Package: simDAG (via r-universe)

September 17, 2024

Title Simulate Data from a DAG and Associated Node Information

Version 0.2.0.9000

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 and more. Also includes a comprehensive framework for discrete-time simulation, which can generate even more complex longitudinal data.

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

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

VignetteBuilder knitr

Config/testthat/edition 3

Contact <robin.denz@rub.de>

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.1

Repository https://robindenz1.r-universe.dev

RemoteUrl https://github.com/robindenz1/simdag

RemoteRef HEAD

RemoteSha 118b40211fa3256795119edefecd23ba2f93a644

Contents

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

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` function additionally enables 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.

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` function.

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 nodes with generalized mixed linear models or more complex survival time models.

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 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_covid_example", package="simDAG")`
- `vignette(topic="v_using_formulas", package="simDAG")`

We are also working on a separate article on this package that is going to be published in a peer-reviewed journal.

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

The `sim_discrete_time` function 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

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` *object to a* `DAG` *object*

Description

This function allows users to add `DAG.node` objects created using the `node` or `node_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`	A `DAG.node` object created using the `node` function or `node_td` function.
`object_1`	Either a `DAG` object or a `DAG.node` 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 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 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.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, not any specifications. May be useful for plotting purposes.

Usage

```
## S3 method for class 'DAG'
as.igraph(x, ...)
```

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.
...	Currently not used.

Value

Returns a 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)
}
```

---

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)
```

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.

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("death", 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 node_custom. 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, type, data and return_model. The first three arguments should simply be added directly to 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). 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. These nodes defined with the names argument will be set to those values.

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. 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))
```

---

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)
```

`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)
```

---

node *Create a node object to grow a DAG step-by-step*

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` function or the `sim_discrete_time` function.

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. 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 currently only supported for nodes of type "binomial", "gaussian", "poisson", "negative_binomial" and "cox". 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 function or the sim_discrete_time function, 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 function is 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.

node_td should only be used if you are planning to perform a discrete-time simulation with the sim_discrete_time function. 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 three custom root-node functions:

- "rbernoulli": Draws randomly from a bernoulli distribution.
- "rcategorical": Draws randomly from any discrete probability density function.
- "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 linear regression.
- "binomial": A node based on logistic 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.
- "poisson": A node based on poisson regression.
- "negative_binomial": A node based on negative binomial regression.
- "cox": A node based on cox-regression.

For custom child node types, see below.

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.

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

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 on the `node_custom` documentation page.

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. For other node types, users may need to write custom functions to make this work. 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 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` 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

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_binomial *Simulate a Node Using Logistic Regression*

Description

Data from the parents is used to generate the node using logistic regression by predicting the covariate specific probability of 1 and sampling from a Bernoulli distribution accordingly.

Usage

```
node_binomial(data, parents, formula=NULL, betas, intercept,
              return_prob=FALSE, output="logical", labels=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. 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.
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.
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".

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 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:

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

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`.

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)
```

`node_competing_events` *Simulate a Time-to-Event Node 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` *Simulate a Node 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.

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` *Simulate a Node 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:

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

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.

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 *Simulate a Node Using Cox-Regression*

Description

Data from the parents is used to generate the node using cox-regression using the method of Bender et al. (2005).

Usage

```
node_cox(data, parents, formula=NULL, betas, surv_dist, lambda, gamma,
         cens_dist, cens_args, name)
```

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".
lambda	A single number used as parameter defined by surv_dist.
gamma	A single number used as parameter defined by surv_dist.
cens_dist	A single character naming the distribution function that should be used to generate the censoring times. 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.

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). 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 adds **two** columns to the resulting dataset instead of one. The first column is called paste0(name, "_event") 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.

To simulate more complex time-to-event data, the user may need to use the `sim_discrete_time` function instead.

Value

Returns a `data.table` of length `nrow(data)` containing two columns. Both starting with the nodes `name` and ending with `_event` and `_time`. The first is a logical vector, the second a numeric one.

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)

# define DAG
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)
```

---

node_custom *Create Your Own Function to Simulate a Root Node, Child Node or Time-Dependent Node*

Description

This page describes in detail how to define custom functions to allow the usage of root nodes, child nodes or time-dependent nodes that are not directly implemented in this package. By doing so, users may create data with any functional dependence they can think of.

Details

The number of available types of nodes is limited, but this package allows the user to easily implement their own node types by writing a single custom function. Users may create their own root nodes, child nodes and time-dependent nodes. The requirements for each node type are listed below. Some simple examples for each node type are given further below.

If you think that your custom node type might be useful to others, please contact the maintainer of this package via the supplied e-mail address or github and we might add it to this package.

Root Nodes:

Any function that generates some vector of size n with `n==nrow(data)`, or a `data.frame` with as many rows as the current data can be used as a child node. The only requirement is:

- **1.)** The function should have an argument called `n` which controls how many samples to generate.

Some examples that are already implemented in R outside of this package are `rnorm()`, `rgamma()` and `rbeta()`. The function may take any amount of further arguments, which will be passed through the three-dot syntax.

Child Nodes:

Again, almost any function may be used to generate a child node. Only four things are required for this to work properly:

- **1.)** Its' name should start with `node_` (if you want to use a string to define it in `type`).
- **2.)** It should contain an argument called `data` (contains the already generated data).
- **3.)** It should contain an argument called `parents` (contains a vector of the child nodes parents).
- **4.)** It should return either a vector of length `n_sim` or a `data.frame` with any number of columns and `n_sim` rows.

The function may include any amount of additional arguments specified by the user.

Time-Dependent Nodes:

By time-dependent nodes we mean nodes that are created using the `node_td` function. In general, this works in essentially the same way as for simple root nodes or child nodes. The requirements are:

- **1.)** Its' name should start with `node_` (if you want to use a string to define it in `type`).
- **2.)** It should contain an argument called `data` (contains the already generated data).
- **3.)** If it is a child node, it should contain an argument called `parents` (contains a vector of the child nodes parents). This is not necessary for nodes that are independently generated.
- **4.)** It should return either a vector of length `n_sim` or a `data.frame` with any number of columns and `n_sim` rows.

Again, any number of additional arguments is allowed and will be passed through the three-dot syntax. Additionally, users may add an argument to this function called `sim_time`. If included in the function definition, the current time of the simulation will be passed to the function on every call made to it. Similarly, the argument `past_states` may be added. If done so, a list containing all previous states of the simulation (as saved using the `save_states` argument of the `sim_discrete_time`) function) will be passed to it internally, giving the user access to the data generated at previous points in time.

Value

Should return either a vector of length `nrow(data)` or a `data.table` or `data.frame` with `nrow(data)` rows.

Author(s)

Robin Denz

Examples

```
library(simDAG)

set.seed(3545)

################ Custom Root Nodes ###################

# using external functions without defining them yourself can be done this way
dag <- empty_dag() +
  node("A", type="rgamma", shape=0.1, rate=2) +
  node("B", type="rbeta", shape1=2, shape2=0.3)

## define your own root node instead
# this function takes the sum of a normally distributed random number and an
# uniformly distributed random number
custom_root <- function(n, min=0, max=1, mean=0, sd=1) {
  out <- runif(n, min=min, max=max) + rnorm(n, mean=mean, sd=sd)
  return(out)
}

dag <- empty_dag() +
  node("A", type="custom_root", min=0, max=10, mean=5, sd=2)

# equivalently, the function can be supplied directly
dag <- empty_dag() +
  node("A", type=custom_root, min=0, max=10, mean=5, sd=2)

################ Custom Child Nodes ###################

# create a custom node function, which is just a gaussian node that
# includes (bad) truncation
node_gaussian_trunc <- function(data, parents, betas, intercept, error,
                                left, right) {
  out <- node_gaussian(data=data, parents=parents, betas=betas,
                       intercept=intercept, error=error)
  out <- ifelse(out <= left, left,
                ifelse(out >= right, right, out))
  return(out)
}

# another custom node function, which simply returns a sum of the parents
parents_sum <- function(data, parents, betas=NULL) {
  out <- rowSums(data[, parents, with=FALSE])
```

```
  return(out)
}

# an example of using these new node types in a simulation
dag <- empty_dag() +
  node("age", type="rnorm", mean=50, sd=4) +
  node("sex", type="rbernoulli", p=0.5) +
  node("custom_1", type="gaussian_trunc", parents=c("sex", "age"),
       betas=c(1.1, 0.4), intercept=-2, error=2, left=10, right=25) +
  node("custom_2", type=parents_sum, parents=c("age", "custom_1"))

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

########## Custom Time-Dependent Nodes ###############

## example for a custom time-dependent node with no parents
# this node simply draws a new value from a normal distribution at
# each point in time
node_custom_root_td <- function(data, n, mean=0, sd=1) {
  return(rnorm(n=n, mean=mean, sd=sd))
}

n_sim <- 100

dag <- empty_dag() +
  node_td(name="Something", type=node_custom_root_td, n=n_sim, mean=10, sd=5)

sim <- sim_discrete_time(dag, n_sim=n_sim, max_t=10)

## example for a custom time-dependent child node
# draw from a normal distribution with different specifications based on
# whether a previously updated time-dependent node is currently TRUE
node_custom_child <- function(data, parents) {
  out <- numeric(nrow(data))
  out[data$other_event] <- rnorm(n=sum(data$other_event), mean=10, sd=3)
  out[!data$other_event] <- rnorm(n=sum(!data$other_event), mean=5, sd=10)
  return(out)
}

dag <- empty_dag() +
  node_td("other", type="time_to_event", prob_fun=0.1) +
  node_td("whatever", type="custom_child", parents="other_event")

sim <- sim_discrete_time(dag, n_sim=50, max_t=10)

## using the sim_time argument in a custom node function
# this function returns a continuous variable that is simply the
# current simulation time squared
node_square_sim_time <- function(data, sim_time, n_sim) {
  return(rep(sim_time^2, n=n_sim))
}

# note that we should not actually define the sim_time argument in the
```

```
# node_td() call below, because it will be passed internally, just like data
dag <- empty_dag() +
  node_td("unclear", type=node_square_sim_time, n_sim=100)

sim <- sim_discrete_time(dag, n_sim=100, max_t=10)

## a node using previous states of the simulation

# this function simply returns the value used two simulation time steps ago +
# a normally distributed random value
node_prev_state <- function(data, past_states, sim_time) {
  if (sim_time < 3) {
    return(rnorm(n=nrow(data)))
  } else {
    return(past_states[[sim_time-2]]$A + rnorm(n=nrow(data)))
  }
}

# note that we again do not specify the sim_time and past_states argument
# directly here, because they are set internally
dag <- empty_dag() +
  node_td("A", type=node_prev_state, parents="A")

# save_states="all" is needed, because we use them internally
sim <- sim_discrete_time(dag, n_sim=100, max_t=10, save_states="all")
```

---

node_gaussian *Simulate a Node Using Linear Regression*

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.

Usage

```
node_gaussian(data, parents, formula=NULL, betas, intercept, error)
```

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.

`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.

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:

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

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)
```

---

node_multinomial *Simulate a Node Using Multinomial Regression*

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.

Since this function produces categorical output (as it should), it may be difficult to use this node type as a parent for other nodes. Nevertheless, it is of course possible using a user-defined node type (see node_custom for some infos on how to define those).

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` *Simulate a Node Using Negative Binomial Regression*

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)
```

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.
`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`).

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`, exponentiating it and passing it to the `mu` argument of the `rnbinom` function of the **stats** package.

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_poisson *Simulate a Node Using Poisson Regression*

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.

Usage

```
node_poisson(data, parents, formula=NULL, betas, intercept)
```

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.
intercept	A single number specifying the intercept that should be used when generating the node.

Details

Essentially, this function simply calculates the linear predictor defined by the betas-coefficients, the intercept and the values of the parents. The exponential 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 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:

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

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)
```

---

node_time_to_event *Simulate a Time-to-Event Node 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, name,
                   prob_fun, ..., event_duration=1,
                   immunity_duration=event_duration,
                   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.
`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.

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. 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.

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. 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)

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

---

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_spere", "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. 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` function to obtain an adjacency matrix from the `DAG` object and directly use this matrix and the **igraph** package (or similar ones) to get much better plots.

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

See Also

`empty_dag`, `node`, `node_td`

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` *Create a Simple 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")
```

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"`.

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")
```

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.

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")
```

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, 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,
              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,
              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.

`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. 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_time` *Using Discrete-Time Simulation to Generate Complex Data from a Given DAG and Node Information*

Description

Similar to the `sim_from_dag` function, this function can be used to generate data from a given DAG. 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=TRUE,
                  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(),
                  save_states="last", save_states_at=NULL,
                  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 node functions specified in tx_nodes. 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.
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".

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.

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.

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.
- `save_states`: The value of the `save_states` argument supplied by the user.
- `data`: The data at time `max_t`.
- `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.

- `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

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 Given DAG and Node Information*

Description

This function can be used to generate data from a given DAG. It additionally requires information on node distributions, beta coefficients and, depending on the node type, more parameters such as intercepts.

Usage

```
sim_from_dag(dag, n_sim, sort_dag=FALSE, check_inputs=TRUE)
```

Arguments

`dag`	A `DAG` object created using the `empty_dag` function with nodes 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` to safe some computation time. This usually won't safe too much time though, because it internally uses the `topological_sort` function from the **Rfast** package, which is very fast.
`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` function 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` 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` function will work in that case.

Value

Returns a single `data.table` including the simulated data with (at least) one column per node specified in `dag` and `n_sim` rows.

Author(s)

Robin Denz

See Also

`empty_dag`, `node`, `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 *Generate 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=parallel::detectCores(),
               data_format="raw", data_format_args=list(),
               seed=stats::runif(1), 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 function will be used to generate the data. 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. 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 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. 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.

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 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.

Author(s)

Robin Denz

See Also

empty_dag, node, node_td, sim_from_dag, sim_discrete_time, sim2data

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