 of the generated data, the vertex with index 2 represents the second row and so on. Further information is given in the details section.
parents	A character vector of names, specifying the parents of the network or NULL (default). Similar to general nodes, specifying the parents allows users to generate a network as a function of the values of the parents, whenever net is a function. If NULL, it is assumed that the network is generated independently of the data (or already passed as igraph object). For convenience, it is also allowed to set parents="" to indicate that the node has no parents.
create_at_t0	Either TRUE or FALSE, specifying whether the network should be generated at time 0 in discrete-time simulations (e.g. when other time-independent nodes and networks are generated) or only after the creation of data time 0. Defaults to TRUE.
...	Optional further named arguments passed to net if it is a function.

Details

What does it mean to add a network to a DAG***?***

When using only node or node_td to define a DAG, all observations are usually generated independently from each other (if not explicitly done otherwise using a custom node function). This reflects the classic i.i.d. assumption that is frequently used everywhere. For some data generation processes, however, this assumption is insufficient. The spread of an infectious disease is a classic example.

The network() function allows users to relax this assumption, by making it possible to define one or more networks that can then be added to DAG objects using the + syntax. These networks should contain a single vertex for each observation that should be generated, placing each row of the dataset into one place in the network. Through the use of the net function it is then possible to define new nodes as a function of the neighbors of an observation, where the neighbors of a vertex are defined as any other vertex that is directly connected to this node. For example, one could use this capability to use the mean age of an observations neighbors in a regression model, or use the number of infected neighbors to model the probability of infection. By combining this network-simulation approach with the already extensive simulation capabilities of DAG based simulations, almost any DGP can be modelled. This approach is described more rigorously in the excellent paper given by Sofrygin et al. (2017).

Supported network types:

Users may add any number of networks to a DAG object, making it possible to embed individuals in multiple distinct networks at the same time. These networks can then be used simultaneously to define a single or multiple (possibly time-varying) nodes, using multiple net function calls in the respective formula arguments. It is also possible to define time-varying or dynamic networks that change over time, possibly as a function of the generated data, simulation time or previous states of the network. Examples are given below and in the associated vignette.

The package directly supports un-directed and directed, un-weighted and weighted networks. It also supports different definitions of what the neighbors of an observation are. Note, however, that only networks which include exactly one vertex per observation are supported.

Weighted Networks:

It is possible to supply weighted networks to network(). The weights are then also stored and available to the user when using the net function through the internal ..weight.. variable. For example, if a weighted network was supplied, the following would be valid syntax: net(weighted.mean(A, ..weight..)) (assuming that A is a previously defined variable). Note that the ..weight.. must be used explicitly, otherwise the weights are ignored.

Directed Networks:

Supplying directed networks is also possible. If this is done, users usually need to specify the mode argument of the net function when defining the formula arguments. This argument allows users to define different kinds of neighborhoods for each observation, based on the direction of the edges.

Order of Generation:

Generally, all networks are created in the order in which they were added to the DAG, unless sort_dag or tx_nodes_order are changed in sim_from_dag or sim_discrete_time respectively. The only exception is that all networks created using the network() function are created *after* all other root nodes have already been generated.

Computational considerations:

Including net() terms in a node might significantly increase the amount of RAM used and the required computation time, especially with very large networks and / or large values of n_sim and / or max_t (the latter is only relevant in discrete-time simulations using sim_discrete_time). The reason for this is that each time a node is generated or updated over time, the mapping of individuals to their neighbors' values plus the subsequent aggregation has to be performed, which requires merge() calls etc. Usually this should not be a problem, but it might be for some large discrete-time simulations. If the same net call is used in multiple nodes it can be beneficial to put it into an extra node call and safe it to avoid re-calculating the same thing over and over again (see examples).

Further information:

For a theoretical treatment, please consult the paper by Sofrygin et al. (2017), who also describe their slightly different implementation of this method in the **simcausal** package. More information on how to specify network-based dependencies in a DAG (using **simDAG**) after adding a network, please consult the net documentation page or the associated vignette.

Value

Returns a DAG.network object which can be added to a DAG object directly.

Author(s)

Robin Denz

References

Sofrygin, Oleg, Romain Neugebauer and Mark J. van der Laan (2017). Conducting Simulations in Causal Inference with Networks-Based Structural Equation Models. arXiv preprint, doi: 10.48550/arXiv.1705.10376

Examples

```
library(igraph)
library(data.table)
library(simDAG)

set.seed(2368)

# generate random undirected / unweighted networks as examples
g1 <- igraph::sample_gnm(n=20, m=30)
g2 <- igraph::sample_gnm(n=20, m=30)

# adding a single network to a DAG, with Y being dependent on
# the mean value of A of its neighbors
dag <- empty_dag() +
  network("Net1", net=g1) +
  node("A", type="rnorm") +
  node("Y", type="gaussian", formula= ~ -2 + net(mean(A))*1.3, error=1.5)

# NOTE: because we supplied the network of size 20 directly, we can only
#       use n_sim=20 here
data <- sim_from_dag(dag, n_sim=20)

# using multiple networks, with Y being differently dependent on
# the mean value of A of its neighbors in both networks
dag <- empty_dag() +
  network("Net1", net=g1) +
  network("Net2", net=g2) +
  node("A", type="rnorm") +
  node("Y", type="gaussian", formula= ~ -2 + net(mean(A), net="Net1")*1.3 +
        net(mean(A), net="Net2")*-2, error=1.5)

# using a function to add networks, to allow any value of 'n_sim' later

# exemplary function that returns a random network of size 'n_sim'
gen_network <- function(n_sim) {
  igraph::sample_gnm(n=n_sim, m=30)
}

# same as first example, but using the function as input
dag <- empty_dag() +
  network("Net1", net=gen_network) +
  node("A", type="rnorm") +
  node("Y", type="gaussian", formula= ~ -2 + net(mean(A))*1.3, error=1.5)
```

```
data <- sim_from_dag(dag, n_sim=25)
```

---

node *Create a node object for a* DAG

Description

These functions should be used in conjunction with the empty_dag function to create DAG objects, which can then be used to simulate data using the sim_from_dag, sim_discrete_time or sim_discrete_event functions.

Usage

```
node(name, type, parents=NULL, formula=NULL, ...)

node_td(name, type, parents=NULL, formula=NULL, ...)
```

Arguments

name	A character vector with at least one entry specifying the name of the node. If a character vector containing multiple different names is supplied, one separate node will be created for each name. These nodes are completely independent, but have the exact same node definition as supplied by the user. If only a single character string is provided, only one node is generated.
type	A single character string specifying the type of the node. Depending on whether the node is a root node, a child node or a time-dependent node different node types are allowed. See details. Alternatively, a suitable function may be passed directly to this argument.
parents	A character vector of names, specifying the parents of the node or NULL (default). If NULL, the node is treated as a root node. For convenience it is also allowed to set parents="" to indicate that the node is a root node.
formula	An optional formula object to describe how the node should be generated or NULL (default). If supplied it should start with ~, having nothing else on the left hand side. The right hand side should define the entire structural equation, including the betas and intercepts. It may contain any valid formula syntax, such as ~ -2 + A*3 + B*4 or ~ -2 + A*3 + B*4 + I(A^2)*0.3 + A:B*1.1, allowing arbitrary non-linear effects, arbitrary interactions and multiple coefficients for categorical variables. Additionally, for some node types, random effects and random slopes are supported. If this argument is defined, there is no need to define the betas and intercept argument. The parents argument should still be specified whenever a categorical variable is used in the formula. This argument is supported for build-in nodes of type "binomial", "gaussian", "poisson", "negative_binomial", "cox", "aftreg", "ahreg", "ehreg", "poreg", "ypreg", "time_to_event" and "next_time" and for any custom node defined by the user. It is also supported for nodes of type "identity", but slightly different input is expected in that case. See examples and the associated vignette for an in-depth explanation.

... Further named arguments needed to specify the node. Those can be parameters of distribution functions such as the p argument in the rbernoulli function for root nodes or arbitrary named arguments such as the betas argument of the node_gaussian function.

Details

To generate data using the sim_from_dag, sim_discrete_time or sim_discrete_event functions, it is required to create a DAG object first. This object needs to contain information about the causal structure of the data (e.g. which variable causes which variable) and the specific structural equations for each variable (information about causal coefficients, type of distribution etc.). In this package, the node and/or node_td functions are used in conjunction with the empty_dag function to create this object.

This works by first initializing an empty DAG using the empty_dag function and then adding multiple calls to the node and/or node_td functions to it using a simple +, where each call to node and/or node_td adds information about a single node that should be generated. Multiple examples are given below.

In each call to node or node_td the user needs to indicate what the node should be called (name), which function should be used to generate the node (type), whether the node has any parents and if so which (parents) and any additional arguments needed to actually call the data-generating function of this node later passed to the three-dot syntax (...).

node ***vs.*** node_td:

By calling node you are indicating that this node is a time-fixed variable which should only be generated once. By using node_td you are indicating that it is a time-dependent node, which will be updated at each step in time when using a discrete-time simulation, or at event changes in discrete-event simulations.

node_td should only be used if you are planning to perform a discrete-time or discrete-event simulation with the sim_discrete_time or sim_discrete_event functions. DAG objects including time-dependent nodes may not be used in the sim_from_dag function.

Implemented Root Node Types:

Any function can be used to generate root nodes. The only requirement is that the function has at least one named argument called n which controls the length of the resulting vector. For example, the user could specify a node of type "rnorm" to create a normally distributed node with no parents. The argument n will be set internally, but any additional arguments can be specified using the ... syntax. In the type="rnorm" example, the user could set the mean and standard deviation using node(name="example", type="rnorm", mean=10, sd=5).

For convenience, this package additionally includes five custom root-node functions:

- "rbernoulli": Draws randomly from a bernoulli distribution.
- "rcategorical": Draws randomly from any discrete probability density function.
- "rsample": Draws random samples from a given vector.
- "rtexp": Draws random values from a left-truncated exponential distribution.
- "rconstant": Used to set a variable to a constant value.

Implemented Child Node Types:

Currently, the following node types are implemented directly for convenience:

- "gaussian": A node based on (mixed) linear regression.
- "binomial": A node based on (mixed) binomial regression.
- "conditional_prob": A node based on conditional probabilities.
- "conditional_distr": A node based on conditional draws from different distributions.
- "multinomial": A node based on multinomial regression.
- "polr": A node based on ordered logistic or probit regression.
- "poisson": A node based on (mixed) poisson regression.
- "negative_binomial": A node based on negative binomial regression.
- "zeroinfl": A node based on a zero-inflated poisson or negative binomial regression.
- "gamma": A node based on gamma regression.
- "beta": A node based on beta regression.
- "identity": A node that is just some R expression of other nodes.
- "mixture": A node that is a mixture of different node definitions.
- "cox": A node based on cox-regression.
- "aalen": A node based on an Aalen additive hazards model.
- "aftreg": A node based on an accelerated failure time model.
- "ahreg": A node based on an accelerated hazard model.
- "ehreg": A node based on a extended hazard model.
- "poreg": A node based on a proportional odds model.
- "ypreg": A node based on a Young and Prentice model.

For custom child node types, see below or consult the vignette on custom node definitions.

Implemented Time-Dependent Node Types:

Currently, the following node types are implemented directly for convenience to use in `node_td` calls:

- "time_to_event": A node based on repeatedly checking whether an event occurs at each point in time.
- "competing_events": A node based on repeatedly checking whether one of multiple mutually exclusive events occurs at each point in time.
- "next_time": A node that draws the time of the next event in discrete-event simulation.

However, the user may also use any of the child node types in a `node_td` call directly. For custom time-dependent node types, please consult the associated vignette.

Custom Node Types

It is very simple to write a new custom `node_function` to be used instead, allowing the user to use any `type` of data-generation mechanism for any type of node (root / child / time-dependent). All that is required of this function is, that it has the named arguments `data` (the sample as generated so far) and, if it's a child node, `parents` (a character vector specifying the parents) and outputs either a vector containing `n_sim` entries, or a `data.frame` with `n_sim` rows and an arbitrary amount of columns. More information about this can be found in the associated vignette: `vignette(topic="v_custom_nodes", package="simDAG")`.

Using child nodes as parents for other nodes:

If the data generated by a child node is categorical (such as when using `node_multinomial`) they can still be used as parents of other nodes for most standard node types without issues. All the user has to do is to use `formula` argument to supply an enhanced formula, instead of defining the `parents` and `betas` argument directly. This works well for all node types that directly support `formula` input and for all custom nodes specified by the user. See the associated vignette: `vignette(topic="v_using_formulas", package="simDAG")` for more information on how to correctly use formulas.

Cyclic causal structures:

The name DAG (directed **acyclic** graph) implies that cycles are not allowed. This means that if you start from any node and only follow the arrows in the direction they are pointing, there should be no way to get back to your original node. This is necessary both theoretically and for practical reasons if we are dealing with static DAGs created using the `node` function. If the user attempts to generate data from a static cyclic graph using the `sim_from_dag` function, an error will be produced.

However, in the realm of discrete-time or discrete-event simulations, cyclic causal structures are perfectly reasonable. A variable A at $t = 1$ may influence a variable B at $t = 2$, which in turn may influence variable A at $t = 3$ again. Therefore, when using the `node_td` function to simulate time-dependent data using the `sim_discrete_time` or `sim_discrete_event` function, cyclic structures are allowed to be present and no error will be produced.

Value

Returns a `DAG.node` object which can be added to a `DAG` object directly.

Note

Contrary to the R standard, this function does **NOT** support partial matching of argument names. This means that supplying `nam="age"` will not be recognized as `name="age"` and instead will be added as additional node argument used in the respective data-generating function call when using `sim_from_dag`.

Author(s)

Robin Denz

References

Denz, Robin and Nina Timmesfeld (2026). "Simulating Complex Cross-Sectional and Longitudinal Data using the simDAG R Package". Journal of Statistical Software 116 (2), doi: 10.18637/jss.v116.i02.

Examples

```
library(simDAG)

# creating a DAG with a single root node
dag <- empty_dag() +
  node("age", type="rnorm", mean=30, sd=4)

# creating a DAG with multiple root nodes
```

```
# (passing the functions directly to 'type' works too)
dag <- empty_dag() +
  node("sex", type=rbernoulli, p=0.5) +
  node("income", type=rnorm, mean=2700, sd=500)

# creating a DAG with multiple root nodes + multiple names in one node
dag <- empty_dag() +
  node("sex", type="rbernoulli", p=0.5) +
  node(c("income_1", "income_2"), type="rnorm", mean=2700, sd=500)

# also using child nodes
dag <- empty_dag() +
  node("sex", type="rbernoulli", p=0.5) +
  node("income", type="rnorm", mean=2700, sd=500) +
  node("sickness", type="binomial", parents=c("sex", "income"),
       betas=c(1.2, -0.3), intercept=-15) +
  node("death", type="binomial", parents=c("sex", "income", "sickness"),
       betas=c(0.1, -0.4, 0.8), intercept=-20)

# creating the same DAG as above, but using the enhanced formula interface
dag <- empty_dag() +
  node("sex", type="rbernoulli", p=0.5) +
  node("income", type="rnorm", mean=2700, sd=500) +
  node("sickness", type="binomial",
       formula= ~ -15 + sexTRUE*1.2 + income*-0.3) +
  node("death", type="binomial",
       formula= ~ -20 + sexTRUE*0.1 + income*-0.4 + sickness*0.8)

# using time-dependent nodes
# NOTE: to simulate data from this DAG, the sim_discrete_time() function needs
#       to be used due to "sickness" being a time-dependent node
dag <- empty_dag() +
  node("sex", type="rbernoulli", p=0.5) +
  node("income", type="rnorm", mean=2700, sd=500) +
  node_td("sickness", type="binomial", parents=c("sex", "income"),
          betas=c(0.1, -0.4), intercept=-50)

# we could also use a DAG with only time-varying variables
dag <- empty_dag() +
  node_td("vaccine", type="time_to_event", prob_fun=0.001, event_duration=21) +
  node_td("covid", type="time_to_event", prob_fun=0.01, event_duration=15,
          immunity_duration=100)
```

---

node_aalen *Generate Data from an Aalen Additive Hazards Model*

Description

Data from the parents is used to generate the node using Aalen additive hazards regression using the inversion method. Currently, only time-constant coefficients and a constant baseline hazard function are supported.

Usage

```
node_aalen(data, parents, formula=NULL, betas, intercept,
           cens_dist=NULL, cens_args, name,
           as_two_cols=TRUE, left=0)
```

Arguments

data	A data.table (or something that can be coerced to a data.table) containing all columns specified by parents.
parents	A character vector specifying the names of the parents that this particular child node has. If non-linear combinations or interaction effects should be included, the user may specify the formula argument instead.
formula	An optional formula object to describe how the node should be generated or NULL (default). This argument only works if the function is used as a node type in a node call. See ?node or the associated vignette for more information about how the formula argument should be specified in this package.
betas	A numeric vector with length equal to parents, specifying the causal beta coefficients used to generate the node.
intercept	A single number, specifying the intercept of the model.
cens_dist	A single character naming the distribution function that should be used to generate the censoring times or a suitable function. For example, "runif" could be used to generate uniformly distributed censoring times. Set to NULL (default) to get no censoring.
cens_args	A list of named arguments which will be passed to the function specified by the cens_dist argument.
name	A single character string specifying the name of the node.
as_two_cols	Either TRUE or FALSE, specifying whether the output should be divided into two columns. When cens_dist is specified, this argument will always be treated as TRUE because two columns are needed to encode both the time to the event and the status indicator. When no censoring is applied, however, users may set this argument to FALSE to simply return the numbers as they are.
left	A single number, specifying the left-truncation time. If set to something > 0, only times that are larger than this value will be generated. Is set to 0 by default, so that no left-truncation is used.

Details

This function generates survival times according to a Aalen additive hazards model with time-constant beta coefficients and a time-constant baseline hazard. Time-dependent effects or time-dependent baseline hazards are currently not supported. To also include censoring, this function allows the user to supply a function that generates random censoring times. If the censoring time is smaller than the generated survival time, the individual is considered censored.

Like the other time-to-event based node type functions, this function usually adds **two** columns to the resulting dataset instead of one. The first column is called paste0(name, "_status") and is a logical variable, where TRUE indicates that the event has happened and FALSE indicates right-censoring. The second column is named paste0(name, "_time") and includes the survival or

censoring time corresponding to the previously mentioned event indicator. This is the standard format for right-censored time-to-event data without time-varying covariates. If no censoring is applied, this behavior can be turned off using the as_two_cols argument.

Value

Returns a data.table of length nrow(data) containing two columns if as_two_cols=TRUE and always when cens_dist is specified. In this case, both columns start with the nodes name and end with _status and _time. The first is a logical vector, the second a numeric one. If as_two_cols=FALSE and cens_dist is NULL, a numeric vector is returned instead.

Author(s)

Robin Denz

References

Aalen, Odd O. A Linear Regression Model for the Analysis of Life Times. Statistics in Medicine. 1989; (8): 907-925.

Examples

```
library(simDAG)

set.seed(34543)

# define DAG, here with two baseline covariates and
# no censoring of Y
dag <- empty_dag() +
  node("A", type="runif") +
  node("B", type="rbernoulli") +
  node("Y", type="aalen", formula= ~ 0.1 + A*0.2 + B*-0.05)

sim_dat <- sim_from_dag(dag=dag, n_sim=1000)
head(sim_dat)
```

---

node_beta *Generate Data from a Beta Regression Model*

Description

Data from the parents is used to generate the node using beta regression by applying the betas to the design matrix, transforming the resulting linear predictor to a conditional mean, and sampling from the rbeta function.

Usage

```
node_beta(data, parents, formula=NULL, betas,
          intercept, precision, link="logit")
```

Arguments

data	A data.table (or something that can be coerced to a data.table) containing all columns specified by parents.
parents	A character vector specifying the names of the parents that this particular child node has. If non-linear combinations or interaction effects should be included, the user may specify the formula argument instead.
formula	An optional formula object to describe how the node should be generated or NULL (default). If supplied it should start with ~, having nothing else on the left hand side. The right hand side may contain any valid formula syntax, such as A + B or A + B + I(A^2), allowing non-linear effects. If this argument is defined, there is no need to define the parents argument. For example, using parents=c("A", "B") is equal to using formula= ~ A + B. Random effects and random slopes are currently not supported here.
betas	A numeric vector with length equal to the number of columns in the design matrix, specifying the causal beta coefficients used to generate the node.
intercept	A single number specifying the intercept that should be used when generating the node.
precision	A single finite, positive number specifying the precision parameter ϕ of the beta distribution. The precision is the sum of the two shape parameters and determines the conditional variance of the generated values relative to their conditional mean.
link	The link function used to transform the linear predictor to the conditional mean. The default is "logit". Other allowed values are "probit", "cloglog", "loglog", and "cauchit".

Details

This function uses the linear predictor defined by the betas and the input design matrix to obtain an observation-specific conditional mean. It does so by calculating the linear predictor using the data, betas, and intercept, applying the inverse of the specified link function, and passing the resulting conditional mean to the rbeta function of the **stats** package.

For an observation with conditional mean μ_i and precision parameter ϕ, the generated outcome follows a beta distribution with shape parameters $\mu_i\phi$ and $(1-\mu_i)\phi$. Consequently, the conditional mean is $E(Y_i \mid X_i) = \mu_i$ and the conditional variance is $\mathrm{Var}(Y_i \mid X_i) = \mu_i(1-\mu_i)/(\phi+1)$.

Every non-missing conditional mean must be finite and strictly between zero and one. This node type currently does not support inclusion of random effects or random slopes in the formula. The precision parameter is fixed across observations.

Value

Returns a numeric vector of length nrow(data).

Author(s)

Robin Denz

See Also

empty_dag, node, node_td, sim_from_dag, sim_discrete_time

Examples

```
library(simDAG)

set.seed(124554)

dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("proportion", type="beta", precision=10,
       formula= ~ -1 + age*0.01 + sexTRUE*0.2)

sim_dat <- sim_from_dag(dag=dag, n_sim=100, sort_dag=FALSE)
head(sim_dat)
```

---

node_binomial *Generate Data from a (Mixed) Binomial Regression Model*

Description

Data from the parents is used to generate the node using binomial regression (usually logistic regression) by predicting the covariate specific probability and sampling from a Bernoulli distribution accordingly. Allows inclusion of arbitrary random effects and slopes for logistic models.

Usage

```
node_binomial(data, parents, formula=NULL, betas, intercept,
              return_prob=FALSE, output="logical", labels=NULL,
              var_corr=NULL, link="logit")
```

Arguments

data	A data.table (or something that can be coerced to a data.table) containing all columns specified by parents.
parents	A character vector specifying the names of the parents that this particular child node has. If non-linear combinations or interaction effects should be included, the user may specify the formula argument instead.
formula	An optional formula object to describe how the node should be generated or NULL (default). If supplied it should start with ~, having nothing else on the left hand side. The right hand side may contain any valid formula syntax, such as A + B or A + B + I(A^2), allowing non-linear effects. If this argument is defined, there is no need to define the parents argument. For example, using parents=c("A", "B") is equal to using formula= ~ A + B. May contain random effects and random slopes, in which case the **simr** package is used to generate the data. See details.

betas	A numeric vector with length equal to parents, specifying the causal beta coefficients used to generate the node.
intercept	A single number specifying the intercept that should be used when generating the node.
return_prob	Either TRUE or FALSE (default). If TRUE, the calculated probability is returned instead of the results of bernoulli trials. This argument is ignored if random effects or random slopes are specified in the formula input.
output	A single character string, must be either "logical" (default), "numeric", "character" or "factor". If output="character" or output="factor", the labels (or levels in case of a factor) can be set using the labels argument.
labels	A character vector of length 2 or NULL (default). If NULL, the resulting vector is returned as is. If a character vector is supplied and output="character" or output="factor" is used, all TRUE values are replaced by the first entry of this vector and all FALSE values are replaced by the second argument of this vector. The output will then be a character variable or factor variable, depending on the output argument. This argument is ignored if output is set to "numeric" or "logical".
var_corr	Variances and covariances for random effects. Only used when formula contains mixed model syntax. If there are multiple random effects, their parameters should be supplied as a named list. More complex structures are also supported. This argument is directly passed to the makeLmer function of the **simr** package. Please consult the documentation of that package for more information on how mixed models should be specified. Some guidance can also be found in the "Issues" section of the official **simr** github page.
link	The link function used to transform the linear predictor to the probability scale. For a standard logistic regression model, this should be set to "logit" (which is the default). Other allowed values are "identity", "probit", "log", "cloglog" and "cauchit", which are defined the same way as in the classic glm function.

Details

Using the normal form a logistic regression model, the observation specific event probability is generated for every observation in the dataset. Using the rbernoulli function, this probability is then used to take one bernoulli sample for each observation in the dataset. If only the probability should be returned return_prob should be set to TRUE.

Formal Description:

Formally, the data generation (when using link="logit") can be described as:

$$Y \sim Bernoulli(logit(\texttt{intercept} + \texttt{parents}_1 \cdot \texttt{betas}_1 + ... + \texttt{parents}_n \cdot \texttt{betas}_n)),$$

where $Bernoulli(p)$ denotes one Bernoulli trial with success probability p, n is the number of parents (length(parents)) and the $logit(x)$ function is defined as:

$$logit(x) = ln(\frac{x}{1-x}).$$

For example, given `intercept=-15`, `parents=c("A", "B")` and `betas=c(0.2, 1.3)` the data generation process is defined as:

$$Y \sim Bernoulli(logit(-15 + A \cdot 0.2 + B \cdot 1.3)).$$

It works the same way for other `link` functions, with the only difference being that $logit()$ would be replaced.

Output Format:

By default this function returns a logical vector containing only `TRUE` and `FALSE` entries, where `TRUE` corresponds to an event and `FALSE` to no event. This may be changed by using the `output` and `labels` arguments. The last three arguments of this function are ignored if `return_prob` is set to `TRUE`.

Random Effects and Random Slopes:

This function also allows users to include arbitrary amounts of random slopes and random effects using the `formula` argument. If this is done, the `formula`, and `data` arguments are passed to the variables of the same name in the `makeGlmer` function of the **simr** package. The `fixef` argument of that function will be passed the numeric vector `c(intercept, betas)` and the `VarCorr` argument receives the `var_corr` argument as input. If used as a node type in a `DAG`, all of this is taken care of behind the scenes. Users can simply use the regular enhanced formula interface of the `node` function to define these formula terms, as shown in detail in the formula vignette (`vignette(topic="v_using_formulas", package="simDAG")`). Please consult that vignette for examples. Also, please note that inclusion of random effects or random slopes usually results in significantly longer computation times.

Value

Returns a logical vector (or numeric vector if `return_prob=TRUE`) of length `nrow(data)`.

Author(s)

Robin Denz

See Also

`empty_dag`, `node`, `node_td`, `sim_from_dag`, `sim_discrete_time`

Examples

```
library(simDAG)

set.seed(5425)

# define needed DAG
dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("smoking", type="binomial", parents=c("age", "sex"),
       betas=c(1.1, 0.4), intercept=-2)
```

```
# define the same DAG, but using a pretty formula
dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("smoking", type="binomial",
       formula= ~ -2 + age*1.1 + sexTRUE*0.4)

# simulate data from it
sim_dat <- sim_from_dag(dag=dag, n_sim=100)

# returning only the estimated probability instead
dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("smoking", type="binomial", parents=c("age", "sex"),
       betas=c(1.1, 0.4), intercept=-2, return_prob=TRUE)

sim_dat <- sim_from_dag(dag=dag, n_sim=100)

## an example using a random effect
if (requireNamespace("simr")) {

library(simr)

dag_mixed <- empty_dag() +
  node("School", type="rcategorical", probs=rep(0.1, 10),
       labels=LETTERS[1:10]) +
  node("Age", type="rnorm", mean=12, sd=2) +
  node("Grade", type="binomial", formula= ~ -10 + Age*1.2 + (1|School),
       var_corr=0.3)

sim_dat <- sim_from_dag(dag=dag_mixed, n_sim=100)
}
```

`node_competing_events` *Generate Data with Multiple Mutually Exclusive Events in Discrete-Time Simulation*

Description

This node essentially models a categorical time-dependent variable for which the time and the type of the event will be important for later usage. It adds two columns to `data`: `name_event` (which type of event the person is currently experiencing) and `name_time` (the time at which the current event started). Can only be used inside of the `sim_discrete_time` function, not outside of it. Past events and their kind are stored in two lists. See details.

Usage

```
node_competing_events(data, parents, sim_time, name,
                      prob_fun, ..., event_duration=c(1, 1),
```

```
                    immunity_duration=max(event_duration),
                    save_past_events=TRUE, check_inputs=TRUE,
                    envir)
```

Arguments

data	A data.table containing all columns specified by parents. Similar objects such as data.frames are not supported.
parents	A character vector specifying the names of the parents that this particular child node has.
sim_time	The current time of the simulation.
name	The name of the node. This will be used as prefix before the _event, _time, _past_event_times and _past_event_kind columns.
prob_fun	A function that returns a numeric matrix with nrow(data) rows and one column storing probabilities of occurrence for each possible event type plus a column for no events. For example, if there are two possible events such as recurrence and death, the matrix would need to contain three columns. The first storing the probability of no-event and the other two columns storing probabilities for recurrence and death per person. Since the numbers are probabilities, the matrix should only contain numbers between 0 and 1 that sum to 1 in each row. These numbers specify the person-specific probability of experiencing the events modeled by this node at the particular point in time of the simulation. The corresponding event will be generated internally using the rcategorical function.
...	An arbitrary number of additional named arguments passed to prob_fun. Ignore this if you do not want to pass any arguments.
event_duration	A numeric vector containing one positive integer for each type of event of interest, specifying how long that event should last. For example, if we are interested in modelling the time to a cardiovascular event with death as competing event, this argument would need 2 entries. One would specify the duration of the cardiovascular event and the other would be Inf (because death is a terminal event).
immunity_duration	A single number >= max(event_duration) specifying how long the person should be immune to all events after experiencing one. The count internally starts when the event starts, so in order to use an immunity duration of 10 time units after the event is over max(event_duration) + 10 should be used.
save_past_events	When the event modeled using this node is recurrent (immunity_duration < Inf & any(event_duration < Inf)), the same person may experience multiple events over the course of the simulation. Those are generally stored in the ce_past_events list and ce_past_causes list which are included in the output of the sim_discrete_time function. This extends the runtime and increases RAM usage, so if you are not interested in the timing of previous events or if you are using save_states="all" this functionality can be turned off by setting this argument to FALSE

check_inputs	Whether to perform plausibility checks for the user input or not. Is set to TRUE by default, but can be set to FALSE in order to speed things up when using this function in a simulation study or something similar.
envir	Only used internally to efficiently store the past event times. Cannot be used by the user.

Details

When performing discrete-time simulation using the sim_discrete_time function, the standard node functions implemented in this package are usually not sufficient because they don't capture the time-dependent nature of some very interesting variables. Often, the variable that should be modelled has some probability of occurring at each point in time. Once it does occur, it has some kind of influence on other variables for a period of time until it goes back to normal (or doesn't). This could be a car crash, a surgery, a vaccination etc. The node_time_to_event node function can be used to model these kinds of nodes in a fairly straightforward fashion.

This function is an extended version of the node_time_to_event function. Instead of simulating a binary event, it can generate multiple competing events, where the occurrence of one event at time t is mutually exclusive with the occurrence of an other event at that time. In other words, multiple events are possible, but only one can occur at a time.

How it Works:

At $t = 1$, this node will be initialized for the first time. It adds two columns to the data: name_event (whether the person currently has an event) and name_time (the time at which the current event started) where name is the name of the node. Additionally, it adds a list with max_t entries to the ce_past_events list returned by the sim_discrete_time function, which records which individuals experienced a new event at each point in time. The ce_past_causes list additionally records which kind of event happened at that time.

In a nutshell, it simply models the occurrence of some event by calculating the probability of occurrence at t and drawing a single multinomial trial from this probability. If the trial is a "success", the corresponding event column will be set to the drawn event type (described using integers, where 0 is no event and all other events are numbered consecutively), the time column will be set to the current simulation time t and the columns storing the past event times and types will receive an entry.

The event column will stay at its new integer value until the event is over. The duration for that is controlled by the event_duration parameter. When modeling terminal events such as death, one can simply set this parameter to Inf, making the event eternal. In many cases it will also be necessary to implement some kind of immunity after the event, which can be done using the immunity_duration argument. This effectively sets the probability of another occurrence of the event to 0 in the next immunity_duration time steps. During the immunity duration, the event may be > 0 (if the event is still ongoing) or 0 (if the event_duration for that event type has already passed).

The probability of occurrence is calculated using the function provided by the user using the prob_fun argument. This can be an arbitrary complex function. The only requirement is that it takes data as a first argument. The columns defined by the parents argument will be passed to this argument automatically. If it has an argument called sim_time, the current time of the simulation will automatically be passed to it as well. Any further arguments can be passed using the prob_fun_args argument. A simple example could be a multinomial logistic regression node, in which the probabilities are calculated as an additive linear combination of the columns defined by

parents. A more complex function could include simulation-time dependent effects, further effects dependent on past event times etc. Examples can be found below and in the vignettes.

What can be done with it:

This type of node naturally support the implementation of competing events, where some may be terminal or recurrent in nature and may be influenced by pretty much anything. By specifying the parents and prob_fun arguments correctly, it is possible to create an event type that is dependent on past events of itself or other time-to-event variables and other variables in general. The user can include any amount of these nodes in their simulation. It may also be used to simulate any kind of binary time-dependent variable that one would usually not associate with the name "event" as well. It is very flexible, but it does require the user to do some coding by themselves.

What can't be done with it:

This function may only be used to generate competing events, meaning that the occurrence of event 1 at $t = 1$ makes it impossible for event 2 at $t = 1$ to occur. If the user wants to generate multiple events that are not mutually exclusive, he or she may add multiple node_time_to_event based nodes to the dag argument of the sim_discrete_time function.

In fact, a competing events node may be simulated using multiple calls to the node_time_to_event based nodes as well, by defining the prob_fun argument of these nodes in such a way that the occurrence of event A makes the occurrence of event B impossible. This might actually be easier to implement in some situations, because it doesn't require the user to manually define a probability function that outputs a matrix of subject-specific probabilities.

Value

Returns a data.table containing the updated columns of the node.

Note

This function cannot be called outside of the sim_discrete_time function. It only makes sense to use it as a type in a node_td function call, as described in the documentation and vignettes.

Author(s)

Robin Denz

See Also

empty_dag, node, node_td, sim_from_dag, sim_discrete_time

Examples

```
library(simDAG)

## a competing_events node with only terminal events, all with a constant
## probability of occurrence, independent of any other variable
prob_death_illness <- function(data) {

  # simply repeat the same probabilities for everyone
  n <- nrow(data)
  p_mat <- matrix(c(rep(0.9, n), rep(0.005, n), rep(0.005, n)),
```

```
                  byrow = FALSE, ncol=3)

  return(p_mat)
}

dag <- empty_dag() +
  node_td("death_illness", type="competing_events", prob_fun=prob_death_illness,
          event_duration=c(Inf, Inf))

## making one of the event-types terminal and the other recurrent
dag <- empty_dag() +
  node_td("death_illness", type="competing_events", prob_fun=prob_death_illness,
          event_duration=c(15, Inf))

## call the sim_discrete_time function to generate data from it
sim <- sim_discrete_time(dag, n_sim=100, max_t=500)

## more examples on how to use the sim_discrete_time function can be found
## in the documentation page of the node_time_to_event function and
## in the package vignettes
```

`node_conditional_distr` *Generate Data by Sampling from Different Distributions based on Strata*

Description

This function can be used to generate any kind of dichotomous, categorical or numeric variables dependent on one or more categorical variables by randomly sampling from user-defined distributions in each strata defined by the nodes parents. An even more flexible node type, allowing arbitrary `node` definitions for different subsets of the previously generated data is included in `node_mixture`.

Usage

```
node_conditional_distr(data, parents, distr, default_distr=NULL,
                       default_distr_args=list(), default_val=NA_real_,
                       coerce2numeric=TRUE, check_inputs=TRUE)
```

Arguments

`data`	A `data.table` (or something that can be coerced to a `data.table`) containing all columns specified by `parents`.
`parents`	A character vector specifying the names of the parents that this particular child node has.
`distr`	A named list where each element corresponds to one stratum defined by parents. If only one name is given in `parents`, this means that there should be one element for possible values of the variable given in `parents`. If the node

has multiple parents, there needs to be one element for possible combinations of parents (see examples). The values of those elements should be a list themselves, with the first argument being a callable function (such as rnorm, rcategorical, ...) and the rest should be named arguments of that function. Any function can be used, as long as it returns a vector of n values, with n being an argument of the function. n is set internally based on the stratum size and cannot be set by the user. If this list does not contain one element for each possible strata defined by parents, the default_val or default_distr arguments will be used.

default_distr A function that should be used to generate values for all strata that are not explicitly mentioned in the distr argument, or NULL (default). If NULL, the default_val argument will be used to fill the missing strata with values. A function passed to this argument should contain the argument n, which should define the number of samples to generate. It should return a vector with n values. Some examples are (again), rnorm or rbernoulli.

default_distr_args
A named list of arguments which are passed to the function defined by the default_distr argument. Ignored if default_distr is NULL.

default_val A single value which is used as an output for strata that are not mentioned in distr. Ignored if default_distr is not NULL.

coerce2numeric A single logical value specifying whether to try to coerce the resulting variable to numeric or not.

check_inputs A single logical value specifying whether to perform input checks or not. May be set to TRUE to speed up things a little if you are sure your input is correct.

Details

Utilizing the user-defined distribution in each stratum of parents (supplied using the distr argument), this function simply calls the user-defined function with the arguments given by the user to generate a new variable. This allows the new variable to consist of a mix of different distributions, based on categorical parents.

Formal Description:

Formally, the data generation process can be described as a series of conditional equations. For example, suppose that there is just one parent node sex with the levels male and female with the goal of creating a continuous outcome that has a normal distribution of $N(10, 3)$ for males and $N(7, 2)$ for females. The conditional equation is then:

$$Y \sim \begin{cases} N(10,3), & \text{if sex="male"} \\ N(7,2), & \text{if sex="female"} \end{cases},$$

If there are more than two variables, the conditional distribution would be stratified by the intersection of all subgroups defined by the variables.

Value

Returns a numeric vector of length nrow(data).

Author(s)

Robin Denz

See Also

empty_dag, node, node_td, sim_from_dag, sim_discrete_time

Examples

```
library(simDAG)

set.seed(42)

#### with one parent node ####

# define conditional distributions
distr <- list(male=list("rnorm", mean=100, sd=5),
              female=list("rcategorical", probs=c(0.1, 0.2, 0.7)))

# define DAG
dag <- empty_dag() +
  node("sex", type="rcategorical", labels=c("male", "female"),
       output="factor", probs=c(0.4, 0.6)) +
  node("chemo", type="rbernoulli", p=0.5) +
  node("A", type="conditional_distr", parents="sex", distr=distr)

# generate data
data <- sim_from_dag(dag=dag, n_sim=1000)


#### with two parent nodes ####

# define conditional distributions with interaction between parents
distr <- list(male.FALSE=list("rnorm", mean=100, sd=5),
              male.TRUE=list("rnorm", mean=100, sd=20),
              female.FALSE=list("rbernoulli", p=0.5),
              female.TRUE=list("rcategorical", probs=c(0.1, 0.2, 0.7)))

# define DAG
dag <- empty_dag() +
  node("sex", type="rcategorical", labels=c("male", "female"),
       output="factor", probs=c(0.4, 0.6)) +
  node("chemo", type="rbernoulli", p=0.5) +
  node("A", type="conditional_distr", parents=c("sex", "chemo"), distr=distr)

# generate data
data <- sim_from_dag(dag=dag, n_sim=1000)
```

node_conditional_prob *Generate Data Using Conditional Probabilities*

Description

This function can be used to generate dichotomous or categorical variables dependent on one or more categorical variables where the probabilities of occurrence in each strata defined by those variables is known.

Usage

```
node_conditional_prob(data, parents, probs, default_probs=NULL,
                      default_val=NA, labels=NULL,
                      coerce2factor=FALSE, check_inputs=TRUE)
```

Arguments

data	A data.table (or something that can be coerced to a data.table) containing all columns specified by parents.
parents	A character vector specifying the names of the parents that this particular child node has.
probs	A named list where each element corresponds to one stratum defined by parents. If only one name is given in parents, this means that there should be one element for possible value of the variable given in parents. If the node has multiple parents, there needs to be one element for possible combinations of parents (see examples). The values of those elements should either be a single number, corresponding to the probability of occurrence of a single event/value in case of a dichotomous variable, or a vector of probabilities that sum to 1, corresponding to class probabilities. In either case, the length of all elements should be the same. If possible strata of parents (or their possible combinations in case of multiple parents) are omitted, the result will be set to default_val for these omitted strata. See argument default_val and argument default_probs for an alternative.
default_probs	If not all possible strata of parents are included in probs, the user may set default probabilities for all omitted strata. For example, if there are three strata (A, B and C) defined by parents and probs only contains defined probabilities for strata A, the probabilities for strata B and C can be set simultaneously by using this argument. Should be a single value between 0 and 1 for Bernoulli trials and a numeric vector with sum 1 for multinomial trials. If NULL (default) the value of the produced output for missing strata will be set to default_val (see below).
default_val	Value of the produced variable in strata that are not included in the probs argument. If default_probs is not NULL, that arguments functionality will be used instead.

`labels`	A vector of labels for the generated output. If `NULL` (default) and the output is dichotomous, a logical variable will be returned. If `NULL` and the output is categorical, it simply uses integers starting from 1 as class labels.
`coerce2factor`	A single logical value specifying whether to return the drawn events as a factor or not.
`check_inputs`	A single logical value specifying whether input checks should be performed or not. Set to `FALSE` to save some computation time in simulations.

Details

Utilizing the user-defined discrete probability distribution in each stratum of `parents` (supplied using the `probs` argument), this function simply calls either the `rbernoulli` or the `rcategorical` function.

Formal Description:

Formally, the data generation process can be described as a series of conditional equations. For example, suppose that there is just one parent node `sex` with the levels `male` and `female` with the goal of creating a binary outcome that has a probability of occurrence of 0.5 for males and 0.7 for females. The conditional equation is then:

$$Y \sim Bernoulli(p),$$

where:

$$p = \begin{cases} 0.5, & \text{if } \texttt{sex="male"} \\ 0.7, & \text{if } \texttt{sex="female"} \end{cases},$$

and $Bernoulli(p)$ is the Bernoulli distribution with success probability p. If the outcome has more than two categories, the Bernoulli distribution would be replaced by $Multinomial(p)$ with p being replaced by a matrix of class probabilities. If there are more than two variables, the conditional distribution would be stratified by the intersection of all subgroups defined by the variables.

An even more flexible node type, allowing arbitrary `node` definitions for different subsets of the previously generated data is included in `node_mixture`.

Value

Returns a numeric vector of length `nrow(data)`.

Author(s)

Robin Denz

See Also

`empty_dag`, `node`, `node_td`, `sim_from_dag`, `sim_discrete_time`

Examples

```
library(simDAG)

set.seed(42)

#### two classes, one parent node ####

# define conditional probs
probs <- list(male=0.5, female=0.8)

# define DAG
dag <- empty_dag() +
  node("sex", type="rcategorical", labels=c("male", "female"),
       output="factor", probs=c(0.5, 0.5)) +
  node("chemo", type="rbernoulli", p=0.5) +
  node("A", type="conditional_prob", parents="sex", probs=probs)

# generate data
data <- sim_from_dag(dag=dag, n_sim=1000)


#### three classes, one parent node ####

# define conditional probs
probs <- list(male=c(0.5, 0.2, 0.3), female=c(0.8, 0.1, 0.1))

# define DAG
dag <- empty_dag() +
  node("sex", type="rcategorical", labels=c("male", "female"),
       output="factor", probs=c(0.5, 0.5)) +
  node("chemo", type="rbernoulli", p=0.5) +
  node("A", type="conditional_prob", parents="sex", probs=probs)

# generate data
data <- sim_from_dag(dag=dag, n_sim=1000)


#### two classes, two parent nodes ####

# define conditional probs
probs <- list(male.FALSE=0.5,
              male.TRUE=0.8,
              female.FALSE=0.1,
              female.TRUE=0.3)

# define DAG
dag <- empty_dag() +
  node("sex", type="rcategorical", labels=c("male", "female"),
       output="factor", probs=c(0.5, 0.5)) +
  node("chemo", type="rbernoulli", p=0.5) +
  node("A", type="conditional_prob", parents=c("sex", "chemo"), probs=probs)
```

```
# generate data
data <- sim_from_dag(dag=dag, n_sim=1000)


#### three classes, two parent nodes ####

# define conditional probs
probs <- list(male.FALSE=c(0.5, 0.1, 0.4),
              male.TRUE=c(0.8, 0.1, 0.1),
              female.FALSE=c(0.1, 0.7, 0.2),
              female.TRUE=c(0.3, 0.4, 0.3))

# define dag
dag <- empty_dag() +
  node("sex", type="rcategorical", labels=c("male", "female"),
       output="factor", probs=c(0.5, 0.5)) +
  node("chemo", type="rbernoulli", p=0.5) +
  node("A", type="conditional_prob", parents=c("sex", "chemo"), probs=probs)

# generate data
data <- sim_from_dag(dag=dag, n_sim=1000)
```

---

node_cox *Generate Data from a Cox-Regression Model*

Description

Data from the parents is used to generate the node using cox-regression using the method of Bender et al. (2005). Directly allows users to specify arbitrary baseline hazard functions.

Usage

```
node_cox(data, parents, formula=NULL, betas,
         surv_dist, lambda, gamma,
         cens_dist=NULL, cens_args, name,
         as_two_cols=TRUE, left=0,
         basehaz_grid=NULL, extrapolate=FALSE,
         as_integer=FALSE, ...)
```

Arguments

data	A data.table (or something that can be coerced to a data.table) containing all columns specified by parents.
parents	A character vector specifying the names of the parents that this particular child node has. If non-linear combinations or interaction effects should be included, the user may specify the formula argument instead.

formula — An optional formula object to describe how the node should be generated or NULL (default). If supplied it should start with ~, having nothing else on the left hand side. The right hand side may contain any valid formula syntax, such as A + B or A + B + I(A^2), allowing non-linear effects. If this argument is defined, there is no need to define the parents argument. For example, using parents=c("A", "B") is equal to using formula= ~ A + B.

betas — A numeric vector with length equal to parents, specifying the causal beta coefficients used to generate the node.

surv_dist — A single character specifying the distribution that should be used when generating the survival times. Can be either "weibull" or "exponential". Alternatively, a custom function with a single argument, defining the baseline hazard, may be supplied. This may be any function, as long as it is vectorized and returns strictly positive numbers for any supplied time.

lambda — A single number used as parameter defined by surv_dist. Ignored if surv_dist is a function.

gamma — A single number used as parameter defined by surv_dist. Ignored if surv_dist is a function.

cens_dist — A single character naming the distribution function that should be used to generate the censoring times or a suitable function. For example, "runif" could be used to generate uniformly distributed censoring times. Set to NULL to get no censoring (default).

cens_args — A list of named arguments which will be passed to the function specified by the cens_dist argument.

name — A single character string specifying the name of the node.

as_two_cols — Either TRUE or FALSE, specifying whether the output should be divided into two columns. When cens_dist is specified, this argument will always be treated as TRUE because two columns are needed to encode both the time to the event and the status indicator. When no censoring is applied, however, users may set this argument to FALSE to simply return the numbers as they are.

left — Either a single number >= 0, or a numeric vector of length nrow(data) containing only numbers >= 0. When simulating the survival times, only numbers larger than these will be generated using correctly left-truncated sampling. Note that this does not affect the censoring times, only the generated survival times will be left-truncated. Set to 0 (default) to not use left-truncation.

basehaz_grid — A numeric vector specifying the time grid used to numerically approximate the cumulative baseline hazard, whenever surv_dist is a function (ignored otherwise). This grid should not include 0. Internally, it is used to first evaluate the surv_dist at each value on the grid. Then, the trapezoid rule is used to integrate the function over time to obtain the cumulative baseline hazard, which is then inverted to simulate the survival times.

extrapolate — Either TRUE or FALSE, specifying whether extrapolation of the inverted cumulative baseline hazard should be allowed when supplying a function to surv_dist (ignored otherwise). Whenever a cumulative hazard is generated that is larger than the cumulative baseline hazard at max(basehaz_grid), extrapolation is needed. If this argument is set to FALSE (default), an error will be returned in

such cases. In almost all scenarios, this argument should be kept at FALSE. Only when users are strictly interested in values lower than max(basehaz_grid), it may be set to TRUE. In this case, all values that should be larger than max(basehaz_grid) are simply set to max(basehaz_grid) with no indicator. May be useful for discrete-event simulations conducted using sim_discrete_event.

as_integer — Either TRUE or FALSE, specifying whether the generated continuous times should be rounded up to the next integer (using ceiling).

... — Further arguments passed to internal functions. Should usually not be used by users.

Details

The survival times are generated according to the Cox proportional-hazards regression model as defined by the user. How exactly the data-generation works is described in detail in Bender et al. (2005). Briefly, it uses the method of inverted cumulative hazards. When surv_dist is not a function, the exact equations given in Bender et al. (2005) are used. When a custom function is supplied to surv_dist instead, numerical approximations are used. To also include censoring, this function allows the user to supply a function that generates random censoring times. If the censoring time is smaller than the generated survival time, the individual is considered censored.

Unlike the other node type functions, this function usually adds **two** columns to the resulting dataset instead of one. The first column is called paste0(name, "_status") and is a logical variable, where TRUE indicates that the event has happened and FALSE indicates right-censoring. The second column is named paste0(name, "_time") and includes the survival or censoring time corresponding to the previously mentioned event indicator. This is the standard format for right-censored time-to-event data without time-varying covariates. If no censoring is applied, this behavior can be turned off using the as_two_cols argument.

To simulate more complex time-to-event data, the user may need to use the sim_discrete_time or sim_discrete_event functions instead.

Value

Returns a data.table of length nrow(data) containing two columns if as_two_cols=TRUE and always when cens_dist is specified. In this case, both columns start with the nodes name and end with _status and _time. The first is a logical vector, the second a numeric one. If as_two_cols=FALSE and cens_dist is NULL, a numeric vector is returned instead.

Note

This function was updated internally in version 0.5.0 to make it faster and to allow the left argument. Generating data using this updated version will generally result in different results as compared to earlier versions, even when using the same random number generator seed. To replicate earlier results, please install earlier versions of this package.

Author(s)

Robin Denz

References

Bender R, Augustin T, Blettner M. Generating survival times to simulate Cox proportional hazards models. Statistics in Medicine. 2005; 24 (11): 1713-1723.

Examples

```
library(simDAG)

set.seed(3454)

# using a Cox model with a Weibull baseline hazard function
dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("death", type="cox", parents=c("sex", "age"), betas=c(1.1, 0.4),
       surv_dist="weibull", lambda=1.1, gamma=0.7, cens_dist="runif",
       cens_args=list(min=0, max=1))

sim_dat <- sim_from_dag(dag=dag, n_sim=1000)

## supplying a custom baseline hazard function

# some arbitrary baseline hazard function with two hills
fbasehaz <- function(t) {
  0.002 +
    0.01 * exp(-((t - 200)^2) / (2 * 50^2)) +   # first hill
    0.008 * exp(-((t - 700)^2) / (2 * 80^2))    # second hill
}

# some example DAG
dag <- empty_dag() +
  node(c("A", "B"), type="rbernoulli") +
  node("Y", type="cox", formula= ~ 0.5*A + -1.5*B, surv_dist=fbasehaz,
       basehaz_grid=1:100000, extrapolate=FALSE)

data <- sim_from_dag(dag, n_sim=100)
```

---

node_gamma *Generate Data from a Gamma Regression Model*

Description

Data from the parents is used to generate the node using gamma regression by applying the betas to the design matrix and sampling from the rgamma function.

Usage

```
node_gamma(data, parents, formula=NULL, betas,
           intercept, shape, link="log")
```

Arguments

data	A data.table (or something that can be coerced to a data.table) containing all columns specified by parents.
parents	A character vector specifying the names of the parents that this particular child node has. If non-linear combinations or interaction effects should be included, the user may specify the formula argument instead.
formula	An optional formula object to describe how the node should be generated or NULL (default). If supplied it should start with ~, having nothing else on the left hand side. The right hand side may contain any valid formula syntax, such as A + B or A + B + I(A^2), allowing non-linear effects. If this argument is defined, there is no need to define the parents argument. For example, using parents=c("A", "B") is equal to using formula= ~ A + B. Random effects and random slopes are currently not supported here.
betas	A numeric vector with length equal to parents, specifying the causal beta coefficients used to generate the node.
intercept	A single number specifying the intercept that should be used when generating the node.
shape	A single positive number specifying the shape parameter (shape argument in rgamma). The shape parameter determines the conditional variance of the generated values relative to their conditional mean.
link	The link function used to transform the linear predictor to the conditional mean used in rgamma. For a standard gamma regression model, this should be set to "log" (which is the default). Other allowed values are "identity" and "inverse".

Details

This function uses the linear predictor defined by the betas and the input design matrix to sample from a subject-specific gamma distribution. It does so by calculating the linear predictor using the data, betas and intercept, applying the inverse of the specified link function to obtain the conditional mean, and passing it to the rgamma function with shape=shape and scale=mu_i/shape of the **stats** package.

For an observation with conditional mean μ_i and shape parameter α, the generated outcome follows a gamma distribution with shape α and scale μ_i/α. Consequently, the conditional mean is $E(Y_i \mid X_i) = \mu_i$ and the conditional variance is $\mathrm{Var}(Y_i \mid X_i) = \mu_i^2/\alpha$.

Every non-missing conditional mean must be finite and strictly positive. This node type currently does not support inclusion of random effects or random slopes in the formula.

Value

Returns a numeric vector of length nrow(data).

Author(s)

Robin Denz

See Also

empty_dag, node, node_td, sim_from_dag, sim_discrete_time

Examples

```
library(simDAG)

set.seed(124554)

dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("cost", type="gamma", shape=3,
       formula= ~ 1.5 + age*0.01 + sexTRUE*0.2)

sim_dat <- sim_from_dag(dag=dag, n_sim=100, sort_dag=FALSE)
head(sim_dat)
```

---

node_gaussian *Generate Data from a (Mixed) Linear Regression Model*

Description

Data from the parents is used to generate the node using linear regression by predicting the covariate specific mean and sampling from a normal distribution with that mean and a specified standard deviation. Allows inclusion of arbitrary random effects and slopes.

Usage

```
node_gaussian(data, parents, formula=NULL, betas, intercept, error,
              var_corr=NULL, link="identity")
```

Arguments

data	A data.table (or something that can be coerced to a data.table) containing all columns specified by parents.
parents	A character vector specifying the names of the parents that this particular child node has. If non-linear combinations or interaction effects should be included, the user may specify the formula argument instead.
formula	An optional formula object to describe how the node should be generated or NULL (default). If supplied it should start with ~, having nothing else on the left hand side. The right hand side may contain any valid formula syntax, such as A + B or A + B + I(A^2), allowing non-linear effects. If this argument is defined, there is no need to define the parents argument. For example, using parents=c("A", "B") is equal to using formula= ~ A + B. May contain random effects and random slopes, in which case the **simr** package is used to generate the data. See details.

`betas`	A numeric vector with length equal to `parents`, specifying the causal beta coefficients used to generate the node.
`intercept`	A single number specifying the intercept that should be used when generating the node.
`error`	A single number specifying the sigma error that should be used when generating the node. By setting this argument to 0, the linear predictor is returned directly. If `formula` contains mixed model syntax, this argument is passed to the `sigma` argument of the `makeLmer` function of the **simr** package.
`var_corr`	Variances and covariances for random effects. Only used when `formula` contains mixed model syntax. If there are multiple random effects, their parameters should be supplied as a named list. More complex structures are also supported. This argument is directly passed to the `makeLmer` function of the **simr** package. Please consult the documentation of that package for more information on how mixed models should be specified. Some guidance can also be found in the "Issues" section of the official **simr** github page.
`link`	The link function used to transform the linear predictor before adding the random error to it. For a standard linear regression model, this should be set to `link="identity"` (which is the default). Other allowed values are `"log"` and `"inverse"`, which are defined the same way as in the classic `glm` function.

Details

Using the general linear regression equation, the observation-specific value that would be expected given the model is generated for every observation in the dataset generated thus far. We could stop here, but this would create a perfect fit for the node, which is unrealistic. Instead, we add an error term by taking one sample of a normal distribution for each observation with mean zero and standard deviation `error`. This error term is then added to the predicted mean.

Formal Description:

Formally, the data generation can be described as:

$$Y \sim \texttt{intercept} + \texttt{parents}_1 \cdot \texttt{betas}_1 + ... + \texttt{parents}_n \cdot \texttt{betas}_n + N(0, \texttt{error}),$$

where $N(0, \texttt{error})$ denotes the normal distribution with mean 0 and a standard deviation of `error` and n is the number of parents (`length(parents)`).

For example, given `intercept=-15`, `parents=c("A", "B")`, `betas=c(0.2, 1.3)` and `error=2` the data generation process is defined as:

$$Y \sim -15 + A \cdot 0.2 + B \cdot 1.3 + N(0, 2).$$

When using a `link` other than `"identity"`, the procedure is equivalent, except that the link function is applied to the linear predictor before adding the random error term. For example, when using `link="log"`, $exp(-15 + A \cdot 0.2 + B \cdot 1.3) + N(0, 2)$ is used instead.

Random Effects and Random Slopes:

This function also allows users to include arbitrary amounts of random slopes and random effects using the `formula` argument. If this is done, the `formula`, and `data` arguments are passed to

the variables of the same name in the makeLmer function of the **simr** package. The fixef argument of that function will be passed the numeric vector c(intercept, betas) and the VarCorr argument receives the var_corr argument as input. If used as a node type in a DAG, all of this is taken care of behind the scenes. Users can simply use the regular enhanced formula interface of the node function to define these formula terms, as shown in detail in the formula vignette (vignette(topic="v_using_formulas", package="simDAG")). Please consult that vignette for examples. Also, please note that inclusion of random effects or random slopes usually results in significantly longer computation times.

Value

Returns a numeric vector of length nrow(data).

Author(s)

Robin Denz

See Also

empty_dag, node, node_td, sim_from_dag, sim_discrete_time

Examples

```
library(simDAG)

set.seed(12455432)

# define a DAG
dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("bmi", type="gaussian", parents=c("sex", "age"),
       betas=c(1.1, 0.4), intercept=12, error=2)

# define the same DAG, but with a pretty formula for the child node
dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("bmi", type="gaussian", error=2,
       formula= ~ 12 + sexTRUE*1.1 + age*0.4)

sim_dat <- sim_from_dag(dag=dag, n_sim=100)

## an example using a random effect
if (requireNamespace("simr")) {

library(simr)

dag_mixed <- empty_dag() +
  node("School", type="rcategorical", probs=rep(0.1, 10),
       labels=LETTERS[1:10]) +
  node("Age", type="rnorm", mean=12, sd=2) +
```

```
  node("Grade", type="gaussian", formula= ~ -2 + Age*1.2 + (1|School),
       var_corr=0.3, error=1)

sim_dat <- sim_from_dag(dag=dag_mixed, n_sim=20)
}
```

---

node_identity *Generate Data based on an expression*

Description

This node type may be used to generate a new node given a regular R expression that may include function calls or any other valid R syntax. This may be useful to combine components of a node which need to be simulated with separate node calls, or just as a convenient shorthand for some variable transformations. Also allows calculation of just the linear predictor and generation of intermediary variables using the enhanced formula syntax.

Usage

```
node_identity(data, parents, formula, kind="expr",
              betas, intercept, var_names=NULL,
              name=NULL, dag=NULL)
```

Arguments

data	A data.table (or something that can be coerced to a data.table) containing all columns specified by parents.
parents	A character vector specifying the names of the parents that this particular child node has. When using this function as a node type in node or node_td, this argument usually does not need to be specified because the formula argument is required and contains all needed information already.
formula	A formula object. The specific way this argument should be specified depends on the value of the kind argument used. It can be an expression (kind="expr"), a simDAG style enhanced formula to calculate the linear predictor only (kind="linpred") or used as a way to store intermediary variable transformations (kind="data").
kind	A single character string specifying how the formula should be interpreted, with three allowed values: "expr", "linpred" and "data". If "expr" (default), the formula should contain a ~ symbol with nothing on the LHS, and any valid R expression that can be evaluated on data on the RHS. This expression needs to contain at least one variable name (otherwise users may simply use rconstant as node type). It may contain any number of function calls or other valid R syntax, given that all contained objects are included in the global environment. Note that the usual formula syntax, using for example A:B*0.2 to specify an interaction won't work in this case. If that is the goal, users should use kind="linpred", in which case the formula is interpreted in the normal simDAG way and the linear combination of the variables is calculated. Finally, if kind="data", the formula may contain any enhanced formula syntax, such as

	A:B or net() calls, but it should not contain beta-coefficients or an intercept. In this case, the transformed variables are returned in the order given, using the name as column names. See examples.
betas	Only used internally when kind="linpred".
intercept	Only used internally when kind="linpred". If no intercept should be present, it should still be added to the formula using a simple 0, for example ~ 0 + A*0.2 + B*0.3
var_names	Only used when kind="data". In this case, and only if there are multiple terms on the right-hand side of formula, the resulting columns will be re-named according to this argument. Should have the same length as the number of terms in formula. Names are given in the same order as the variables appear in formula. If only a single term is on the right-hand side of formula, the name supplied in the node function call will automatically be used as the nodes name and this argument is ignored. Set to NULL (default) to just use the terms as names.
name	A single character string, specifying the name of the node. Passed internally only. See var_names.
dag	The dag that this node is a part of. Will be passed internally if needed (for example when performing networks-based simulations). This argument can therefore always be ignored by users.

Details

When using kind="expr", custom functions and objects can be used without issues in the formula, but they need to be present in the global environment, otherwise the underlying eval() function call will fail. Using this function outside of node or node_td is essentially equal to using with(data, eval(formula)) (without the ~ in the formula). If kind!="expr", this function cannot be used outside of a defined DAG.

Please note that when using identity nodes with kind="data" and multiple terms in formula, the printed structural equations and plots of a dag object may not be correct.

Value

Returns a numeric vector of length nrow(data).

Author(s)

Robin Denz

See Also

empty_dag, node, node_td, sim_from_dag, sim_discrete_time

Examples

```
library(simDAG)

set.seed(12455432)
```

```
#### using kind = "expr" ####

# define a DAG
dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("something", type="identity", formula= ~ age + sex + 2)

sim_dat <- sim_from_dag(dag=dag, n_sim=100)
head(sim_dat)

# more complex alternative
dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("something", type="identity",
       formula= ~ age / 2 + age^2 - ifelse(sex, 2, 3) + 2)

sim_dat <- sim_from_dag(dag=dag, n_sim=100)
head(sim_dat)

#### using kind = "linpred" ####

# this would work with both kind="expr" and kind="linpred"
dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("pred", type="identity", formula= ~ 1 + age*0.2 + sex*1.2,
       kind="linpred")

sim_dat <- sim_from_dag(dag=dag, n_sim=10)
head(sim_dat)

# this only works with kind="linpred", due to the presence of a special term
dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5, output="numeric") +
  node("pred", type="identity", formula= ~ 1 + age*0.2 + sex*1.2 + age:sex*-2,
       kind="linpred")

sim_dat <- sim_from_dag(dag=dag, n_sim=10)
head(sim_dat)

#### using kind = "data" ####

# simply return the transformed data, useful if the terms are used
# frequently in multiple nodes in the DAG to save computation time

# using only a single interaction term
dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5, output="numeric") +
  node("age_sex_interact", type="identity", formula= ~ age:sex, kind="data")
```

```
sim_dat <- sim_from_dag(dag=dag, n_sim=10)
head(sim_dat)

# using multiple terms
dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5, output="numeric") +
  node("name_not_used", type="identity", formula= ~ age:sex + I(age^2),
       kind="data", var_names=c("age_sex_interact", "age_squared"))

sim_dat <- sim_from_dag(dag=dag, n_sim=10)
head(sim_dat)
```

`node_mixture` *Generate Data from a Mixture of Node Definitions*

Description

This node type allows users to apply different nodes to different subsets of the already generated data, making it possible to generate data for arbitrary mixture distributions. It is similar to `node_conditional_distr` and `node_conditional_prob`, with the main difference being that the former only allow univariate distributions conditional on categorical variables, while this function allows any kind of node definition and condition. This makes it, for example, possible to generate data for a variable from different regression models for different subsets of simulated individuals.

Usage

```
node_mixture(data, parents, name, distr, default=NA)
```

Arguments

`data`	A `data.table` (or something that can be coerced to a `data.table`) containing all columns specified by `parents`.
`parents`	A character vector specifying the names of the parents that this particular child node has. This vector should include all nodes that are used in the conditions and the `node` calls specified in `distr`.
`name`	A single character string specifying the name of the node.
`distr`	A unnamed list that specifies both the conditions and the `node` definitions. It should be specified in a similar way as the `fcase` function in pairs of conditions (coded as strings) and `node` definitions. This means that a condition comes first, for example `"A==0"`, followed by some call `node` and so on. Arbitrary numbers of those pairs are allowed with no restrictions to what can be specified in the `node` calls. The `name` argument has to be specified in all `node` calls, but it does not matter which value is used as it will be ignored in further processing. Currently only supports time-fixed nodes defined using the `node` function, not time-dependent nodes defined using the `node_td` function. See examples.
`default`	A single value of some kind, used as a default value for those individuals not covered by all the conditions defined in `distr`. Defaults to `NA`.

Details

Internally, the data is generated by extracting only the relevant part of the already generated `data` as defined by the condition and using `node` function to generate the new response-part. This generation is done in the order in which the `distr` was specified, meaning that data for the first condition is checked first and so on. There are no safeguards to guarantee that the conditions do not overlap. For example, users are free to set the first condition to something like `A > 10` and the next one to `A > 11`, in which case the value for every individual with `A > 11` is generated twice (first with the first specification, secondly with the next specification). In this case, only the last generated value is retained.

Note that it is also possible to use the mixture node itself inside the conditions or `node` calls in `distr`, because it is directly added to the `data` before the first condition is applied (by setting everyone to the `default` value). See examples.

Additionally, because the output of each of the parts of the mixture distributions is forced into one vector, they might be coerced from one class to another, depending on the input to `distr` and the order used. This also needs to be taken care of by the user.

Value

Returns a vector of length `nrow(data)`. The class of the vector is determined by what is specified in `distr`.

Author(s)

Robin Denz

See Also

`empty_dag`, `node`, `node_td`, `sim_from_dag`, `sim_discrete_time`

Examples

```
library(simDAG)

set.seed(1234)

## different linear regression models per level of a different covariate
# here, A is the group that is used for the conditioning, B is a predictor
# and Y is the mixture distributed outcome
dag <- empty_dag() +
  node("A", type="rbernoulli") +
  node("B", type="rnorm") +
  node("Y", type="mixture", parents="A",
       distr=list(
         "A==0", node(".", type="gaussian", formula= ~ -2 + B*2, error=1),
         "A==1", node(".", type="gaussian", formula= ~ 3 + B*5, error=1)
       ))
data <- sim_from_dag(dag, n_sim=100)
head(data)

# also works with multiple conditions
```

```
dag <- empty_dag() +
  node(c("A", "C"), type="rbernoulli") +
  node("B", type="rnorm") +
  node("Y", type="mixture", parents=c("A", "C"),
    distr=list(
      "A==0 & C==1", node(".", type="gaussian", formula= ~ -2 + B*2, error=1),
      "A==1", node(".", type="gaussian", formula= ~ 3 + B*5, error=1)
    ))
data <- sim_from_dag(dag, n_sim=100)
head(data)

# using the mixture node itself in the condition
# see cookbook vignette, section on outliers for more info
dag <- empty_dag() +
  node(c("A", "B", "C"), type="rnorm") +
  node("Y", type="mixture", parents=c("A", "B", "C"),
       distr=list(
         "TRUE", node(".", type="gaussian", formula= ~ -2 + A*0.1 + B*1 + C*-2,
                      error=1),
         "Y > 2", node(".", type="rnorm", mean=10000, sd=500)
       ))
data <- sim_from_dag(dag, n_sim=100)
```

`node_multinomial` *Generate Data from a Multinomial Regression Model*

Description

Data from the parents is used to generate the node using multinomial regression by predicting the covariate specific probability of each class and sampling from a multinomial distribution accordingly.

Usage

```
node_multinomial(data, parents, betas, intercepts,
                 labels=NULL, output="factor",
                 return_prob=FALSE)
```

Arguments

`data`	A `data.table` (or something that can be coerced to a `data.table`) containing all columns specified by `parents`.
`parents`	A character vector specifying the names of the parents that this particular child node has.
`betas`	A numeric matrix with `length(parents)` columns and one row for each class that should be simulated, specifying the causal beta coefficients used to generate the node.

intercepts	A numeric vector with one entry for each class that should be simulated, specifying the intercepts used to generate the node.
labels	An optional character vector giving the factor levels of the generated classes. If NULL (default), the integers are simply used as factor levels.
output	A single character string specifying the output format. Must be one of "factor" (default), "character" or "numeric". If the argument labels is supplied, the output will coerced to "character" by default.
return_prob	Either TRUE or FALSE (default). Specifies whether to return the matrix of class probabilities or not. If you are using this function inside of a node call, you cannot set this to TRUE because it will return a matrix. It may, however, be useful when using this function by itself, or as a probability generating function for the node_competing_events function.

Details

This function works essentially like the node_binomial function. First, the matrix of betas coefficients is used in conjunction with the values defined in the parents nodes and the intercepts to calculate the expected subject-specific probabilities of occurrence for each possible category. This is done using the standard multinomial regression equations. Using those probabilities in conjunction with the rcategorical function, a single one of the possible categories is drawn for each individual.

When actually fitting a multinomial regression model (with functions such as multinom from the **nnet** package), the coefficients will usually not be equal to the ones supplied in betas. The reason is that these functions usually standardize the coefficients to the coefficient of the reference category.

Value

Returns a vector of length nrow(data). Depending on the used arguments, this vector may be of type character, numeric of factor. If return_prob was used it instead returns a numeric matrix containing one column per possible event and nrow(data) rows.

Author(s)

Robin Denz

See Also

empty_dag, node, node_td, sim_from_dag, sim_discrete_time

Examples

```
library(simDAG)

set.seed(3345235)

dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("UICC", type="multinomial", parents=c("sex", "age"),
```

```
        betas=matrix(c(0.2, 0.4, 0.1, 0.5, 1.1, 1.2), ncol=2),
        intercepts=1)

sim_dat <- sim_from_dag(dag=dag, n_sim=100)
```

---

node_negative_binomial *Generate Data from a Negative Binomial Regression Model*

Description

Data from the parents is used to generate the node using negative binomial regression by applying the betas to the design matrix and sampling from the rnbinom function.

Usage

```
node_negative_binomial(data, parents, formula=NULL, betas,
                       intercept, theta, link="log")
```

Arguments

data	A data.table (or something that can be coerced to a data.table) containing all columns specified by parents.
parents	A character vector specifying the names of the parents that this particular child node has. If non-linear combinations or interaction effects should be included, the user may specify the formula argument instead.
formula	An optional formula object to describe how the node should be generated or NULL (default). If supplied it should start with ~, having nothing else on the left hand side. The right hand side may contain any valid formula syntax, such as A + B or A + B + I(A^2), allowing non-linear effects. If this argument is defined, there is no need to define the parents argument. For example, using parents=c("A", "B") is equal to using formula= ~ A + B. Contrary to the node_gaussian, node_binomial and node_poisson node types, random effects and random slopes are currently not supported here.
betas	A numeric vector with length equal to parents, specifying the causal beta coefficients used to generate the node.
intercept	A single number specifying the intercept that should be used when generating the node.
theta	A single number specifying the theta parameter (size argument in rnbinom).
link	The link function used to transform the linear predictor to the mu value used in rnbinom. For a standard negative binomial regression model, this should be set to "log" (which is the default). Other allowed values are "identity" and "sqrt".

Details

This function uses the linear predictor defined by the `betas` and the input design matrix to sample from a subject-specific negative binomial distribution. It does to by calculating the linear predictor using the `data`, `betas` and `intercept`, applying the inverse of the link function to it and passing it to the `mu` argument of the `rnbinom` function of the **stats** package.

This node type currently does not support inclusion of random effects or random slopes in the `formula`.

Value

Returns a numeric vector of length `nrow(data)`.

Author(s)

Robin Denz

See Also

`empty_dag`, `node`, `node_td`, `sim_from_dag`, `sim_discrete_time`

Examples

```
library(simDAG)

set.seed(124554)

dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("smoking", type="negative_binomial", theta=0.05,
       formula= ~ -2 + sexTRUE*1.1 + age*0.4)

sim_dat <- sim_from_dag(dag=dag, n_sim=100, sort_dag=FALSE)
```

---

node_next_time *Generate the Next Time of an Event in Discrete-Event Simulation*

Description

This node essentially models a dichotomous time-dependent variable for which the time of the event will be important for later usage. Can only be used inside of the `sim_discrete_event` function, not outside of it or in other simulation functions. See details.

Usage

```
node_next_time(data, formula, prob_fun, ...,
               distr_fun=rtexp, distr_fun_args=list(),
               model=NULL, event_duration=Inf,
               immunity_duration=event_duration,
               event_count=FALSE)
```

Arguments

data	A data.table containing all columns specified by parents. Similar objects such as data.frames are not supported.
formula	An optional formula that may be used to define the probability that will be passed to distr_fun using a binomial regression model. If supplied, the node_binomial function will be called internally with return_prob=TRUE (calculating only the probability as estimated using the model). See ?node or the associated vignette for more information about how a formula should be defined in this package. Note that net terms are currently not supported in this node type. This argument is ignored if prob_fun is specified.
prob_fun	A function that returns a numeric vector of size nrow(data). These numbers should be used to summarise the effect of the considered covariates per person. The summarised score is then used in the distr_fun function call that follows internally. For example, when using distr_fun="rtexp" (default), the prob_fun should generate the person-specific probability of experiencing the event during 1 time unit. Any function may be used, as long as it has a named argument called data. Alternatively this argument can be set to a single number, resulting in a fixed summary score being used for every simulated individual at every point in time. The formula argument may be used as a convenient alternative if users want to specify a binomial regression model.
...	An arbitrary amount of additional named arguments passed to prob_fun if prob_fun is specified. If formula is specified and both prob_fun and model are not, these additional arguments are passed directly to the node_binomial function instead. If model is specified, these arguments are passed to the respective model function. Ignore this if you do not want to pass any arguments. Also ignored if prob_fun is a single number.
distr_fun	A function that returns the (left-truncated) next time at which the variable turns to TRUE. Any function that has at least three named arguments n (the number of times to draw), rate (the summary score returned by prob_fun) and l (the time of left-truncation) may be used. The function additionally needs to be vectorised over both rate and l, so that a vector of different values may be supplied. The left-truncation is required, so that it only generated times that are strictly larger than l. A classic example for such a function is the rtexp function (the default). See examples and the associated vignette.
distr_fun_args	A list of named arguments that should be passed to the function specified in the distr_fun argument.
model	Alternative way to specify how the next time should be generated. Takes a single character string, specifying a time-to-event node or NULL (default) to not use

this functionality. Currently supported values are "cox", "aalen", "aftreg", "ahreg", "ehreg", "poreg" and "ypreg". The node function corresponding to paste0("node_", model) is used then, so for example using model="cox" means that the node_cox function is used to generate the next times. If this argument is specified, both prob_fun and distr_fun are ignored. Concurrent use of the formula argument is supported. Further arguments that need to be passed to the respective node function can be passed through the ... syntax.

event_duration A single number > 0 specifying how long the event should last. During this period, the corresponding variable is set to TRUE.

immunity_duration
A single number >= event_duration specifying how long the person should be immune to the event after it is over. The count internally starts when the event starts, so in order to use an immunity duration of 10 time units after the event is over event_duration + 10 should be used. The corresponding variable is set to FALSE after the event_duration is up and until the immunity_duration is over.

event_count Either TRUE or FALSE (default), specifying whether an additional column should be added that counts the number of times this variable has been TRUE previously. If TRUE, the column will be named by taking the name of the node and appending "_event_count". It is 0 at the beginning and increases by 1 at each point in time that the variable changes its status from FALSE to TRUE. Note that this may increase the time it takes to run the simulation. If the count of previous events is only needed for processing in the variable itself, a faster alternative is to keep this argument at FALSE and to use the internal .event_count column instead. Only use this argument if other variables should be dependent on the event count of this variable.

Details

This function is the only time-dependent node type that may currently be used when conducting discrete-event simulations using the sim_discrete_event function. It is very similar to the node_time_to_event function in spirit, as it is used to model a binary variable over time. It is, however, not usable in sim_discrete_time calls. Use the node_time_to_event function there instead.

How it works:

At the beginning ($t = 0$) of the simulation, any variable added using this function is set to FALSE for all individuals. Then, the function supplied in the prob_fun argument is applied to all individuals in the current data, potentially using information from baseline covariates and other time-dependent nodes (the latter of which are all FALSE at this stage). The obtained summary score is then passed to the distr_fun in order to generate the time at which the variable changes from FALSE to TRUE for each individual.

For example, consider the situation in which only one time-dependent variable is included. In this case, the simulation time for each individual jumps to the generated event time immediately. The variable is then set to TRUE. Afterwards, the simulation time jumps until the end of the event_duration (if that duration is Inf, the simulation is over). The variable is then set back to FALSE. Next, the simulation time jumps to the end of the immunity_duration (again, if this is Inf, the simulation is over).

With more then one time-dependent variable, the situation is a little more complicated. Consider two time-dependent variables A and B. At $t = 0$, both are FALSE for every individual. The prob_fun and subsequently the distr_fun of both variables are called to generate the time of the next event in each of them. Lets say those are 20 and 42, respectively. The simulation time is then advanced to 20, setting A to TRUE. At this point, the prob_fun and distr_fun arguments are called again for B, because B might be dependent on current values of A, drawing a new next event time for B. Crucially, this time is drawn from a left-truncated distr_fun, so that it is always larger than the current time of 20. Lets say that new time is 53.

The simulation is then advanced again, but not necessarily to 53. Lets say the event_duration of A is only 10. In this case the simulation time is only advanced to 30. A is then set to FALSE again and the next time for B is re-computed using its prob_fun and distr_fun. At this point, if the immunity_duration of A is not Inf, the next time for A is also re-computed, left-truncated on the current simulation time + the immunity_duration. Again, the time is advanced to the next event and the cycle continues.

This process is repeated until either (1) all variables reach a terminal state, (2) the simulation time for each individual is >= max_t, (3) a break condition defined by break_if is reached or (4) no individuals are left after their removal through the remove_if argument. Otherwise, the simulation runs forever.

What can be done with it:

This type of node naturally supports the implementation of terminal and recurrent events that may be influenced by baseline variables and other such events over time dynamically. By specifying the parents and prob_fun arguments correctly, it is possible to create an event type that is dependent on past events of itself or other time-to-event variables and other variables in general, allowing non-markovian data to be generated. The user can include any amount of these nodes in their simulation. It may also be used to simulate any kind of binary time-dependent variable that one would usually not associate with the name "event" as well.

What can't be done with it:

Currently this function only allows binary events. Categorical event types or continuous time-dependent variables are currently not supported. The event_duration and immunity_duration can also only be fixed for each node, and are not allowed to vary per person.

Value

This function is never actually called. It is only used so that the node type "next_time" can be safely specified in node_td calls. It does not make sense to ever use it outside a node_td call, as it always returns NULL.

Author(s)

Robin Denz

See Also

empty_dag, node_td, sim_discrete_event, node_time_to_event

Examples

```
library(simDAG)

## a simple terminal time-to-event node, with a constant probability of
## occurrence, independent of any other variable
dag <- empty_dag() +
  node_td("death", type="next_time", prob_fun=0.0001,
          event_duration=Inf)

## a simple recurrent time-to-event node with a constant probability of
## occurrence, independent of any other variable
dag <- empty_dag() +
  node_td("car_crash", type="next_time", prob_fun=0.001, event_duration=1)

## a next-time node with a probability function dependent on a
## time-fixed variable
prob_car_crash <- function(data) {
  ifelse(data$sex==1, 0.001, 0.01)
}

dag <- empty_dag() +
  node("sex", type="rbernoulli", p=0.5) +
  node_td("car_crash", type="next_time", prob_fun=prob_car_crash,
          parents="sex")

## a little more complex car crash simulation, where the probability for
## a car crash is dependent on the sex, and the probability of death is
## highly increased for 3 days after a car crash happened
prob_car_crash <- function(data) {
  ifelse(data$sex==1, 0.001, 0.01)
}

prob_death <- function(data) {
  ifelse(data$car_crash, 0.1, 0.0001)
}

dag <- empty_dag() +
  node("sex", type="rbernoulli", p=0.5) +
  node_td("car_crash", type="next_time", prob_fun=prob_car_crash,
          parents="sex", event_duration=3) +
  node_td("death", type="next_time", prob_fun=prob_death,
          parents="car_crash", event_duration=Inf)

# use the sim_discrete_time function to simulate data from one of these DAGs:
sim <- sim_discrete_event(dag, n_sim=20, max_t=500)

## more examples can be found in the vignettes of this package
```

---

node_poisson *Generate Data from a (Mixed) Poisson Regression Model*

Description

Data from the parents is used to generate the node using poisson regression by predicting the covariate specific lambda and sampling from a poisson distribution accordingly. Allows inclusion of arbitrary random effects and slopes.

Usage

```
node_poisson(data, parents, formula=NULL, betas, intercept,
             var_corr=NULL, link="log")
```

Arguments

`data`	A `data.table` (or something that can be coerced to a `data.table`) containing all columns specified by `parents`.
`parents`	A character vector specifying the names of the parents that this particular child node has. If non-linear combinations or interaction effects should be included, the user may specify the `formula` argument instead.
`formula`	An optional `formula` object to describe how the node should be generated or `NULL` (default). If supplied it should start with `~`, having nothing else on the left hand side. The right hand side may contain any valid formula syntax, such as `A + B` or `A + B + I(A^2)`, allowing non-linear effects. If this argument is defined, there is no need to define the `parents` argument. For example, using `parents=c("A", "B")` is equal to using `formula= ~ A + B`. May contain random effects and random slopes, in which case the **simr** package is used to generate the data. See details.
`betas`	A numeric vector with length equal to `parents`, specifying the causal beta coefficients used to generate the node.
`intercept`	A single number specifying the intercept that should be used when generating the node.
`var_corr`	Variances and covariances for random effects. Only used when `formula` contains mixed model syntax. If there are multiple random effects, their parameters should be supplied as a named list. More complex structures are also supported. This argument is directly passed to the `makeLmer` function of the **simr** package. Please consult the documentation of that package for more information on how mixed models should be specified. Some guidance can also be found in the "Issues" section of the official **simr** github page.
`link`	The link function used to transform the linear predictor to the `lambda` value used in `rpois`. For a standard Poisson regression model, this should be set to `"log"` (which is the default). Other allowed values are `"identity"` and `"sqrt"`, which are defined the same way as in the classic `glm` function.

Details

Essentially, this function simply calculates the linear predictor defined by the `betas`-coefficients, the `intercept` and the values of the `parents`. The `link` function is then applied to this predictor and the result is passed to the `rpois` function. The result is a draw from a subject-specific poisson distribution, resembling the user-defined poisson regression model.

Formal Description:

Formally, the data generation (using `link="log"`) can be described as:

$$Y \sim Poisson(\lambda),$$

where $Poisson()$ means that the variable is Poisson distributed with:

$$P_\lambda(k) = \frac{\lambda^k e^{-\lambda}}{k!}.$$

Here, k is the count and e is eulers number. The parameter λ is determined as:

$$\lambda = \exp(\texttt{intercept} + \texttt{parents}_1 \cdot \texttt{betas}_1 + ... + \texttt{parents}_n \cdot \texttt{betas}_n),$$

where n is the number of parents (`length(parents)`).

For example, given `intercept=-15`, `parents=c("A", "B")`, `betas=c(0.2, 1.3)` the data generation process is defined as:

$$Y \sim Poisson(\exp(-15 + A \cdot 0.2 + B \cdot 1.3)).$$

Random Effects and Random Slopes:

This function also allows users to include arbitrary amounts of random slopes and random effects using the `formula` argument. If this is done, the `formula`, and `data` arguments are passed to the variables of the same name in the `makeGlmer` function of the **simr** package. The `fixef` argument of that function will be passed the numeric vector `c(intercept, betas)` and the `VarCorr` argument receives the `var_corr` argument as input. If used as a node type in a `DAG`, all of this is taken care of behind the scenes. Users can simply use the regular enhanced formula interface of the `node` function to define these formula terms, as shown in detail in the formula vignette (`vignette(topic="v_using_formulas", package="simDAG")`). Please consult that vignette for examples. Also, please note that inclusion of random effects or random slopes usually results in significantly longer computation times.

Value

Returns a numeric vector of length `nrow(data)`.

Author(s)

Robin Denz

See Also

`empty_dag`, `node`, `node_td`, `sim_from_dag`, `sim_discrete_time`

Examples

```
library(simDAG)

set.seed(345345)

dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("smoking", type="poisson",
       formula= ~ -2 + sexTRUE*1.1 + age*0.4)

sim_dat <- sim_from_dag(dag=dag, n_sim=100)

## an example using a random effect
if (requireNamespace("simr")) {

library(simr)

dag_mixed <- empty_dag() +
  node("School", type="rcategorical", probs=rep(0.1, 10),
       labels=LETTERS[1:10]) +
  node("Age", type="rnorm", mean=12, sd=2) +
  node("Grade", type="poisson", formula= ~ -2 + Age*1.2 + (1|School),
       var_corr=0.3)

sim_dat <- sim_from_dag(dag=dag_mixed, n_sim=20)
}
```

---

node_polr *Generate Data from an Ordered Logistic or Probit Regression*

Description

Data from the parents is used to generate the node using by first calculating the linear predictor. An appropriately distributed random error term is then added. The resulting latent variable is categorized at user-specified cutpoints to obtain the resulting ordered factor.

Usage

```
node_polr(data, parents, formula=NULL, betas,
          cutpoints, link="logistic", labels=FALSE,
          output="factor")
```

Arguments

data	A data.table (or something that can be coerced to a data.table) containing all columns specified by parents.

parents	A character vector specifying the names of the parents that this particular child node has. If non-linear combinations or interaction effects should be included, the user may specify the formula argument instead.
formula	An optional formula object to describe how the node should be generated or NULL (default). If supplied it should start with ~, having nothing else on the left hand side. The right hand side may contain any valid formula syntax, such as A + B or A + B + I(A^2), allowing non-linear effects. If this argument is defined, there is no need to define the parents argument. For example, using parents=c("A", "B") is equal to using formula= ~ A + B. Contrary to the node_gaussian, node_binomial and node_poisson node types, random effects and random slopes are currently not supported here.
betas	A numeric vector with length equal to parents, specifying the causal beta coefficients used to generate the node.
cutpoints	A numeric vector containing the points at which the underlying latent variable is cut to produce the ordered factor. In general, length(cutpoints) + 1 categories will be produced.
link	The link function describing the distribution of the error term. Allowed values are "logistic", "probit", "loglog", "cloglog" and "cauchit", mirroring the method argument in the polr function.
labels	Labels for the levels of the resulting categories. By default (labels = FALSE), simple integer codes are returned. Corresponds to the argument of the same name in the cut function.
output	A single character string specifying the kind of output that should be returned. Defaults to "factor", but may also be set to "character".

Details

This node type currently does not support inclusion of random effects or random slopes in the formula.

Value

Returns a character of factor vector of length nrow(data).

Author(s)

Robin Denz

See Also

empty_dag, node, node_td, sim_from_dag, sim_discrete_time

Examples

```
library(simDAG)

set.seed(124554)
```

```
dag <- empty_dag() +
  node(c("X1", "X2"), type="rnorm", mean=0, sd=1) +
  node("Y", type="polr", formula= ~ -2*X1 + 1.5*X2, cutpoints=c(0.7, 1.5, 3.2),
       link="logistic")

sim_dat <- sim_from_dag(dag=dag, n_sim=100)
```

---

node_rsurv *Generate Data from Parametric Survival Models*

Description

Data from the parents is used to generate the node using either an accelerated failure time model, an accelerated hazard model, an extended hazard model, a proportional odds model or a Young and Prentice model, as implemented in the **rsurv** package (Demarqui 2024).

Usage

```
node_aftreg(data, parents, betas, baseline, dist=NULL,
            package=NULL, u=stats::runif(nrow(data)),
            cens_dist=NULL, cens_args, name, as_two_cols=TRUE,
            left=0, right=Inf, ...)

node_ahreg(data, parents, betas, baseline, dist=NULL,
           package=NULL, u=stats::runif(nrow(data)),
           cens_dist=NULL, cens_args, name, as_two_cols=TRUE,
           left=0, right=Inf, ...)

node_ehreg(data, parents, betas, phi, baseline, dist=NULL,
           package=NULL, u=stats::runif(nrow(data)),
           cens_dist=NULL, cens_args, name, as_two_cols=TRUE,
           left=0, right=Inf, ...)

node_poreg(data, parents, betas, baseline, dist=NULL,
           package=NULL, u=stats::runif(nrow(data)),
           cens_dist=NULL, cens_args, name, as_two_cols=TRUE,
           left=0, right=Inf, ...)

node_ypreg(data, parents, betas, phi, baseline, dist=NULL,
           package=NULL, u=stats::runif(nrow(data)),
           cens_dist=NULL, cens_args, name, as_two_cols=TRUE,
           left=0, right=Inf, ...)
```

Arguments

data	A data.table (or something that can be coerced to a data.table) containing all columns specified by parents. Passed to the argument of the same name in raftreg, rahreg, rehreg, rporeg or rypreg.

parents — A character vector specifying the names of the parents that this particular child node has. Converted into a formula and passed to the argument of the same name in raftreg, rahreg, rehreg, rporeg or rypreg.

betas — A numeric vector with length equal to parents, specifying the causal beta coefficients used to generate the node. Passed to the beta argument in raftreg, rahreg, rehreg, rporeg or rypreg.

phi — A numeric vector with length equal to parents, specifying the phi beta coefficients used to generate the node. Only required for extended hazard and Yang and Prentice models. Passed to the phi argument in rehreg or rypreg.

baseline — A single character string, specifying the name of the baseline survival distribution. Passed to the argument of the same name in raftreg, rahreg, rehreg, rporeg or rypreg.

dist — An alternative way to specify the baseline survival distribution. Passed to the argument of the same name in raftreg, rahreg, rehreg, rporeg or rypreg.

package — A single character string, specifying the name of the package where the assumed quantile function is implemented. Passed to the argument of the same name in raftreg, rahreg, rehreg, rporeg or rypreg.

u — A numeric vector of quantiles of length nrow(data). Usually this should simply be passed a vector of randomly generated uniformly distributed values between 0 and 1, as defined by the default. Passed to the argument of the same name in raftreg, rahreg, rehreg, rporeg or rypreg.

cens_dist — A single character naming the distribution function that should be used to generate the censoring times or a suitable function. For example, "runif" could be used to generate uniformly distributed censoring times. Set to NULL to get no censoring.

cens_args — A list of named arguments which will be passed to the function specified by the cens_dist argument.

name — A single character string specifying the name of the node.

as_two_cols — Either TRUE or FALSE, specifying whether the output should be divided into two columns. When cens_dist is specified, this argument will always be treated as TRUE because two columns are needed to encode both the time to the event and the status indicator. When no censoring is applied, however, users may set this argument to FALSE to simply return the numbers as they are.

left — Either a single number >= 0, or a numeric vector of length nrow(data) containing only numbers >= 0. When simulating the survival times, only numbers larger than these will be generated using correctly left-truncated sampling. Note that this does not affect the censoring times, only the generated survival times will be left-truncated. Set to 0 (default) to not use left-truncation. Corresponds to the lwr argument in the underlying **rsurv** functions. Only works with rsurv package version >= 0.0.3 (currently only available on github).

right — Same as the left argument, but for right-truncation, meaning that no values greater than right will be generated. Corresponds to the upr argument in the underlying **rsurv** functions. Only works with rsurv package version >= 0.0.3 (currently only available on github).

... — Further arguments passed to raftreg, rahreg, rehreg, rporeg or rypreg.

Details

Survival times are generated according to the specified parametric survival model. The actual generation of the values is done entirely by calls to the **rsurv** package. All arguments are directly passed to the corresponding function in **rsurv**. Only the censoring is added on top of it. These convenience wrappers only exist to allow direct integration of these data generation functions with the interface provided by **simDAG**. Please consult the documentation and excellent paper by Demarqui (2024) for more information about the models and how to specify the arguments.

Value

Returns a `data.table` of length `nrow(data)` containing two columns if `as_two_cols=TRUE` and always when `cens_dist` is specified. In this case, both columns start with the nodes `name` and end with `_event` and `_time`. The first is a logical vector, the second a numeric one. If `as_two_cols=FALSE` and `cens_dist` is `NULL`, a numeric vector is returned instead.

Author(s)

Robin Denz

References

Demarqui Fabio N. Simulation of Survival Data with the Package rsurv. (2024) arXiv:2406.01750v1.

Examples

```
library(simDAG)

set.seed(3454)

if (requireNamespace("rsurv")) {

library(rsurv)

# accelerated failure time model
dag <- empty_dag() +
  node(c("A", "B", "C"), type="rnorm", mean=0, sd=1) +
  node("Y", type="aftreg", formula= ~ -2 + A*0.2 + B*0.1 + A:B*1,
       baseline="weibull", shape=1, scale=2)
data <- sim_from_dag(dag, n_sim=100)

# accelerated hazard model
dag <- empty_dag() +
  node(c("A", "B", "C"), type="rnorm", mean=0, sd=1) +
  node("Y", type="ahreg", formula= ~ -2 + A*0.2 + B*0.1,
       baseline="weibull", shape=1, scale=2)
data <- sim_from_dag(dag, n_sim=100)

# extended hazard model
dag <- empty_dag() +
  node(c("A", "B", "C"), type="rnorm", mean=0, sd=1) +
  node("Y", type="ehreg", formula= ~ -2 + A*0.2 + B*0.1,
```

```
      baseline="weibull", shape=1, scale=2,
      phi=c(-1, 1))
data <- sim_from_dag(dag, n_sim=100)

# proportional odds model
dag <- empty_dag() +
  node(c("A", "B", "C"), type="rnorm", mean=0, sd=1) +
  node("Y", type="poreg", formula= ~ -2 + A*0.2 + B*0.1,
       baseline="weibull", shape=1, scale=2)
data <- sim_from_dag(dag, n_sim=100)

# Young and Prentice model
dag <- empty_dag() +
  node(c("A", "B", "C"), type="rnorm", mean=0, sd=1) +
  node("Y", type="ypreg", formula= ~ -2 + A*0.2 + B*0.1,
       baseline="weibull", shape=1, scale=2,
       phi=c(-1, 1))
data <- sim_from_dag(dag, n_sim=100)
}
```

`node_time_to_event` *Generate Data from repeated Bernoulli Trials in Discrete-Time Simulation*

Description

This node essentially models a dichotomous time-dependent variable for which the time of the event will be important for later usage. It adds two columns to `data`: `name_event` (whether the person currently has an event) and `name_time` (the time at which the current event started). Past events are stored in a list. Can only be used inside of the `sim_discrete_time` function, not outside of it. See details.

Usage

```
node_time_to_event(data, parents, sim_time, past_states, name,
                   formula, prob_fun=NULL, ..., event_duration=1,
                   immunity_duration=event_duration, unif=NULL,
                   time_since_last=FALSE, event_count=FALSE,
                   save_past_events=TRUE, check_inputs=TRUE,
                   envir)
```

Arguments

`data`	A `data.table` containing all columns specified by `parents`. Similar objects such as `data.frames` are not supported.
`parents`	A character vector specifying the names of the parents that this particular child node has. Those child nodes should be valid column names in `data`. Because the state of this variable is by definition dependent on its previous states, the columns produced by this function will automatically be considered its parents without the user having to manually specify this.

`sim_time` The current time of the simulation. If `sim_time` is an argument in the function passed to the `prob_fun` argument, this time will automatically be passed to it as well.

`past_states` A list of `data.tables` including previous states of the simulation. This argument cannot be specified directly by the user. Instead, it is passed to this function internally whenever a function is passed to the `prob_fun` argument which includes a named argument called `past_states`. May be useful to specify nodes that are dependent on specific past states of the simulation.

`name` The name of the node. This will be used as prefix before the `_event`, `_time` columns. If the `time_since_last` or `event_count` arguments are set to `TRUE`, this will also be used as prefix for those respective columns.

`formula` An optional enhanced formula, as used throughout the package. This may be used instead of the `prob_fun` argument, to specify a binomial regression model that should be used to calculate the probability instead. If specified (and `prob_fun=NULL`), the `node_binomial` function is used with `return_prob=TRUE` to obtain the probabilities. If `prob_fun` is specified, this argument is ignored.

`prob_fun` A function that returns a numeric vector of size `nrow(data)` containing only numbers between 0 and 1. These numbers specify the person-specific probability of experiencing the event modeled by this node at the particular point in time of the simulation. The corresponding event will be generated internally using the `rbernoulli` function. The function needs to have a named argument called `data`. If the function has an argument named `sim_time`, the current simulation time will also be passed to this function automatically, allowing time-dependent probabilities to be generated. Alternatively this argument can be set to a single number (between 0 and 1), resulting in a fixed probability of occurrence for every simulated individual at every point in time.

`...` An arbitrary amount of additional named arguments passed to `prob_fun` if `prob_fun` is specified, or to `node_binomial` if `formula` is specified and `prob_fun` is not. Ignore this if you do not want to pass any arguments. Also ignored if `prob_fun` is a single number.

`event_duration` A single number > 0 specifying how long the event should last. The point in time at which an event occurs also counts into this duration. For example, if an event occurs at $t = 2$ and it has a duration of 3, the event will be set to `TRUE` on $t \in \{2, 3, 4\}$. Therefore, all events must have a duration of at least 1 unit (otherwise they never happened).

`immunity_duration` A single number >= `event_duration` specifying how long the person should be immune to the event after it is over. The count internally starts when the event starts, so in order to use an immunity duration of 10 time units after the event is over `event_duration + 10` should be used.

`unif` Specifies the (usually uniformly distributed) numeric vector that should be used to perform the Bernoulli trials. If `NULL` (default), the uniform numbers are generated internally at each point in time. If a single character string is supplied, a column with the same name in `data` will be used for these numbers (can, but does not need to be mentioned in `parents`). If a numeric vector is supplied directly, these values will be used instead. This argument may be useful to make two or more time-to-event nodes use the same "seed".

time_since_last
: Either TRUE or FALSE (default), indicating whether an additional column should be generated that tracks the number of time units since the individual had its last event onset. For example, if the individual experienced a single event at $t = 10$, this column would be NA before time 10, 0 at time 10 and increased by 1 at each point in time. If another event happens, the time is set to 0 again. The column is named paste0(name, "_time_since_last"). The difference to the column ending with "_time" is that this column will not be set to NA again if the immunity_duration is over. It keeps counting until the end of the simulation, which may be useful when constructing event-time dependent probability functions.

event_count
: Either TRUE or FALSE (default), indicating whether an additional column should be generated that tracks the number of events the individual has already experienced. This column is 0 for all individuals at t = 0. Each time a new event occurs, the counter is increased by one. Note that only new events increase this counter. For example, an individual with an event at t = 10 that has an event_duration of 15 will have a value of 0 before t = 10, and will have a value of 1 at t = 10 and afterwards. The column will be named paste0(name, "_event_count").

save_past_events
: When the event modeled using this node is recurrent (immunity_duration < Inf & event_duration < Inf), the same person may experience multiple events over the course of the simulation. Those are generally stored in the tte_past_events list which is included in the output of the sim_discrete_time function. This extends the runtime and increases RAM usage, so if you are not interested in the timing of previous events or if you are using save_states="all" this functionality can be turned off by setting this argument to FALSE.

check_inputs
: Whether to perform plausibility checks for the user input or not. Is set to TRUE by default, but can be set to FALSE in order to speed things up when using this function in a simulation study or something similar.

envir
: Only used internally to efficiently store the past event times. Cannot be used by the user.

Details

When performing discrete-time simulation using the sim_discrete_time function, the standard node functions implemented in this package are usually not sufficient because they don't capture the time-dependent nature of some very interesting variables. Often, the variable that should be modelled has some probability of occurring at each point in time. Once it does occur, it has some kind of influence on other variables for a period of time until it goes back to normal (or doesn't). This could be a car crash, a surgery, a vaccination etc. The time_to_event node function can be used to model these kinds of nodes in a fairly straightforward fashion.

How it Works:

At $t = 1$, this node will be initialized for the first time. It adds two columns to the data: name_event (whether the person currently has an event) and name_time (the time at which the current event started) where name is the name of the node. Additionally, it adds a list with max_t entries to the tte_past_events list returned by the sim_discrete_time function, which records which individuals experienced a new event at each point in time.

In a nutshell, it simply models the occurrence of some event by calculating the probability of occurrence at t and drawing a single bernoulli trial from this probability. If the trial is a "success", the corresponding event column will be set to TRUE, the time column will be set to the current simulation time t and the column storing the past event times will receive an entry.

The _event column will stay TRUE until the event is over. The duration for that is controlled by the event_duration parameter. When modeling terminal events such as death, one can simply set this parameter to Inf, making the event eternal. In many cases it will also be necessary to implement some kind of immunity after the event, which can be done using the immunity_duration argument. This effectively sets the probability of another occurrence of the event to 0 in the next immunity_duration time steps. During the immunity duration, the event may be TRUE (if the event is still ongoing) or FALSE (if the event_duration has already passed). The _time column is similarly set to the time of occurrence of the event and reset to NA when the immunity_duration is over.

The probability of occurrence is calculated using the function provided by the user using the prob_fun argument. This can be an arbitrary complex function. The only requirement is that it takes data as a first argument. The columns defined by the parents argument will be passed to this argument automatically. If it has an argument called sim_time, the current time of the simulation will automatically be passed to it as well. Any further arguments can be passed using the ... syntax. A simple example could be a logistic regression node, in which the probability is calculated as an additive linear combination of the columns defined by parents (this could also be achieved more cleanly using the formula argument). A more complex function could include simulation-time dependent effects, further effects dependent on past event times etc. Examples can be found below and in the vignettes.

How it is Used:

This function should never be called directly by the user. Instead, the user should define a DAG object using the empty_dag and node_td functions and set the type argument inside of a node_td call to "time_to_event". This DAG can be passed to the sim_discrete_time function to generate the desired data. Many examples and more explanations are given below and in the vignettes of this package.

What can be done with it:

This type of node naturally supports the implementation of terminal and recurrent events that may be influenced by pretty much anything. By specifying the parents and prob_fun arguments correctly, it is possible to create an event type that is dependent on past events of itself or other time-to-event variables and other variables in general. The user can include any amount of these nodes in their simulation. It may also be used to simulate any kind of binary time-dependent variable that one would usually not associate with the name "event" as well. It is very flexible, but it does require the user to do some coding by themselves (e.g. creating a suitable function for the prob_fun argument).

What can't be done with it:

Currently this function only allows binary events. Categorical event types may be implemented using the node_competing_events function, which works in a very similar fashion.

Value

Returns a data.table containing at least two columns with updated values of the node.

Note

This function cannot be called outside of the sim_discrete_time function. It only makes sense to use it as a type in a node_td function call, as described in the documentation and vignettes.

Author(s)

Robin Denz, Katharina Meiszl

See Also

empty_dag, node_td, sim_discrete_time

Examples

```
library(simDAG)

## a simple terminal time-to-event node, with a constant probability of
## occurrence, independent of any other variable
dag <- empty_dag() +
  node_td("death", type="time_to_event", prob_fun=0.0001,
          event_duration=Inf)

## a simple recurrent time-to-event node with a constant probability of
## occurrence, independent of any other variable
dag <- empty_dag() +
  node_td("car_crash", type="time_to_event", prob_fun=0.001, event_duration=1)

## a time-to-event node with a time-dependent probability function that
## has an additional argument
prob_car_crash <- function(data, sim_time, base_p) {
  return(base_p + sim_time * 0.0001)
}

dag <- empty_dag() +
  node_td("car_crash", type="time_to_event", prob_fun=prob_car_crash,
          event_duration=1, base_p=0.0001)

## a time-to-event node with a probability function dependent on a
## time-fixed variable
prob_car_crash <- function(data) {
  ifelse(data$sex==1, 0.001, 0.01)
}

dag <- empty_dag() +
  node("sex", type="rbernoulli", p=0.5) +
  node_td("car_crash", type="time_to_event", prob_fun=prob_car_crash,
          parents="sex")

## a little more complex car crash simulation, where the probability for
## a car crash is dependent on the sex, and the probability of death is
## highly increased for 3 days after a car crash happened
prob_car_crash <- function(data) {
```

```
  ifelse(data$sex==1, 0.001, 0.01)
}

prob_death <- function(data) {
  ifelse(data$car_crash_event, 0.1, 0.0001)
}

dag <- empty_dag() +
  node("sex", type="rbernoulli", p=0.5) +
  node_td("car_crash", type="time_to_event", prob_fun=prob_car_crash,
          parents="sex") +
  node_td("death", type="time_to_event", prob_fun=prob_death,
          parents="car_crash_event")

# use the sim_discrete_time function to simulate data from one of these DAGs:
sim <- sim_discrete_time(dag, n_sim=20, max_t=500)

# using a logistic regression model to specify the probability with the
# enhanced formula interface
dag <- empty_dag() +
    node(c("A", "B"), type="rnorm") +
    node_td("Y", type="time_to_event", formula= ~ -2 + A*1.2 + B*-0.2,
            event_duration=10)

## more examples can be found in the vignettes of this package
```

---

node_zeroinfl *Generate Data from a Zero-Inflated Count Model*

Description

Data from the parents is used to first simulate data for the regular count model, which may follow either a poisson regression or a negative binomial regression, as implemented in node_poisson and node_negative_binomial respectively. Then, zeros are simulated using a logistic regression model as implemented in node_binomial. Whenever the second binomial part returned a 0, the first part is set to 0, leaving the rest untouched. Supports random effects and random slopes (if possible) in both models. See examples.

Usage

```
node_zeroinfl(data, parents, parents_count,
              parents_zero, formula_count, formula_zero,
              betas_count, betas_zero,
              intercept_count, intercept_zero,
              family_count="poisson", theta,
              link_count, link_zero="logit",
              var_corr_count, var_corr_zero)
```

Arguments

data	A data.table (or something that can be coerced to a data.table) containing all columns specified by parents, parents_count and parents_zero.
parents	A character vector specifying the names of the parents that this particular child node has. Note that this argument does not have to be specified if parents_count and parents_zero are specified. If non-linear combinations or interaction effects should be included, the user should specify the formula_count and/or formula_zero arguments instead.
parents_count	Same as parents but should only contain the parents of the count model part of the node.
parents_zero	Same as parents but should only contain the parents of the zero-inflation model part of the node.
formula_count	An enhanced formula passed to the node_poisson or the node_negative_binomial function, used to generate the count part of the node. If this argument is specified, there is no need to specify the parents_count, betas_count and intercept_count arguments. The syntax is the same as in the usual formula argument as described in node.
formula_zero	An enhanced formula passed to the node_binomial function, used to generate the zero-inflated part of the node. If this argument is specified, there is no need to specify the parents_zero, betas_zero and intercept_zero arguments. The syntax is the same as in the usual formula argument as described in node.
betas_count	A numeric vector with length equal to parents_count, specifying the causal beta coefficients used to generate the node in the count model.
betas_zero	A numeric vector with length equal to parents_zero, specifying the causal beta coefficients used to generate the node in the zero-inflation model.
intercept_count	A single number specifying the intercept that should be used when generating the count model part of the node.
intercept_zero	A single number specifying the intercept that should be used when generating the zero-inflated part of the node.
family_count	Either "poisson" for a zero-inflated poisson regression or "negative_binomial" for a zero-inflated negative binomial regression.
theta	A single number specifying the theta parameter (size argument in rnbinom). Ignore if family_count="poisson".
link_count	A single character string, passed to the link argument of the respective node function used for the count model part. If not supplied, the default of the respective link function is used.
link_zero	A single character string specifying the link in the node_binomial function.
var_corr_count	If random effects or random slopes are included in formula_count, this argument should be specified to define the variance structure of these effects. It will be passed to the var_corr argument of node_poisson. Random effects or slopes are currently not supported with family_count="negative_binomial".
var_corr_zero	If random effects or random slopes are included in formula_zero, this argument should be specified to define the variance structure of these effects. It will be passed to the var_corr argument of node_binomial.

Details

It is important to note that data for both underlying models (the count model and the zero-inflation model) are simulated from completely independent of each other. When using random effects in either of the two models, they may therefore use completely different values for each process.

Value

Returns a numeric vector of length `nrow(data)`.

Author(s)

Robin Denz

See Also

`empty_dag`, `node`, `node_td`, `sim_from_dag`, `sim_discrete_time`

Examples

```
library(simDAG)

set.seed(5425)

# zero-inflated poisson regression
dag <- empty_dag() +
  node(c("A", "B"), type="rnorm", mean=0, sd=1) +
  node("Y", type="zeroinfl",
       formula_count= ~ -2 + A*0.2 + B*0.1 + A:B*0.4,
       formula_zero= ~ 1 + A*1 + B*2,
       family_count="poisson",
       parents=c("A", "B"))
data <- sim_from_dag(dag, n_sim=100)

# above is functionally the same as:
dag <- empty_dag() +
  node(c("A", "B"), type="rnorm", mean=0, sd=1) +
  node("Y_count", type="poisson", formula= ~ -2 + A*0.2 + B*0.1 + A:B*0.4) +
  node("Y_zero", type="binomial", formula= ~ 1 + A*1 + B*2) +
  node("Y", type="identity", formula= ~ Y_zero * Y_count)
data <- sim_from_dag(dag, n_sim=100)

# same as above, but specifying each individual component instead of formulas
dag <- empty_dag() +
  node(c("A", "B", "C"), type="rnorm", mean=0, sd=1) +
  node("Y", type="zeroinfl",
       parents_count=c("A", "B"),
       betas_count=c(0.2, 0.1),
       intercept_count=-2,
       parents_zero=c("A", "B"),
       betas_zero=c(1, 2),
       intercept_zero=1,
       family_count="poisson",
```

```
      parents=c("A", "B"))
data <- sim_from_dag(dag, n_sim=100)

# zero-inflated negative-binomial regression
dag <- empty_dag() +
  node(c("A", "B"), type="rnorm", mean=0, sd=1) +
  node("Y", type="zeroinfl",
       formula_count= ~ -2 + A*0.2 + B*3 + A:B*0.4,
       formula_zero= ~ 3 + A*0.1 + B*0.3,
       family_count="negative_binomial", theta=1,
       parents=c("A", "B"))
data <- sim_from_dag(dag, n_sim=100)
```

`plot.calibrated_DAG` *Plot a* `calibrated_DAG` *object*

Description

This function may be used to visually inspect how well a `calibrate_dag` function call was able to approximate the desired targets. If fitted with `n_reps_eval` > 1, it plots the distribution of the targets estimated on the `n_reps_eval` evaluation datasets using the determined parameters. Alternatively plots the bias directly.

Usage

```
## S3 method for class 'calibrated_DAG'
plot(x, type="errors", xlab=NULL,
     ylab="auto", line_at_0=TRUE, ...)
```

Arguments

`x`	A `calibrated_DAG` object created using the `calibrate_dag` function.
`type`	A single character string, specifying what should be plotted. Should be either `"targets"` (to plot the raw target estimates), `"scaled_targets"` (to plot the target estimates after applying the user-specified scaling factors, if applicable), `"errors"` (to plot the difference between the estimated targets and the desired targets, the default) or `"scaled_errors"` (same as errors, but after applying the user-specified scaling factors, if applicable).
`xlab`	A single character string specifying the name of the x-axis or `NULL` (default) to not use a name.
`ylab`	A single character string specifying the name of the y-axis or `NULL` to not use a name. If set to `"auto"` (default), the label will be determined automatically based on the supplied `type`.
`line_at_0`	Either `TRUE` or `FALSE`, specifying whether to plot a dashed line at 0 whenever `type` = `"errors"` or `type` = `"scaled_errors"` is used, ignored otherwise.
`...`	Further arguments passed to the `geom_boxplot` function, allowing users to customize their appearance further.

Details

The function internally relies on the **ggplot2** package, so any further `ggplot2` syntax may be used to customize the plot. Its main use is to visually assess whether the observed discrepancy between the estimated and desired targets is due to sampling error or a genuine difference.

Value

Returns a standard `ggplot2` object.

Author(s)

Robin Denz

Examples

```
# Jointly calibrate treatment and outcome prevalence.
# The DAG contains two free parameters and two targets.
# Parameter ranges are optional.

ldag <- lazy_dag(
  empty_dag() +
    node("X", type="rnorm") +
    node("A", type="binomial",
         formula= ~ param(intercept_A) + X * 0.5) +
    node("Y", type="binomial",
         formula= ~ param(intercept_Y) + A * 0.7 + X * 0.4)
)

targets <- list(
  prevalence_A = list(
    target = 0.4,
    eval_fun = function(data) mean(data$A)
  ),
  prevalence_Y = list(
    target = 0.3,
    eval_fun = function(data) mean(data$Y)
  )
)

params <- list(
  intercept_A = list(start = 0),
  intercept_Y = list(start = 0)
)

fit <- calibrate_dag(
  lazy_dag = ldag,
  targets = targets,
  params = params,
  n_sim_run = 2000,
  n_sim_eval = 1000,
  n_reps_eval = 10,
  seed = 123,
```

```
  optim_control = list(
    maxit = 200,
    trace = 0,
    ndeps = c(0.05, 0.05),
    factr = 100
  )
)
summary(fit)

# plot the targets directly
plot(fit, type="targets")

# plot the errors instead
plot(fit, type="errors")
```

---

plot.DAG *Plot a* DAG *object*

Description

Using the node information contained in the DAG object this function plots the corresponding DAG in a quick and convenient way. Some options to customize the plot are available, but it may be advisable to use other packages made explicitly to visualize DAGs instead if those do not meet the users needs.

Usage

```
## S3 method for class 'DAG'
plot(x, layout="nicely", node_size=0.2,
     node_names=NULL, node_color="black",
     node_fill="red", node_linewidth=0.5,
     node_linetype="solid", node_alpha=1,
     node_text_color="black", node_text_alpha=1,
     node_text_size=8, node_text_family="sans",
     node_text_fontface="bold", arrow_color="black",
     arrow_linetype="solid", arrow_linewidth=1,
     arrow_alpha=1, arrow_head_size=0.3,
     arrow_head_unit="cm", arrow_type="closed",
     arrow_node_dist=0.03, gg_theme=ggplot2::theme_void(),
     include_td_nodes=TRUE, mark_td_nodes=TRUE,
     ...)
```

Arguments

x	A DAG object created using the empty_dag function with nodes added to it using the + syntax. See empty_dag or node for more details.

`layout` A single character string specifying the layout of the plot. This internally calls the `layout_` function of the **igraph** package, which offers a great variety of ways to layout the nodes of a graph. Defaults to `"nicely"`. Some other options are: `"as_star"`, `"as_tree"`, `"in_circle"`, `"on_sphere"`, `"randomly"` and many more. For more details see `?layout_`.

`node_size` Either a single positive number or a numeric vector with one entry per node in the DAG, specifying the radius of the circles used to draw the nodes. If a single number is supplied, all nodes will be the same size (default).

`node_names` A character vector with one entry for each node in the DAG specifying names that should be used for in the nodes or `NULL` (default). If `NULL`, the node names that were set during the creation of the DAG object will be used as names.

`node_color` A single character string specifying the color of the outline of the node circles.

`node_fill` A single character string specifying the color with which the nodes are filled. Ignored if time-varying nodes are present and both `include_td_nodes` and `mark_td_nodes` are set to `TRUE`.

`node_linewidth` A single number specifying the width of the outline of the node circles.

`node_linetype` A single character string specifying the linetype of the outline of the node circles.

`node_alpha` A single number between 0 and 1 specifying the transparency level of the nodes.

`node_text_color`
A single character string specifying the color of the text inside the node circles.

`node_text_alpha`
A single number between 0 and 1 specifying the transparency level of the text inside the node circles.

`node_text_size` A single number specifying the size of the text inside of the node circles.

`node_text_family`
A single character string specifying the family of the text inside the node circles.

`node_text_fontface`
A single character string specifying the fontface of the text inside the node circles.

`arrow_color` A single character string specifying the color of the arrows between the nodes.

`arrow_linetype` A single character string specifying the linetype of the arrows.

`arrow_linewidth`
A single number specifying the width of the arrows.

`arrow_alpha` A single number between 0 and 1 specifying the transparency level of the arrows.

`arrow_head_size`
A single number specifying the size of the arrow heads. The unit for this size parameter can be changed using the `arrow_head_unit` argument.

`arrow_head_unit`
A single character string specifying the unit of the `arrow_head_size` argument.

`arrow_type` Either `"open"` or `"closed"`, which controls the type of head the arrows should have. See `?arrow`.

`arrow_node_dist`
: A single positive number specifying the distance between nodes and the arrows. By setting this to values greater than 0 the arrows will not touch the node circles, leaving a bit of space instead.

`gg_theme`
: A `ggplot2` theme. By default this is set to `theme_void`, to get rid off everything but the plotted nodes (e.g. everything about the axis and the background). Might be useful to change this to something else when searching for good parameters of the number arguments of this function.

`include_td_nodes`
: Whether to include time-varying nodes added to the `dag` using the `node_td` function or not. If one node is both specified as a time-fixed and time-varying node, it's parents in both calls will be pooled and it will be considered a time-varying node if this argument is `TRUE`. It will, however, also show up if it's argument is `FALSE`. In this case however, only the parents of that node in the standard `node` call will be considered.

`mark_td_nodes`
: Whether to distinguish time-varying and time-fixed nodes by `fill` color. If `TRUE`, the color will be set automatically using the standard `ggplot2` palette, ignoring the color specified in `node_fill`. Ignored if `include_td_nodes=FALSE` or if there are no time-varying variables.

`...`
: Further arguments passed to the `layout` function specified by the argument of the same name.

Details

This function uses the **igraph** package to find a suitable layout for the plot and then uses the **ggplot2** package in conjunction with the `geom_circle` function of the **ggforce** package to plot the directed acyclic graph defined by a `DAG` object. Since it returns a `ggplot` object, the user may use any standard `ggplot2` syntax to augment the plot or to save it using the `ggsave` function.

Note that there are multiple great packages specifically designed to plot directed acyclic graphs, such as the **igraph** package. See Pitts and Fowler (2024) for a review. This function is not meant to be a competitor to those packages. The functionality offered here is rather limited. It is designed to produce decent plots for small DAGs which are easy to create. If this function is not enough to create an adequate plot, users can use the `dag2matrix`, `as.igraph.DAG` or `as_tidy_dagitty.DAG` functions to transform the `DAG` into other formats, which allow usage of much better plotting routines, such as the ones provided by the **igraph** or **ggdag** packages.

If the `DAG` supplied to this function contains time-varying variables, the resulting plot may contain cycles or even bi-directional arrows, depending on the `DAG`. The reason for that is, that the time-dimension is not shown in the plot. Note also that even though, technically, every time-varying node has itself as a parent, no arrows showing this dependence will be added to the plot.

Value

Returns a standard `ggplot2` object.

Author(s)

Robin Denz

References

Pitts, Amy J. and Charlotte R. Fowler (2024). Comparison of Open-Source Software for Producing Directed Acyclic Graphs. In: *Journal of Causal Inference* 12.1

See Also

empty_dag, node, node_td, as.igraph.DAG, as_tidy_dagitty.DAG

Examples

```
library(simDAG)

# 2 root nodes, 1 child node
dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("smoking", type="binomial", parents=c("sex", "age"), betas=c(1.1, 0.4),
       intercept=-2)

if (requireNamespace("ggplot2") & requireNamespace("ggforce")) {

library(ggplot2)
library(igraph)
library(ggforce)

plot(dag)

# get plot using the igraph package instead
g1 <- as.igraph(dag)
plot(g1)

# plot with a time-varying node
dag <- dag +
  node_td("lottery", type="time_to_event", parents=c("age", "smoking"))

plot(dag)
}
```

---

plot.simDT *Plot a Flowchart for a Discrete-Time Simulation*

Description

Given a simDT object obtained with the sim_discrete_time function, plots a relatively simple flowchart of how the simulation was performed. Shows only some general information extracted from the dag.

Usage

```
## S3 method for class 'simDT'
plot(x, right_boxes=TRUE,
     box_hdist=1, box_vdist=1,
     box_l_width=0.35, box_l_height=0.23,
     box_r_width=box_l_width,
     box_r_height=box_l_height + 0.1,
     box_alpha=0.5, box_linetype="solid",
     box_linewidth=0.5, box_border_colors=NULL,
     box_fill_colors=NULL, box_text_color="black",
     box_text_alpha=1, box_text_angle=0,
     box_text_family="sans", box_text_fontface="plain",
     box_text_size=5, box_text_lineheight=1,
     box_1_text_left="Create initial data",
     box_1_text_right=NULL, box_2_text="Increase t by 1",
     box_l_node_labels=NULL, box_r_node_labels=NULL,
     box_last_text=paste0("t <= ", x$max_t, "?"),
     arrow_line_type="solid", arrow_line_width=0.5,
     arrow_line_color="black", arrow_line_alpha=1,
     arrow_head_angle=30, arrow_head_size=0.3,
     arrow_head_unit="cm", arrow_head_type="closed",
     arrow_left_pad=0.3, hline_width=0.5,
     hline_type="dashed", hline_color="black",
     hline_alpha=1, ...)
```

Arguments

x	A simDT object created using the sim_discrete_time function.
right_boxes	Either TRUE (default) or FALSE, specifying whether to add boxes on the right with some additional information about the nodes on the left.
box_hdist	A single positive number specifying the horizontal distance of the left and the right boxes.
box_vdist	A single positive number specifying the vertical distance of the boxes.
box_l_width	A single positive number specifying the width of the boxes on the left side.
box_l_height	A single positive number specifying the height of the boxes on the left side.
box_r_width	A single positive number specifying the width of the boxes on the right side. Ignored if right_boxes=FALSE.
box_r_height	A single positive number specifying the height of the boxes on the right side. Ignored if right_boxes=FALSE.
box_alpha	A single number between 0 and 1 specifying the transparency level of the boxes.
box_linetype	A single positive number specifying the linetype of the box outlines.
box_linewidth	A single positive number specifying the width of the box outlines.
box_border_colors	A character vector of length two specifying the colors of the box outlines. Set to NULL (default) to use ggplot2 default colors.

box_fill_colors
: A character vector of length two specifying the colors of the inside of the boxes. Set to NULL (default) to use ggplot2 default colors.

box_text_color
: A single character string specifying the color of the text inside the boxes.

box_text_alpha
: A single number between 0 and 1 specifying the transparency level of the text inside the boxes.

box_text_angle
: A single positive number specifying the angle of the text inside the boxes.

box_text_family
: A single character string specifying the family of the text inside the boxes. May be one of "sans", "serif", "mono".

box_text_fontface
: A single character string specifying the fontface of the text inside the boxes. May be one of "plain", "bold", "italic", "bold.italic".

box_text_size
: A single number specifying the size of the text inside the boxes.

box_text_lineheight
: A single number specifying the lineheight of the text inside the boxes.

box_1_text_left
: A single character string specifying the text inside the first box from the top on the left side.

box_1_text_right
: A single character string specifying the text inside the first box from the top on the right side or NULL. If NULL (default) it will simply state which variables were generated at t = 0.

box_2_text
: A single character string specifying the text inside the second box from the top.

box_l_node_labels
: A character vector with one entry for each time-varying node used in the simulation. These will be used to fill the boxes on the left side of the plot. Set to NULL to use default values.

box_r_node_labels
: A character vector with one entry for each time-varying node used in the simulation. These will be used to fill the boxes on the right side of the plot. Set to NULL to use default values. Ignored if right_boxes=FALSE.

box_last_text
: A single character string specifying the text inside the last box on the left side. By default it uses the max_t argument from the initial function call to construct a fitting text.

arrow_line_type
: A single character string specifying the linetype of the arrows.

arrow_line_width
: A single positive number specifying the line width of the arrows.

arrow_line_color
: A single character string specifying the color of the arrows.

arrow_line_alpha
: A single number between 0 and 1 specifying the transparency level of the arrows.

`arrow_head_angle`
: A single number specifying the angle of the arrow heads.

`arrow_head_size`
: A single number specifying the size of the arrow heads. The unit is defined by the `arrow_head_size` argument.

`arrow_head_unit`
: A single character string specifying which unit to use when specifying the `arrow_head_size` argument. Defaults to `"cm"`.

`arrow_head_type`
: A single character string specifying which type of arrow head to use. See `?arrow` for more details.

`arrow_left_pad`
: A single positive number specifying the distance between the left boxes and the arrow line to the left of it.

`hline_width`
: A single number specifying the width of the horizontal lines between the left and right boxes.

`hline_type`
: A single character string specifying the linetype of the horizontal lines between the left and right boxes.

`hline_color`
: A single character string specifying the color of the horizontal lines between the left and right boxes.

`hline_alpha`
: A single number between 0 and 1 specifying the transparency level of the horizontal lines between the left and right boxes.

`...`
: Currently not used.

Details

The resulting flowchart includes two columns of boxes next to each other. On the left side it always starts with the same two boxes: a box about the creation of the initial data and a box about increasing the simulation time by 1. Next, there will be a box for each time-varying variable in the `simDT` object. Afterwards there is another box which asks if the maximum simulation time was reached. An arrow to the left that points back to the second box from the top indicates the iterative nature of the simulation process. The right column of boxes includes additional information about the boxes on the left.

The text in all boxes may be changed to custom text by using the `box_1_text_left`, `box_1_text_right`, `box_2_text`, `box_l_node_labels`, `box_r_node_labels` and `box_last_text` arguments. It is also possible to completely remove the left line of boxes and to change various sizes and appearances. Although these are quite some options, it is still a rather fixed function in nature. One cannot add further boxes or arrows in a simple way. The general structure may also not be changed. It may be useful to visualize a general idea of the simulation flow, but it may be too limited for usage in scientific publications if the simulation is more complex.

The graphic is created using the `ggplot2` package and the output is a standard `ggplot` object. This means that the user can change the result using standard `ggplot` syntax (adding more stuff, changing geoms, ...).

Value

Returns a standard `ggplot` object.

Author(s)

Robin Denz

See Also

empty_dag, node, node_td, sim_discrete_time

Examples

```
library(simDAG)

set.seed(435345)

## exemplary car crash simulation, where the probability for
## a car crash is dependent on the sex, and the probability of death is
## highly increased for 3 days after a car crash happened
prob_car_crash <- function(data) {
  ifelse(data$sex==1, 0.001, 0.01)
}

prob_death <- function(data) {
  ifelse(data$car_crash_event, 0.1, 0.0001)
}

dag <- empty_dag() +
  node("sex", type="rbernoulli", p=0.5) +
  node_td("car_crash", type="time_to_event", prob_fun=prob_car_crash,
          parents="sex") +
  node_td("death", type="time_to_event", prob_fun=prob_death,
          parents="car_crash_event")

# generate some data
sim <- sim_discrete_time(dag, n_sim=20, max_t=500, save_states="last")

if (requireNamespace("ggplot2")) {

# default plot
plot(sim)

# removing boxes on the right
plot(sim, right_boxes=FALSE)
}
```

---

rbernoulli *Generate Random Draws from a Bernoulli Distribution*

Description

A very fast implementation for generating bernoulli trials. Can take a vector of probabilities which makes it very useful for simulation studies.

Usage

```
rbernoulli(n, p=0.5, output="logical", reference=NULL)
```

Arguments

n	How many draws to make.
p	A numeric vector of probabilities, used when drawing the trials.
output	A single character string, specifying which format the output should be returned as. Must be one of "logical" (default), "numeric", "character" or "factor".
reference	A single character string, specifying which of the two possible values should be considered the reference when output="factor" (ignored otherwise).

Details

Internally, it uses only a single call to runif, making it much faster and more memory efficient than using rbinomial.

Note that this function accepts values of p that are smaller then 0 and greater than 1. For p < 0 it will always return FALSE, for p > 1 it will always return TRUE.

Value

Returns a vector of length n in the desired output format.

Author(s)

Robin Denz

Examples

```
library(simDAG)

# generating 5 bernoulli random draws from an unbiased coin
rbernoulli(n=5, p=0.5)

# using different probabilities for each coin throw
rbernoulli(n=5, p=c(0.1, 0.2, 0.3, 0.2, 0.7))

# return as numeric instead
rbernoulli(n=5, p=0.5, output="numeric")
```

rcategorical *Generate Random Draws from a Discrete Set of Labels with Associated Probabilities*

Description

Allows different class probabilities for each person by supplying a matrix with one column for each class and one row for each person.

Usage

```
rcategorical(n, probs, labels=NULL, output="numeric",
             reference=NULL, all_levels=FALSE)
```

Arguments

n	How many draws to make. Passed to the size argument of the sample function if probs is not a matrix.
probs	Either a numeric vector of probabilities which sums to one or a matrix with one column for each desired class and n rows. Passed to the probs argument of the sample function if a numeric vector is passed.
labels	A vector of labels to draw from. If NULL (default), it simply uses integers starting from 1. Passed to the x argument of the sample function if probs is not a matrix.
output	A single character string specifying the output format of the results. Must be either "numeric" (default), "character" or "factor". If labels are supplied, the output will be parsed as characters by default.
reference	A single character string, specifying which of the possible values should be considered the reference when output="factor" (ignored otherwise).
all_levels	Either TRUE or FALSE, specifying whether the resulting factor (when output="factor") should retain all levels specified by labels, even if some of these levels were never generated. For example, if labels=c("A", "B", "C") and "A" was never drawn, the levels of the factor would be c("B", "C") with all_levels=FALSE and c("A", "B", "C") with all_levels=TRUE. Ignored if output!="factor".

Details

In case of a simple numeric vector (class probabilities should be the same for all draws), this function is only a wrapper for the sample function, to make the code more consistent. It uses weighted sampling with replacement. Otherwise, custom code is used which is faster than the standard rmultinom function.

Value

Returns a numeric vector (or factor vector if coerce2factor=TRUE) of length n.

Author(s)

Robin Denz

Examples

```
library(simDAG)

rcategorical(n=5, labels=c("A", "B", "C"), probs=c(0.1, 0.2, 0.7))

rcategorical(n=2, probs=matrix(c(0.1, 0.2, 0.5, 0.7, 0.4, 0.1), nrow=2))
```

---

rconstant *Use a single constant value for a root node*

Description

This is a small convenience function that simply returns the value passed to it, in order to allow the use of a constant node as root node in the `sim_from_dag` function.

Usage

```
rconstant(n, constant)
```

Arguments

`n`	The number of times the constant should be repeated.
`constant`	A single value of any kind which is used as the only value of the resulting variable.

Value

Returns a vector of length `n` with the same type as `constant`.

Author(s)

Robin Denz

Examples

```
library(simDAG)

rconstant(n=10, constant=7)

rconstant(n=4, constant="Male")
```

rsample *Sample values from a given vector*

Description

This function is a simple wrapper around the sample function, allowing users to directly sample values from a given input vector (with or without replacement and with or without defining selection probabilities) or data.frame like object.

Usage

```
rsample(n, x, replace=FALSE, prob=NULL)
```

Arguments

n	How many draws to make.
x	A vector containing one or more elements from which to sample from, or a data.frame like object. If a data.frame is supplied, random rows from it will be sampled. Note that if the supplied data.frame has more than one column and this function is used as a node type, the names of the variables in the supplied x will be used as variable names and the given node name will be discarded.
replace	Either TRUE or FALSE, specifying whether the sampling should be performed with or without replacement.
prob	A numeric vector of probability weights for obtaining the elements of the vector being sampled or NULL (default). If NULL, a simple random sample without weights will be performed.

Details

This function is very similar to the rcategorical function, with the main difference being that rsample() directly supports any kind of vector input, not just a few categorical values, but it does not support matrix input in the prob argument. Use rcategorical if the goal is to sample from a categorical distribution with few categories or different probabilities per person and use rsample() for general sampling purposes.

Note that this function is just a wrapper around the sample function, with the only additional functionality being that it also may be used to directly sample from data.frames. It is only meant to conveniently allow sampling within the packages syntax (the original function does not use the n argument, and can thus not be used directly without a wrapper).

Value

Returns a vector of length n with the same type as x if x is a vector and a data.frame with n rows if x is a data.frame.

Author(s)

Robin Denz

Examples

```
library(simDAG)

# without replacement
dag <- empty_dag() +
  node("A", type="rsample", x=1:10, replace=FALSE)
data <- sim_from_dag(dag, n_sim=5)
head(data)

# with replacement and selection probabilities
dag <- empty_dag() +
  node("X", type="rbernoulli", p=0.5) +
  node("A", type="rsample", x=c(1, 2, 3, 4), replace=TRUE,
       prob=c(0.1, 0.3, 0.1, 0.5))
data <- sim_from_dag(dag, n_sim=100)
head(data)

# sampling rows from a data.frame object
# NOTE: The node name for the rsample() node will be ignored, because
#       a data.frame is supplied to "x". The names of the variables in the
#       data are used directly instead.
dag <- empty_dag() +
  node("placeholder", type="rsample", x=data) +
  node("Y", type="binomial", formula= ~ -2 + A*0.5 + X*-1)
data2 <- sim_from_dag(dag, n_sim=50)
head(data2)
```

---

rtexp *Generate Data from a left-truncated exponential distribution*

Description

This function is a simple wrapper around the rexp function, allowing users to directly sample values from a left-truncated exponential distribution.

Usage

```
rtexp(n, rate, l=NULL)
```

Arguments

n	How many draws to make.
rate	A numeric vector of numbers > 0, specifying the rate parameter of the exponential distribution.
l	A numeric vector of numbers > 0, specifying the value at which the distribution should be left-truncated.

Details

This function mostly exists so it can be used conveniently when performing discrete-event simulations.

Value

Returns a numeric vector of length n.

Author(s)

Robin Denz

Examples

```
library(simDAG)

rtexp(n=10, rate=0.05, l=20)

# without replacement
dag <- empty_dag() +
  node("A", type="rtexp", rate=0.01, l=100)
data <- sim_from_dag(dag, n_sim=5)
head(data)
```

---

set_lazy_dag *Create a fully defined* DAG *object from a* lazy_DAG *object*

Description

Given a un-evaluated lazy_DAG object, created using the lazy_dag function and a named list or vector of parameters, this function returns a fully specified DAG object that can then be used to run actual simulations using sim_from_dag, sim_discrete_time or sim_discrete_event. See details and ?lazy_dag for more information.

Usage

```
set_lazy_dag(lazy_dag, params)
```

Arguments

lazy_dag	A lazy_DAG object created using the lazy_dag function. See ?lazy_dag for more information on how to specify a lazy DAG. See ?node for more information on how to specify a regular DAG.
params	A named list or a named vector, containing one entry per param() call used in the creation of the lazy_dag object. These entries should be scalar values that can be substituted directly into the DAG definition.

Details

Given the previously un-evaluated lazy_DAG object, this function substitutes all param() pieces with the corresponding parameter defined in params and then finally evaluates the DAG. Note that the lazy_DAG object itself is not changed in place, so that the set_lazy_dag() function may be used multiple times on the same lazy_DAG to create multiple different data generation processes.

Value

Returns a DAG object.

Author(s)

Robin Denz

Examples

```
library(simDAG)

# define a lazy DAG with one unspecified parameter
ldag <- lazy_dag(
  empty_dag() +
    node("A", type="rnorm", mean=1, sd=param(SD))
)

# get a real DAG object with SD = 1 and another with SD = 2
dag1 <- set_lazy_dag(ldag, params=c(SD=1))
dag2 <- set_lazy_dag(ldag, params=c(SD=2))

# define some lazy DAG, leaving the intercept of "death" and one beta
# coefficient of "smoking" unspecified
ldag <- lazy_dag(
  empty_dag() +
    node("age", type="rnorm", mean=10, sd=2) +
    node("sex", parents="", type="rbernoulli", p=0.5) +
    node("smoking", type="binomial",
         formula= ~ -2 + age*param(Sbeta) + sex*0.2) +
    node("death", type="binomial",
         formula= ~ param(Dintercept) + age*1 + sex*2)
)

# get a real DAG object from this with set values
dag3 <- set_lazy_dag(ldag, params=c(Sbeta=2.1, Dintercept=-10))
```

---

sim2data *Transform* sim_discrete_time *output into the start-stop, long- or wide-format*

Description

This function transforms the output of the sim_discrete_time function into a single data.table structured in the start-stop format (also known as counting process format), the long format (one row per person per point in time) or the wide format (one row per person, one column per point in time for time-varying variables). See details.

Usage

```
sim2data(sim, to, use_saved_states=sim$save_states=="all",
         overlap=FALSE, target_event=NULL,
         keep_only_first=FALSE, remove_not_at_risk=FALSE,
         remove_vars=NULL, as_data_frame=FALSE,
         check_inputs=TRUE, ...)

## S3 method for class 'simDT'
as.data.table(x, keep.rownames=FALSE, to, overlap=FALSE,
              target_event=NULL, keep_only_first=FALSE,
              remove_not_at_risk=FALSE,
              remove_vars=NULL,
              use_saved_states=x$save_states=="all",
              check_inputs=TRUE, ...)

## S3 method for class 'simDT'
as.data.frame(x, row.names=NULL, optional=FALSE, to,
              overlap=FALSE, target_event=NULL,
              keep_only_first=FALSE, remove_not_at_risk=FALSE,
              remove_vars=NULL,
              use_saved_states=x$save_states=="all",
              check_inputs=TRUE, ...)
```

Arguments

sim, x	An object created with the sim_discrete_time function.
to	Specifies the format of the output data. Must be one of: "start_stop", "long", "wide".
use_saved_states	Whether the saved simulation states (argument save_states in sim_discrete_time function) should be used to construct the resulting data or not. See details.
overlap	Only used when to="start_stop". Specifies whether the intervals should overlap or not. If TRUE, the "stop" column is simply increased by one, as compared to the output when overlap=FALSE. This means that changes for a given t are recorded at the start of the next interval, but the previous interval ends on that same day.
target_event	Only used when to="start_stop". By default (keeping this argument at NULL) all time-to-event nodes are treated equally when creating the start-stop intervals. This can be changed by supplying a single character string to this argument, naming one time-to-event node. This node will then be treated as the outcome.

The output then corresponds to what would be needed to fit a Cox proportional hazards model. See details.

`keep_only_first`
: Only used when `to="start_stop"` and `target_event` is not `NULL`. Either `TRUE` or `FALSE` (default). If `TRUE`, all information after the first event per person will be discarded. Useful when `target_event` should be treated as a terminal variable.

`remove_not_at_risk`
: Only used when `to="start_stop"` and `target_event` is not `NULL`. Either `TRUE` or `FALSE` (default). If `TRUE`, the `event_duration` and `immunity_duration` of the `target_event` are taken into account when constructing the start-stop data. More precisely, the time in which individuals are not at-risk because they are either still currently experiencing the event or because they are immune to the event is removed from the start-stop data. This may be necessary when fitting some survival regression models, because these time-periods should not be counted as time at-risk.

`remove_vars`
: An optional character vector specifying which variables should *not* be included in the output. Set to `NULL` to include all variables included in the `sim` object (default).

`as_data_frame`
: Set this argument to `TRUE` to return a `data.frame` instead of a `data.table`.

`check_inputs`
: Whether to perform input checks (`TRUE` by default). Prints warning messages if the output may be incorrect due to missing information.

`keep.rownames`
: Currently not used.

`row.names`
: Passed to the `as.data.frame` function which is called on the finished `data.table`. See `?as.data.frame` for more information.

`optional`
: Passed to the `as.data.frame` function which is called on the finished `data.table`. See `?as.data.frame` for more information.

`...`
: Further arguments passed to `as.data.frame` (conversion from finished `data.table` to `data.frame`). Only available when directly calling `sim2data` with `as_data_frame=TRUE` or when using `as.data.frame.simDT`.

Details

The raw output of the `sim_discrete_time` function may be difficult to use for further analysis. Using one of these functions, it is straightforward to transform that output into three different formats, which are described below. Note that some caution needs to be applied when using this function, which is also described below. Both `as.data.table` and `as.data.frame` internally call `sim2data` and only exist for user convenience.

***The start-stop format*:**

The start-stop format (`to="start_stop"`), also known as counting process or period format corresponds to a `data.table` containing multiple rows per person, where each row corresponds to a period of time in which no variables changed. These intervals are defined by the `start` and `stop` columns. The `start` column gives the time at which the period started, the `stop` column denotes the time when the period ended. By default these intervals are coded to be non-overlapping, meaning that the edges of the periods are included in the period itself. For example, if the respective period is exactly 1 point in time long, `start` will be equal to `stop`. If non-overlapping periods are desired, the user can specify `overlap=TRUE` instead.

By default, all time-to-event nodes are treated equally. This is not optimal when the goal is to fit survival regression models. In this case, we usually want the target event to be treated in a special way (see for example Chiou et al. 2023). In general, instead of creating new intervals for it we want existing intervals to end at event times with the corresponding event indicator. This can be achieved by naming the target outcome in the `target_event` variable. The previously specified duration of this target event is then ignored. To additionally remove all time periods in which individuals are not at-risk due to the event still going on or them being immune to it (as specified using the `event_duration` and `immunity_duration` parameters of `node_time_to_event`), users may set `remove_not_at_risk=TRUE`. If only the first occurrence of the event is of interest, users may also set `keep_only_first=TRUE` to keep only information up until the first event per person.

***The long format*:**

The long format (`to="long"`) corresponds to a `data.table` in which there is one row per person per point in time. The unique person identifier is stored in the `.id` column and the unique points in time are given in the `.time` column.

***The wide format*:**

The wide format (`to="wide"`) corresponds to a `data.table` with exactly one row per person and multiple columns per points in time for each time-varying variable. All time-varying variables are coded as their original variable name with an underscore and the time-point appended to the end. For example, the variable `sickness` at time-point 3 is named `"sickness_3"`.

***Output with* `use_saved_states=TRUE`:**

If `use_saved_states=TRUE`, this function will use only the data that is stored in the `past_states` list of the `sim` object to construct the resulting `data.table`. This results in the following behavior, depending on which `save_states` option was used in the original `sim_discrete_time` function call:

- `save_states="all"`: A complete `data.table` in the desired format with information for **all observations** at **all points in time** for **all variables** will be created. This is the safest option, but also uses the most RAM and computational time.
- `save_states="at_t"`: A `data.table` in the desired format with correct information for **all observations** at the **user specified times** (`save_states_at` argument) for **all variables** will be created. The state of the simulation at all other times will be ignored, because it wasn't stored. This may be useful in some scenarios, but is generally discouraged unless you have good reasons to use it. A warning message about this is printed if `check_inputs=TRUE`.
- `save_states="last"`: Since only the last state of the simulation was saved, an error message is returned. **No** `data.table` is produced.

***Output with* `use_saved_states=FALSE`:**

If `use_saved_states=FALSE`, this function will use only the data that is stored in the final state of the simulation (`data` object in `sim`) and information about `node_time_to_event` objects. If all `tx_nodes` are `time_to_event` nodes or if all the user cares about are the `time_to_event` nodes and time-fixed variables, this is the best option.

A `data.table` in the desired format with correct information about `all observations` at `all times` is produced, but only with correct entries for **some time-varying variables**, namely `time_to_event` nodes. Note that this information will also only be correct if the user used `save_past_events=TRUE` in all `time_to_event` nodes. Support for `competing_events` nodes will be implemented in the future as well.

The other time-varying variables specified in the `tx_nodes` argument will still appear in the output, but it will only be the value that was observed at the last state of the simulation.

Optional columns created using a `time_to_event` ***node*****:**

When using a time-dependent node of type `"time_to_event"` with `event_count=TRUE` or `time_since_last=TRUE`, the columns created using either argument are **not** included in the output if `to="start_stop"`, but will be included if `to` is set to either `"long"` or `"wide"`. The reason for this behavior is that including these columns would lead to nonsense intervals in the start-stop format, but makes sense in the other formats.

What about `tx_nodes` ***that are not*** `time_to_event` ***nodes?*****:**

If you want the correct output for all `tx_nodes` and one or more of those are not `time_to_event` nodes, you will have to use `save_states="all"` in the original `sim_discrete_time` call. We plan to add support for `competing_events` with other `save_states` arguments in the near future. Support for arbitrary `tx_nodes` will probably take longer.

Value

Returns a single `data.table` (or `data.frame`) containing all simulated variables in the desired format.

Note

Using the node names `"start"`, `"stop"`, `".id"`, `".time"` or names that are automatically generated by time-dependent nodes of type `"time_to_event"` may break this function.

Author(s)

Robin Denz

References

Sy Han Chiou, Gongjun Xu, Jun Yan, and Chiung-Yu Huang (2023). "Regression Modeling for Recurrent Events Possibly with an Informative Terminal Event Using R Package reReg". In: Journal of Statistical Software. 105.5, pp. 1-34.

See Also

`sim_discrete_time`

Examples

```
library(simDAG)

set.seed(435345)

## exemplary car crash simulation, where the probability for
## a car crash is dependent on the sex, and the probability of death is
## highly increased for 3 days after a car crash happened
prob_car_crash <- function(data) {
  ifelse(data$sex==1, 0.001, 0.01)
```

```
}

prob_death <- function(data) {
  ifelse(data$car_crash_event, 0.1, 0.001)
}

dag <- empty_dag() +
  node("sex", type="rbernoulli", p=0.5) +
  node_td("car_crash", type="time_to_event", prob_fun=prob_car_crash,
          parents="sex", event_duration=3) +
  node_td("death", type="time_to_event", prob_fun=prob_death,
          parents="car_crash_event", event_duration=Inf)

# generate some data, only saving the last state
# not a problem here, because the only time-varying nodes are
# time-to-event nodes where the event times are saved
sim <- sim_discrete_time(dag, n_sim=20, max_t=500, save_states="last")

# transform to standard start-stop format
d_start_stop <- sim2data(sim, to="start_stop")
head(d_start_stop)

# transform to "death" centric start-stop format
# and keep only information until death, cause it's a terminal event
# (this could be used in a Cox model)
d_start_stop <- sim2data(sim, to="start_stop", target_event="death",
                         keep_only_first=TRUE, overlap=TRUE)
head(d_start_stop)

# transform to long-format
d_long <- sim2data(sim, to="long")
head(d_long)

# transform to wide-format
d_wide <- sim2data(sim, to="wide")
#head(d_wide)
```

`sim_discrete_event` *Simulate Data from a* DAG *with Time-Dependent Variables in Continuous Time*

Description

Similar to `sim_discrete_time`, this function allows users to generate complex data with time-varying variables from a DAG defined using `node` and `node_td` calls. In contrast to `sim_discrete_time`, time is modelled as a continuous variable using a discrete-event simulation approach. See details.

Usage

```
sim_discrete_event(dag, n_sim=NULL, t0_sort_dag=FALSE,
```

```
t0_data=NULL, t0_transform_fun=NULL,
t0_transform_args=list(),
max_t=Inf, remove_if, break_if,
max_iter=1000, redraw_at_t=NULL,
censor_at_max_t=FALSE, target_event=NULL,
keep_only_first=FALSE,
remove_not_at_risk=FALSE,
include_event_counts=TRUE,
check_inputs=TRUE, debug=NULL)
```

Arguments

dag	A DAG object created using the empty_dag function with node_td calls added to it (see details and examples). If the dag contains root nodes and child nodes which are time-fixed (those who were added using node calls), data according to this DAG will be generated for time = 0. That data will then be used as starting data for the following simulation. Alternatively, the user can specify the t0_data argument directly. In either case, the supplied dag needs to contain at least one time-dependent node of type "next_time", added using the node_td function. Other time-dependent node types are currently not supported.
n_sim	A single number specifying how many observations should be generated. If a data.table is supplied to the t0_data argument, this argument is ignored. The sample size will then correspond to the number of rows in t0_data.
t0_sort_dag	Corresponds to the sort_dag argument in the sim_from_dag function. Ignored if t0_data is specified and dag contains only time-dependent nodes.
t0_data	An optional data.table like object (also accepts a data.frame, tibble etc.) containing values for all relevant variables at $t = 0$. This dataset will then be transformed over time according to the nodes specified using node_td calls in dag. Alternatively, data for $t = 0$ may be generated automatically by this function if standard node calls were added to the dag. If t0_data is specified and the dag contains time-fixed nodes, all columns from t0_data are passed first and the nodes are then simulated on top using sim_from_dag.
t0_transform_fun	An optional function that takes the data created at $t = 0$ as the first argument. The function will be applied to the starting data and its output will replace the data.table. Can be used to perform arbitrary data transformations after the starting data was created. Set to NULL (default) to not use this functionality.
t0_transform_args	A named list of additional arguments passed to the t0_transform_fun. Ignored if t0_transform_fun=NULL.
max_t	A single number specifying the time needs to be reached for all individuals for the simulation to be terminated. This can be set to Inf, if the end of the simulation can be terminated through other means (e.g. the remove_if or break_if arguments, or when all variables have an event duration or immunity duration that is infinite).
remove_if	A condition that will be evaluated directly on the generated data.table after each jump to the next event. All rows for which the condition is TRUE are removed from the data at this point in time. The condition may contain names of

any variable that were generated. If all rows are removed through this condition, the simulation stops early. This argument may be useful to save computation time, if a large number of variables or many state changed need to be considered and the user only cares about the first time a condition is met for some individuals. Keep this argument unspecified (default) to not use this functionality.

`break_if` A condition that will be evaluated after each jump to the next event (but after subsetting, if `remove_if` was specified). If the condition is met, the simulation stops early. Contrary to the `remove_if` argument, this condition should return exactly one `TRUE` or `FALSE` value and is not directly evaluated on the `data`. To use variables generated in the simulation in this condition, users should use the `$` syntax (e.g. use `data$X` instead of just `X`). Keep this argument unspecified (default) to not use this functionality.

`max_iter` A single positive number, specifying the maximum amount of loops the simulation is allowed to run before it terminates. This argument exists so that if all of `max_t`, `remove_if` and `break_if` fail to terminate the simulation eventually, the code does not run forever. In nearly all cases it is, however, preferable to end the simulation using one of the other three arguments. A warning message is therefore returned whenever the simulation is stopped through reaching this limit.

`redraw_at_t` A numeric vector of positive values specifying times at which the time to the next event should be re-drawn, regardless of whether an event occurred at this time or not. This may be useful to specify effects or baseline probabilities that vary over discrete intervals of time. Note that using this argument potentially adds multiple additional rows to the output, in which no variables change. Set to `NULL` to not use this functionality (default).

`censor_at_max_t`

Either `TRUE` or `FALSE`, specifying whether the last generated time should be censored at the user-specified value of `max_t`. Since the simulation jumps times at events, the last observed event time may often be larger than `max_t` initially. Setting this to `TRUE` censors these values appropriately and potentially discards the last state-change.

`target_event` By default (keeping this argument at `FALSE`) all time-varying nodes are treated equally when creating the start-stop intervals. This can be changed by supplying a single character string to this argument, naming one of the nodes. This node will then be treated as the outcome. The output then corresponds to what would be needed to fit a Cox proportional hazards model with that node as the outcome.

`keep_only_first`

Only used when `target_event` is not `NULL`. Either `TRUE` or `FALSE` (default). If `TRUE`, all information after the first event per person will be discarded. Useful when `target_event` should be treated as a terminal variable.

`remove_not_at_risk`

Only used when `target_event` is not `NULL`. Either `TRUE` or `FALSE` (default). If `TRUE`, all information after an event that is recorded during the `immunity_duration` of an event (e.g. when the person is not at-risk for another event) is removed from the start-stop data. This may be needed when the goal is to fit time-to-event models to the data in some situations.

include_event_counts
: Either TRUE or FALSE, specifying whether event counts of time-dependent nodes in which event_count=TRUE was used should be included in the output or not.

check_inputs
: Whether to perform plausibility checks for the user input or not. Is set to TRUE by default, but can be set to FALSE in order to speed things up when using this function in a simulation study or something similar.

debug
: Strictly used for developmental purposes. Keep at NULL to ignore this argument.

Details

This function and the corresponding node interface implemented through the node_next_time function are still fairly new and may not have reached a stable state. The functionality is tested and should work fine, but the arguments, syntax and general functionality may still change in upcoming releases.

What is Discrete-Event Simulation?:

In discrete-event simulations (DES), the system is modelled as a sequence of distinct events that occur over time any may influence each other. In contrast to discrete-time simulations, the time in a DES is only advanced by some amount whenever an event occurs. The state of the system is then updated according to this event and the next advancement is made. The possibly simplest example of a DES, as compared to a discrete-time simulation is a system with just one variable, Y for a single individual. At the start, Y is zero. We are interested only in the time at which Y turns 1 for the first time. The probability of Y turning one in a single unit of time is set to a fixed value of 0.01. In a discrete-time simulation, we would perform a single Bernoulli trial with probability 0.01. If this trial returns a 1, we are finished and save the current simulation time. If the Bernoulli trial returns a 0, we increase the time by 1 and repeat the process until Y is eventually 1. In DES on the other hand, we would simply draw the time until Y turns 1 from a suitable distribution (in this example a simple exponential distribution with rate=0.01 would be sufficient).

In such simple cases, using a discrete-time simulation approach is clearly a worse strategy. There is no reason to perform so many computations when drawing a single exponentially distributed random number is enough. With more complex data generation processes, for example include time-varying variables that influence each other over time, using DES gets more complicated.

How it Works:

Internally, this function works by first simulating data using the sim_from_dag function. Alternatively, the user can supply a custom data.table using the t0_data argument. This data defines the state of all entities at $t = 0$. Afterwards, the following algorithm is used for each simulated individual:

(1) The time of the next change in a variable is generated for each of the included time-varying variable separately, possibly dependent on the other variables. (2) The minimum of these times is used as the new simulation time. (3) The variable corresponding to the choosen time is updated.

This process is repeated until no changes are needed anymore (e.g. when all time-dependent variables have reached absorbing states, or max_t is reached), or when a user-specified break condition is reached (argument break_if), or when no individuals are left after conditional subsetting (argument remove_if). After the first iteration, the times that are sampled in step (1) have to be sampled from left-truncated distributions, where the truncation time is equal to the current simulation time. This has to be the case because that time has already passed for that individual, so the next event change must be at least some time afterwards. Users may specify any function to calculate the rate

or probability used depending on the state. Users may also use any function to draw the time of the next change.

Specifying the `dag` ***argument***:

The `dag` argument should be specified as described in the `node` documentation page. More examples specific to discrete-event simulations can be found in the vignettes and the examples. The only difference to specifying a `dag` for the `sim_from_dag` function is that the `dag` here should contain at least one time-dependent node added using the `node_td` function, that uses `type="next_time"`.

Networks-Based Simulation:

Currently only time-constant networks added using the `network` function are supported. They are also only supported for data generation at $t = 0$. All following calculations will be made ignoring the network. If time-dependent networks or network dependencies in time-dependent variables are desired, the `sim_discrete_time` function has to be used instead.

Speed Considerations:

In general, this function should be *a lot faster* than a corresponding `sim_discrete_time` call, because it does not require going through all considered points in time directly. Its computation time therefore does not change substantially (or at all) with higher values of `max_t`. Instead, it only increases with higher values of `n_sim`, the amount of time-dependent variables and, most importantly, with the frequency by which these variables change. The more frequently a variable changes back and forth between `TRUE` and `FALSE`, the more iterations are needed and thus the more time is needed.

Current limitations:

Unlike the `sim_discrete_time` function, which does not assume any parametric distributions, this function requires the user to specify a function that may be used to generate the time of the next change in a binary variable. Multiple built-in options are provided, but it is nevertheless less flexible. Additionally, only binary time-dependent variables are supported (with no restrictions set on time-fixed variables). Some forms of dependencies are harder (but not impossible) to specify using the discrete-event approach.

For example, simulating time-dependent effects that are smooth functions of time is difficult (piecewise constant effects may be simulated easily using the `redraw_at_t` argument). If only the general event probability should vary over time, times may be generated using a Weibull or some other parametric functions (see associated vignette). In any case, users will have to know very clearly what functions to use and then have to provide a function that is able to generate truncated random values from this distribution. If these requirements cannot be met, discrete-time simulation may be the only alternative.

Value

Returns a single `data.table` including at least the following columns:

- `.id`: The unique individual identifier, coded as integers.
- `start`: The start of the time period, coded as a numeric value.
- `stop`: The end of the time period, coded as a numeric value.

Additionally, the returned data will include all time-constant and time-dependent variables that were generated. Some options on how this data should be formatted are given by the function itself (see `censor_at_max_t`, `target_event` and `keep_only_first`). The long- and wide-format are not supported, because the time is usually modelled as a continuous variable.

Author(s)

Robin Denz

References

Denz, Robin and Nina Timmesfeld (2026). "Simulating Complex Cross-Sectional and Longitudinal Data using the simDAG R Package". Journal of Statistical Software 116 (2), doi: 10.18637/jss.v116.i02.

Banks, Jerry, John S. Carson II, Barry L. Nelson, and David M. Nicol (2014). Discrete-Event System Simulation. Vol. 5. Edinburgh Gate: Pearson Education Limited.

Gillespie, Daniel T. 1976. "A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Reactions." Journal of Computational Physics 22 (4): 403–34.

Anderson, David F. 2007. "A Modified Next Reaction Method for Simulating Chemical Systems with Time Dependent Propensities and Delays." The Journal of Chemical Physics 127 (21).

See Also

empty_dag, node, node_td, node_next_time, sim_discrete_time

Examples

```
library(simDAG)

set.seed(454236)

## simulating death dependent on age, sex, bmi
## NOTE: this example is explained in detail in one of the vignettes

# initializing a DAG with nodes for generating data at t0
dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("bmi", type="gaussian", parents=c("sex", "age"),
       betas=c(1.1, 0.4), intercept=12, error=2)

# a function to calculate the probability of death as a
# linear combination of age, sex and bmi on the log scale
prob_death <- function(data, beta_age, beta_sex, beta_bmi,
                       beta_sickness, intercept) {
  prob <- intercept + data$age*beta_age + data$sex*beta_sex +
    data$bmi*beta_bmi + data$sickness*beta_sickness
  prob <- 1/(1 + exp(-prob))
  return(prob)
}

# adding time-dependent nodes to the dag
dag <- dag +
  node_td("sickness", type="next_time", prob_fun=0.01,
          event_duration=50, immunity_duration=Inf) +
  node_td("death", type="next_time", parents=c("age", "sex", "bmi"),
          prob_fun=prob_death, beta_age=0.1, beta_bmi=0.3, beta_sex=-0.2,
```

```
            beta_sickness=1.1,
            intercept=-20, event_duration=Inf)

# run simulation for 100 people, until everyone died
sim_dt <- sim_discrete_event(n_sim=100, dag=dag, max_t=Inf,
                             remove_if=death==TRUE,
                             target_event="death")
```

`sim_discrete_time` *Simulate Data from a* DAG *with Time-Dependent Variables in Discrete Time*

Description

Similar to the `sim_from_dag` function, this function can be used to generate data from a given DAG created using the `empty_dag` and `node` or `node_td` functions (and possibly `network` or `network_td` functions). In contrast to the `sim_from_dag` function, this function utilizes a discrete-time simulation approach. This is not an "off-the-shelves" simulation function, it should rather be seen as a "framework-function", making it easier to create discrete-time-simulations. It usually requires custom functions written by the user. See details.

Usage

```
sim_discrete_time(dag, n_sim=NULL, t0_sort_dag=FALSE,
                  t0_data=NULL, t0_transform_fun=NULL,
                  t0_transform_args=list(), max_t,
                  tx_nodes_order=NULL, tx_transform_fun=NULL,
                  tx_transform_args=list(),
                  remove_if, break_if,
                  save_states="last", save_states_at=NULL,
                  save_networks=FALSE,
                  verbose=FALSE, check_inputs=TRUE)
```

Arguments

dag	A DAG object created using the `empty_dag` function with `node_td` calls added to it (see details and examples). If the dag contains root nodes and child nodes which are time-fixed (those who were added using `node` calls), data according to this DAG will be generated for time = 0. That data will then be used as starting data for the following simulation. Alternatively, the user can specify the `t0_data` argument directly. In either case, the supplied dag needs to contain at least one time-dependent node added using the `node_td` function.
n_sim	A single number specifying how many observations should be generated. If a `data.table` is supplied to the `t0_data` argument, this argument is ignored. The sample size will then correspond to the number of rows in `t0_data`.
t0_sort_dag	Corresponds to the `sort_dag` argument in the `sim_from_dag` function. Ignored if `t0_data` is specified.

t0_data
: An optional data.table like object (also accepts a data.frame, tibble etc.) containing values for all relevant variables at $t = 0$. This dataset will then be transformed over time according to the nodes specified using node_td calls in dag. Alternatively, data for $t = 0$ may be generated automatically by this function if standard node calls were added to the dag.

t0_transform_fun
: An optional function that takes the data created at $t = 0$ as the first argument. The function will be applied to the starting data and its output will replace the data.table. Can be used to perform arbitrary data transformations after the starting data was created. Set to NULL (default) to not use this functionality.

t0_transform_args
: A named list of additional arguments passed to the t0_transform_fun. Ignored if t0_transform_fun=NULL.

max_t
: A single integer specifying the final point in time to which the simulation should be carried out. The simulation will start at $t = 1$ (after creating the starting data with the arguments above) and will continue until max_t by increasing the time by one unit at every step, updating the time-dependent nodes along the way.

tx_nodes_order
: A numeric vector specifying the order in which the time-dependent nodes added to the dag object using the node_td function should be executed at each time step. If NULL (default), the nodes will be generated in the order in which they were originally added.

tx_transform_fun
: An optional function that takes the data created after every point in time $t > 0$ as the first argument and the simulation time as the second argument. The function will be applied to that data after all node functions at that point in time have been executed and its output will replace the previous data.table. Can be used to perform arbitrary data transformations at every point in time. Set to NULL (default) to not use this functionality.

tx_transform_args
: A named list of additional arguments passed to the tx_transform_fun. Ignored if tx_transform_fun=NULL.

remove_if
: A condition that will be evaluated directly on the generated data.table at the beginning of each time-period. All rows for which the condition is TRUE are removed from the data at this point in time. The condition may contain names of any variable that were generated. If all individuals are removed through this condition, the simulation stops early. This argument may be useful to save computation time, if a large number of points in time should be considered and the user only cares about the first time a condition is met for some individuals. Keep this argument unspecified (default) to not use this functionality.

break_if
: A condition that will be evaluated at the beginning of each time-period (but after subsetting, if remove_if was specified). If the condition is met, the simulation stops early. Contrary to the remove_if argument, this condition should return exactly one TRUE or FALSE value and is not directly evaluated on the data. To use variables generated in the simulation in this condition, users should use the $ syntax (e.g. use data$X instead of just X). Keep this argument unspecified (default) to not use this functionality.

save_states	Specifies the amount of simulation states that should be saved in the output object. Has to be one of "all", "at_t" or "last" (default). If set to "all", a list of containing the data.table after every point in time will be added to the output object. If "at_t", only the states at specific points in time specified by the save_states_at argument will be saved (plus the final state). If "last", only the final state of the data.table is added to the output.
save_states_at	The specific points in time at which the simulated data.table should be saved. Ignored if save_states!="at_t".
save_networks	Either TRUE or FALSE, specifying whether networks should be saved over time. Only relevant if dag contains one or more network or network_td calls. If set to TRUE all networks (including time-independent ones) are saved according to the specification of the save_states argument.
verbose	If TRUE prints one line at every point in time before a node function is executed. This can be useful when debugging custom node functions. Defaults to FALSE.
check_inputs	Whether to perform plausibility checks for the user input or not. Is set to TRUE by default, but can be set to FALSE in order to speed things up when using this function in a simulation study or something similar.

Details

Sometimes it is necessary to simulate complex data that cannot be described easily with a single DAG and node information. This may be the case if the desired data should contain multiple time-dependent variables or time-to-event variables in which the event has time-dependent effects on other events. An example for this is data on vaccinations and their effects on the occurrence of adverse events (see vignette). Discrete-Time Simulation can be an effective tool to generate these kinds of datasets.

What is Discrete-Time Simulation?:

In a discrete-time simulation, there are entities who have certain states associated with them that only change at discrete points in time. For example, the entities could be people and the state could be alive or dead. In this example we could generate 100 people with some covariates such as age, sex etc.. We then start by increasing the simulation time by one day. For each person we now check if the person has died using a bernoulli trial, where the probability of dying is generated at each point in time based on some of the covariates. The simulation time is then increased again and the process is repeated until we reach max_t.

Due to the iterative process it is very easy to simulate arbitrarily complex data. The covariates may change over time in arbitrary ways, the event probability can have any functional relationship with the covariates and so on. If we want to model an event type that is not terminal, such as occurrence of cardiovascular disease, events can easily be simulated to be dependent on the timing and number of previous events. Since Discrete-Time Simulation is a special case of Discrete-Event Simulation, introductory textbooks on the latter can be of great help in getting a better understanding of the former.

How it Works:

Internally, this function works by first simulating data using the sim_from_dag function. Alternatively, the user can supply a custom data.table using the t0_data argument. This data defines the state of all entities at $t = 0$. Afterwards, the simulation time is increased by one unit and the data is transformed in place by calling each node function defined by the time-dependent nodes

which were added to the dag using the node_td function (either in the order in which they were added to the dag object or by the order defined by the tx_nodes_order argument). Usually, each transformation changes the state of the entities in some way. For example if there is an age variable, we would probably increase the age of each person by one time unit at every step. Once max_t is reached, the resulting data.table will be returned. It contains the state of all entities at the last step with additional information of when they experienced some events (if node_time_to_event was used as time-dependent node). Multiple in-depth examples can be found in the vignettes of this package.

Specifying the dag ***argument***:

The dag argument should be specified as described in the node documentation page. More examples specific to discrete-time simulations can be found in the vignettes and the examples. The only difference to specifying a dag for the sim_from_dag function is that the dag here should contain at least one time-dependent node added using the node_td function. Usage of the formula argument with non-linear or interaction terms is discouraged for performance reasons.

Networks-Based Simulation:

As in the sim_from_dag function, networks-based simulations are also directly supported. Users may define static networks (using the network function) and / or dynamic networks that may evolve over time(using the network_td function). By using the net function inside the formula argument of node or node_td calls, complex dependencies among observations depending on the neighbors of each observation may then be simulated. More information is given in the associated vignette and the documentation pages of network and network_td.

Speed Considerations:

All functions in this package rely on the data.table backend in order to make them more memory efficient and faster. It is however important to note that the time to simulate a dataset increases non-linearly with an increasing max_t value and additional time-dependent nodes. This is usually not a concern for smaller datasets, but if n_sim is very large (say > 1 million) this function will get rather slow. Note also that using the formula argument is a lot more computationally expensive than using the parents, betas approach to specify certain nodes.

In some cases, the remove_if or break_if arguments may reduce the computation time considerably. For example, if the user is only interested in the first time that some variable Y turns TRUE, it may make sense to use remove_if=Y==TRUE. Under the hood, the function then removes any individual where Y is already TRUE, so that the data shrinks and no further computations are performed for these individuals. Unfortunately, whether or not this actually does improve performance is dependent on multiple factors. With large n_sim and max_t, a constant or skewed probability distribution of Y and especially when expensive calculations are performed at each point in time, the performance gains may be very large. This is, however, not always the case. The added computational burden of actually doing the subsetting itself at each point in time may offset any performance gains or even deteriorate performance in other scenarios. We recommend checking the computation time on a single example with and without using either remove_if and/or break_if (if appropriate) and making the decision based on that small benchmark.

If speed is of particular importance, it may make sense to use the sim_discrete_event function instead. The discrete-event simulation framework is much faster, but a little less flexible than the discrete-time framework. See the relevant documentation page for more information.

What do I do with the output?:

This function outputs a simDT object, not a data.table. To obtain an actual dataset from the output of this function, users should use the sim2data function to transform it into the desired

format. Currently, the long-format, the wide-format and the start-stop format are supported. See sim2data for more information.

A Few Words of Caution:

In most cases it will be necessary for the user to write their own functions in order to actually use the sim_discrete_time function. Unlike the sim_from_dag function, in which many popular node types can be implemented in a re-usable way, discrete-time simulation will always require some custom input by the user. This is the price users have to pay for the almost unlimited flexibility offered by this simulation methodology.

Value

Returns a simDT object, containing some general information about the simulated data as well as the final state of the simulated dataset (and more states, depending on the specification of the save_states argument). In particular, it includes the following objects:

- past_states: A list containing the generated data at the specified points in time.
- past_networks: A list containing the generated / updated networks at the specified points in time.
- save_states: The value of the save_states argument supplied by the user.
- data: The data at time max_t. Note that if remove_if was used, this data may not include all n_sim individuals.
- data_t0: The data at time 0. Only included if remove_if was specified.
- tte_past_events: A list storing the times at which events happened in variables of type "time_to_event", if specified.
- ce_past_events: A list storing the times at which events happened in variables of type "competing_events", if specified.
- ce_past_causes: A list storing the types of events which happened at in variables of type "competing_events", if specified.
- tx_nodes: A list of all time-varying nodes, as specified in the supplied dag object.
- max_t: The value of max_t, as supplied by the user.
- d_max_t: A data.table containing n_sim rows and the two columns .id (unique person identifier) and max_t (the maximum time that an individual was actually included in the data generation). This is only included if remove_if was specified by the user.
- break_t: The time at which the simulation was stopped either because the break condition as defined in the break_if argument was first met, or the time at which no further individuals were included in the data because everyone was removed through the remove_if argument. If neither break_if nor remove_if were specified, this is simply equal to max_t.
- t0_var_names: A character vector containing the names of all variable names that do not vary over time.

To obtain a single dataset from this function that can be processed further, please use the sim2data function.

Author(s)

Robin Denz, Katharina Meiszl

References

Denz, Robin and Nina Timmesfeld (2026). "Simulating Complex Cross-Sectional and Longitudinal Data using the simDAG R Package". Journal of Statistical Software 116 (2), doi: 10.18637/jss.v116.i02.

Tang, Jiangjun, George Leu, und Hussein A. Abbass. 2020. Simulation and Computational Red Teaming for Problem Solving. Hoboken: IEEE Press.

Banks, Jerry, John S. Carson II, Barry L. Nelson, and David M. Nicol (2014). Discrete-Event System Simulation. Vol. 5. Edinburgh Gate: Pearson Education Limited.

See Also

empty_dag, node, node_td, sim2data, plot.simDT

Examples

```
library(simDAG)

set.seed(454236)

## simulating death dependent on age, sex, bmi
## NOTE: this example is explained in detail in one of the vignettes

# initializing a DAG with nodes for generating data at t0
dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("bmi", type="gaussian", parents=c("sex", "age"),
       betas=c(1.1, 0.4), intercept=12, error=2)

# a function that increases age as time goes on
node_advance_age <- function(data) {
  return(data$age + 1/365)
}

# a function to calculate the probability of death as a
# linear combination of age, sex and bmi on the log scale
prob_death <- function(data, beta_age, beta_sex, beta_bmi, intercept) {
  prob <- intercept + data$age*beta_age + data$sex*beta_sex + data$bmi*beta_bmi
  prob <- 1/(1 + exp(-prob))
  return(prob)
}

# adding time-dependent nodes to the dag
dag <- dag +
  node_td("age", type="advance_age", parents="age") +
  node_td("death", type="time_to_event", parents=c("age", "sex", "bmi"),
          prob_fun=prob_death, beta_age=0.1, beta_bmi=0.3, beta_sex=-0.2,
          intercept=-20, event_duration=Inf, save_past_events=FALSE)

# run simulation for 100 people, 50 days long
sim_dt <- sim_discrete_time(n_sim=100,
                            dag=dag,
```

```
                    max_t=50,
                    verbose=FALSE)
```

`sim_from_dag` *Simulate Data from a* `DAG`

Description

This function can be used to generate data from a given `DAG`. The `DAG` should be created using the `empty_dag` and `node` functions, which require the user to fully specify all variables, including information about distributions, beta coefficients and, depending on the node type, more parameters such as intercepts. Network dependencies among observations may also be included using the `network` function.

Usage

```
sim_from_dag(dag, n_sim, sort_dag=FALSE, return_networks=FALSE,
             check_inputs=TRUE)
```

Arguments

`dag`	A `DAG` object created using the `empty_dag` function with `node` calls (and potentially `network` calls) added to it using the `+` syntax. See details.
`n_sim`	A single number specifying how many observations should be generated.
`sort_dag`	Whether to topologically sort the DAG before starting the simulation or not. If the nodes in `dag` were already added in a topologically sorted manner, this argument can be kept at `FALSE`. It is recommended to not rely on this argument too heavily, because sorting may sometimes fail when only a `formula` is supplied to one or more `node` calls.
`return_networks`	Whether to also return networks that were included or generated due to the presence of `network` calls in the supplied `dag` or not. If set to `TRUE`, a named list of length 2 will be returned instead of only returning the generated data. Defaults to `FALSE`.
`check_inputs`	Whether to perform plausibility checks for the user input or not. Is set to `TRUE` by default, but can be set to `FALSE` in order to speed things up when using this function in a simulation study or something similar.

Details

How it Works:

First, `n_sim` i.i.d. samples from the root nodes are drawn. Children of these nodes are then generated one by one according to specified relationships and causal coefficients. For example, lets suppose there are two root nodes, `age` and `sex`. Those are generated from a normal distribution and a bernoulli distribution respectively. Afterward, the child node `height` is generated using both of these variables as parents according to a linear regression with defined coefficients, intercept and

sigma (random error). This works because every DAG has at least one topological ordering, which is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering. By using `sort_dag=TRUE` it is ensured that the nodes are processed in such an ordering.

This procedure is simple in theory, but can get very complex when manually coded. This function offers a simplified workflow by only requiring the user to define the `dag` object with appropriate information (see documentation of `node` function). A sample of size `n_sim` is then generated from the DAG specified by those two arguments.

Specifying the DAG:

Concrete details on how to specify the needed `dag` object are given in the documentation page of the `node` and `network` functions and in the vignettes of this package.

Can this function create longitudinal data?

Yes and no. It theoretically can, but only if the user-specified `dag` directly specifies a node for each desired point in time. Using the `sim_discrete_time` or `sim_discrete_event` functions is better in some cases. A brief discussion about this topic can be found in the vignettes of this package.

If time-dependent nodes were added to the `dag` using `node_td` calls, this function may not be used. Only the `sim_discrete_time` and `sim_discrete_event` functions will work in that case.

Networks-Based Simulation

In some cases the assumption that observations (rows) are independent from each other is not sufficient. This function allows to relax this assumption by directly supporting network-based dependencies among individuals. Users may specify one or multiple networks of dependencies between individuals and add those to the `dag` using the `network` function. It is then possible to use the `net` function inside the `formula` argument of `node` calls to directly make the value of that node dependent on some other variable values of its' neighbors in the network. See the documentation and the associated vignette for more information.

Value

If `return_networks=FALSE`, returns a single `data.table` including the simulated data with (at least) one column per node specified in `dag` and `n_sim` rows. Otherwise it returns a named list containing the `data` and the `networks` supplied or generated through the course of the simulation.

Author(s)

Robin Denz

References

Denz, Robin and Nina Timmesfeld (2026). "Simulating Complex Cross-Sectional and Longitudinal Data using the simDAG R Package". Journal of Statistical Software 116 (2), doi: 10.18637/jss.v116.i02.

See Also

`empty_dag`, `node`, `network`, `plot.DAG`, `sim_discrete_time`

Examples

```
library(simDAG)

set.seed(345345)

dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("bmi", type="gaussian", parents=c("sex", "age"),
       betas=c(1.1, 0.4), intercept=12, error=2)

sim_dat <- sim_from_dag(dag=dag, n_sim=1000)

# More examples for each directly supported node type as well as for custom
# nodes can be found in the documentation page of the respective node function
```

---

sim_n_datasets *Simulate multiple datasets from a single* DAG *object*

Description

This function takes a single DAG object and generates a list of multiple datasets, possible using parallel processing

Usage

```
sim_n_datasets(dag, n_sim, n_repeats, n_cores=1,
               data_format="raw", data_format_args=list(),
               seed=NULL, progressbar=TRUE, ...)
```

Arguments

dag	A DAG object created using the empty_dag function with nodes added to it using the + syntax. See ?empty_dag or ?node for more details. If the dag contains time-varying nodes added using the node_td function, the sim_discrete_time or sim_discrete_event functions will be used to generate the data (depending on the types of the included time-dependent nodes). Otherwise, the sim_from_dag function will be used.
n_sim	A single number specifying how many observations per dataset should be generated.
n_repeats	A single number specifying how many datasets should be generated.
n_cores	A single number specifying the amount of cores that should be used. If n_cores = 1, a simple for loop is used to generate the datasets with no parallel processing. If n_cores > 1 is used, the **doSNOW** package is used in conjunction with the **doRNG** package to generate the datasets in parallel. By using the **doRNG** package, the results are completely reproducible by setting a seed.

`data_format` An optional character string specifying the output format of the generated datasets, or a function. If `"raw"` (default), the dataset will be returned as generated by the respective data generation function. If the `dag` contains time-varying nodes added using the `node_td` function that are appropriate for discrete-time simulation and this argument is set to either `"start_stop"`, `"long"` or `"wide"`, the `sim2data` function will be called to transform the dataset into the defined format. If any other string is supplied, regardless of whether time-varying nodes are included in the `dag` or not, the function with the name given in the string is called to transform the data. If a function is supplied directly, it will also be applied. This can be any function. The only requirement is that it has a named argument called `data`. Arguments to the function can be set using the `data_format_args` argument (see below).

`data_format_args` An optional list of named arguments passed to the function specified by `data_format`. Set to `list()` to use no arguments. Ignored if `data_format="raw"`.

`seed` A seed for the random number generator. By supplying a value to this argument, the results will be replicable, even if parallel processing is used to generate the datasets (using `n_cores > 1`), thanks to the magic performed by the **doRNG** package. See details.

`progressbar` Either `TRUE` (default) or `FALSE`, specifying whether a progressbar should be used. Currently only works if `n_cores > 1`, ignored otherwise.

`...` Further arguments passed to the `sim_from_dag` function (if the `dag` does not contain time-varying nodes) or the `sim_discrete_time` / `sim_discrete_event` function (if the `dag` contains time-varying nodes).

Details

Generating a number of datasets from a single defined `dag` object is usually the first step when conducting Monte-Carlo simulation studies. This is simply a convenience function which automates this process using parallel processing (if specified).

Note that for more complex Monte-Carlo simulations this function may not be ideal, because it does not allow the user to vary aspects of the data-generation mechanism inside the main `for` loop, because it can only handle a single `dag`. For example, if the user wants to simulate `n_repeats` datasets with confounding and `n_repeats` datasets without confounding, he/she has to call this function twice. This is not optimal, because setting up the clusters for parallel processing takes some processing time. If many different `dags` should be used, it would make more sense to write a single function that generates the `dag` itself for each of the desired settings. This can sadly not be automated by us though.

Value

Returns a list of length `n_repeats` containing datasets generated according to the supplied `dag` object.

Note

In previous versions (< 0.4.1) the `seed` argument was set to `stats::runif(1)`, which is equivalent to using `seed=0`. This was a mistake, because it results in the same output being generated regard-

less of any set.seed call used before calling sim_n_datasets(). This default has been changed to NULL, which is equivalent to not setting a seed. To obtain the same results as in versions < 0.4.1 (when no 'seed' was specified), use seed=0.

Author(s)

Robin Denz

See Also

empty_dag, node, node_td, sim_from_dag, sim_discrete_time, sim2data, sim_discrete_event

Examples

```
library(simDAG)

# some example DAG
dag <- empty_dag() +
  node("death", type="binomial", parents=c("age", "sex"), betas=c(1, 2),
       intercept=-10) +
  node("age", type="rnorm", mean=10, sd=2) +
  node("sex", parents="", type="rbernoulli", p=0.5) +
  node("smoking", parents=c("sex", "age"), type="binomial",
       betas=c(0.6, 0.2), intercept=-2)

# generate 10 datasets without parallel processing
out <- sim_n_datasets(dag, n_repeats=10, n_cores=1, n_sim=100)

if (requireNamespace("doSNOW") & requireNamespace("doRNG") &
    requireNamespace("foreach")) {

# generate 10 datasets with parallel processing
out <- sim_n_datasets(dag, n_repeats=10, n_cores=2, n_sim=100)
}

# generate 10 datasets and transforming the output
# (using the sim2data function internally)
dag <- dag + node_td("CV", type="time_to_event", prob_fun=0.01)
out <- sim_n_datasets(dag, n_repeats=10, n_cores=1, n_sim=100,
                      max_t=20, data_format="start_stop")
```

Index

+.DAG *(add_node)*, 5
.N, *28*

aalen, *35*
add_node, 5
aftreg, *35*
ahreg, *35*
as.dagitty.DAG, 6
as.data.frame.simDT *(sim2data)*, 106
as.data.table.simDT *(sim2data)*, 106
as.igraph.DAG, *7*, 7, *10*, *94, 95*
as_tidy_dagitty.DAG, 9, *94, 95*

beta, *35*
binomial, *19*, *35*

calibrate_dag, 10, *23*, *90*
ceiling, *56*
competing_events, *35*
conditional_distr, *35*
conditional_prob, *19*, *35*
cox, *35*
create_layout, *9*
cut, *78*

dag2matrix, 16, *94*
dag_from_data, 18, *26*
dagitty, *7*
do, 20

ehreg, *35*
empty_dag, *3*, *5–10*, *16–18*, *20*, 22, *23*, *26*, *29*, *33, 34*, *41*, *43*, *47*, *50*, *52*, *59*, *61*, *63*, *66*, *68*, *70*, *73*, *76*, *78*, *85, 86*, *89*, *92*, *95*, *99*, *112*, *116, 117*, *122–125*, *127*

fcase, *65*

gamma, *35*
gaussian, *19*, *35*
geom_boxplot, *90*
glm, *42*, *60*, *75*

identity, *35*

lazy_dag, *10*, *12*, 23, *105*
long2start_stop, 24

makeGlmer, *43*, *76*
makeLmer, *42*, *60, 61*, *75*
matrix2dag, 26
mixture, *35*
multinomial, *35*

negative_binomial, *19*, *35*
net, 27, *30, 31*, *71*, *120*, *124*
network, *5*, *7–9*, *16*, *27, 28*, 29, *115*, *117*, *119, 120*, *123, 124*
network_td, *5*, *27*, *117*, *119, 120*
network_td *(network)*, 29
next_time, *35*
node, *3*, *5*, *7, 8*, *10*, *17, 18*, *20–23*, *26*, *28–31*, 33, *38*, *41*, *43*, *47, 48*, *50*, *52*, *56*, *59*, *61–63*, *65, 66*, *68*, *70*, *76*, *78*, *88, 89*, *92*, *94, 95*, *99*, *111, 112*, *115–118*, *120*, *122–124*, *127*
node_aalen, 37
node_aftreg *(node_rsurv)*, 79
node_ahreg *(node_rsurv)*, 79
node_beta, 39
node_binomial, 41, *68, 69*, *71*, *78*, *83*, *87, 88*
node_competing_events, 44, *68*, *85*
node_conditional_distr, 48, *65*
node_conditional_prob, 51, *65*
node_cox, 54, *72*
node_ehreg *(node_rsurv)*, 79
node_gamma, 57
node_gaussian, *34*, 59, *69*, *78*
node_identity, 62
node_mixture, *48*, *52*, 65
node_multinomial, 67

node_negative_binomial, 69, *87, 88*
node_next_time, 70, *114*, *116*
node_poisson, *69*, 74, *78*, *87, 88*
node_polr, 77
node_poreg *(*node_rsurv*)*, 79
node_rsurv, 79
node_td, *5–10*, *16–18*, *21–23*, *26*, *30*, *41*, *43*, *47*, *50*, *52*, *59*, *61–63*, *65, 66*, *68*, *70*, *73*, *76*, *78*, *85, 86*, *89*, *94, 95*, *99*, *111, 112*, *115–118*, *120*, *122*, *124–127*
node_td *(*node*)*, 33
node_time_to_event, *46, 47*, *72, 73*, 82, *109*, *120*
node_ypreg *(*node_rsurv*)*, 79
node_zeroinfl, 87

optim, *12*, *14*
optim2, *12*, *14*
optimh, *12*, *14*

plot.calibrated_DAG, *13*, 90
plot.DAG, 92, *124*
plot.simDT, 95, *122*
poisson, *19*, *35*
polr, *35*, *78*
poreg, *35*

raftreg, *79, 80*
rahreg, *79, 80*
rbernoulli, *34*, *49*, *52*, *83*, 99
rbeta, *39, 40*
rcategorical, *34*, *45*, *52*, *68*, 101, *103*
rconstant, *34*, *62*, 102
rehreg, *79, 80*
rexp, *104*
rgamma, *57, 58*
rnbinom, *69*
rnorm, *49*
rpois, *75*
rporeg, *79, 80*
rsample, *34*, 103
rtexp, *34*, *71*, 104
rypreg, *79, 80*

sample, *103*
set.seed, *127*
set_lazy_dag, *23*, 105
sim2data, 106, *120–122*, *126, 127*
sim_discrete_event, *3, 4*, *12*, *33, 34*, *36*, *56*, *70*, *72, 73*, *105*, 111, *120*, *124–127*
sim_discrete_time, *3–5*, *12*, *22*, *28–31*, *33, 34*, *36*, *41*, *43, 44*, *46, 47*, *50*, *52*, *56*, *59*, *61*, *63*, *66*, *68*, *70*, *72*, *76*, *78*, *82*, *84–86*, *89*, *95*, *99*, *105*, *107*, *110, 111*, *115, 116*, 117, *124–127*
sim_from_dag, *3–5*, *12*, *16*, *18, 19*, *22*, *26*, *28–31*, *33, 34*, *36*, *41*, *43*, *47*, *50*, *52*, *59*, *61*, *63*, *66*, *68*, *70*, *76*, *78*, *89*, *102*, *105*, *112*, *114, 115*, *117*, *119–121*, 123, *125–127*
sim_n_datasets, *12*, 125
simDAG-package, 3

time_to_event, *35*

uniroot, *11, 12*, *14*

ypreg, *35*

zeroinfl, *35*