Package 'simcausal'

May 9, 2026

Type Package

Version 0.5.7

Title Simulating Longitudinal Data with Causal Inference Applications

Description A flexible tool for simulating complex longitudinal data using structural equations, with emphasis on problems in causal inference. Specify interventions and simulate from intervened data generating distributions. Define and evaluate treatment-specific means, the average treatment effects and coefficients from working marginal structural models. User interface designed to facilitate the conduct of transparent and reproducible simulation studies, and allows concise expression of complex functional dependencies for a large number of time-varying nodes. See the package vignette for more information, documentation and examples.

URL https://github.com/osofr/simcausal

BugReports https://github.com/osofr/simcausal/issues

Depends R (>= 3.2.0)

Imports data.table, igraph, stringr, R6, assertthat, Matrix, methods

Suggests copula, RUnit, ltmle, knitr, ggplot2, Hmisc, mvtnorm, bindata

VignetteBuilder knitr

License GPL-2

RoxygenNote 7.3.1

Encoding UTF-8

NeedsCompilation no

Author Oleg Sofrygin [aut],
Mark J. van der Laan [aut],
Romain Neugebauer [aut],
Fred Gruber [ctb, cre]

Maintainer Fred Gruber <fgruber@gmail.com>

Repository CRAN

Date/Publication 2024-10-19 14:30:02 UTC

Contents

A *Subsetting/Indexing Actions Defined for* DAG *Object*

Description

Subsetting/Indexing Actions Defined for DAG Object

Usage

```
A(DAG)
```

Arguments

DAG	A DAG object that was defined using functions node, set.DAG and action.

Value

returns a list of actions, which are intervened versions of the original observed data DAG.

Examples

```
D <- DAG.empty()
D <- D + node(name="W1", distr="rbern", prob=plogis(-0.5))
D <- D + node(name="W2", distr="rbern", prob=plogis(-0.5 + 0.5*W1))
D <- D + node(name="A", distr="rbern", prob=plogis(-0.5 + 0.5*W1+ 0.5*W2))
D <- set.DAG(D)
# Define two actions, acting on node "A"
D <- D + action("A0", nodes=node("A", distr="rbern", prob=0))
D <- D + action("A1", nodes=node("A", distr="rbern", prob=1))
# Select both actions
A(D)
# Select action "A1" only
A(D)["A1"]
```

add.action *Define and Add Actions (Interventions)*

Description

Define and add new action (intervention) to the existing DAG object. Use either syntax DAG + action(name = ,nodes =) or add.action((DAG = ,name = ,nodes =). Both give identical results, see the examples in the vignette and below for details.

Usage

```
add.action(DAG, name, nodes, ..., attr = list())

action(...)
```

Arguments

DAG	DAG object
name	Unique name of the action
nodes	A list of node objects that defines the action on the DAG (replaces the distributions of the corresponding nodes in DAG)
...	Additional named attributes defining / indexing the action
attr	Additional named attributes defining / indexing the action

Details

In addition to the action name and list of action nodes, both of these functions accept arbitrary named attributes (as additional arguments which must be given a name). This additional attributes can be used to simplify specification of dynamic regimes (actions that depend on the past observed covariates).

The formula of the intervention node is allowed to contain undefined variables, as long as those are later defined as a named argument to `action`.

In Example 2 below, `node("A",..., mean = ifelse(W1 >= theta, 1, 0))`, defines the mean of the node "A" as a function of some undefined variable `theta`, setting `A` to 1 if the baseline node `W1` is above or equal to `theta` and 0 vice versa. One specifies actual values of `theta` while defining a new action, possible creating a series of actions, each indexed by a different value of `theta`. A new action can be defined with `D<-D+action("A1th0.1", nodes=actN, theta=0.1)`.

Note that any name can be used in place of `theta`. This attribute variable can appear anywhere inside the node distribution formula. Finally, the attribute variable can also be time varying and, just like with DAG nodes, can be indexed by square bracket notation, `theta[t]`. See Example 3 for defining time-varying attributes.

Value

A modified `DAG` object with the added action

Examples

```
#----------------------------------------------------------------------------------------
# EXAMPLE 1: Showing two equivalent ways of defining an action for a simple DAG
#----------------------------------------------------------------------------------------

D <- DAG.empty()
D <- D + node(name="W1", distr="rbern", prob=plogis(-0.5))
D <- D + node(name="W2", distr="rbern", prob=plogis(-0.5 + 0.5*W1))
D <- D + node(name="A", distr="rbern", prob=plogis(-0.5 + 0.5*W1+ 0.5*W2))
Dset <- set.DAG(D)

# Syntax '+ action': define two actions, intervening on node "A", imputing order
Dset <- Dset + action("A0", nodes=node("A", distr="rbern", prob=0))
Dset <- Dset + action("A1", nodes=node("A", distr="rbern", prob=1))

# Equivalent syntax 'add.action': define two actions, intervening on node "A"
Dset <- add.action(Dset, "A0", nodes=node("A", distr="rbern", prob=0))
```

```
Dset <- add.action(Dset, "A1", nodes=node("A", distr="rbern", prob=1))

#----------------------------------------------------------------------------------------
# EXAMPLE 2: Adding named attributes that define (index) the action.
# Define intervention on A that is conditional on W1 crossing some threshold theta
#----------------------------------------------------------------------------------------

# Redefining node W1 as uniform [0,1]
D <- DAG.empty()
D <- D + node(name="W1", distr="runif", min=0, max=1)
D <- D + node(name="W2", distr="rbern", prob=plogis(-0.5 + 0.5*W1))
D <- D + node(name="A", distr="rbern", prob=plogis(-0.5 + 0.5*W1+ 0.5*W2))
Dset <- set.DAG(D)

# Define a node that is indexed by unknown variable theta
actN <- node("A",distr="rbern",prob=ifelse(W1 >= theta,1,0))
# Define 3 actions for theta=0.1, 0.5, 0.9
Dset <- Dset + action("A1th0.1", nodes = actN, theta = 0.1)
Dset <- Dset + action("A1th0.5", nodes = actN, theta = 0.5)
Dset <- Dset + action("A1th0.9", nodes = actN, theta = 0.9)

# Simulate 50 observations per each action above
simfull(A(Dset), n=50)

#----------------------------------------------------------------------------------------
# EXAMPLE 3: Time-varying action attributes for longitudinal DAG
#----------------------------------------------------------------------------------------
# Define longitudinal data structure over 6 time-points t=(0:5) with survival outcome "Y"
t_end <- 5
D <- DAG.empty()
D <- D + node("L2", t=0, distr="rbern", prob=0.05)
D <- D + node("L1", t=0, distr="rbern", prob=ifelse(L2[0]==1,0.5,0.1))
D <- D + node("A1", t=0, distr="rbern", prob=ifelse(L1[0]==1, 0.5, 0.1))
D <- D + node("Y",  t=0, distr="rbern",
               prob=plogis(-6.5 + L1[0] + 4*L2[0] + 0.05*I(L2[0]==0)), EFU=TRUE)
D <- D + node("L2", t=1:t_end, distr="rbern", prob=ifelse(A1[t-1]==1, 0.1, 0.9))
D <- D + node("A1", t=1:t_end, distr="rbern",
             prob=ifelse(A1[t-1]==1, 1, ifelse(L1[0]==1 & L2[0]==0, 0.3, 0.5)))
D <- D + node("Y",  t=1:t_end, distr="rbern", prob=plogis(-6.5+L1[0]+4*L2[t]), EFU=TRUE)
D <- set.DAG(D)

#----------------------------------------------------------------------------------------
# Dynamic actions indexed by constant value of parameter theta={0,1})
#----------------------------------------------------------------------------------------
# Define time-varying node A1: sets A1 to 1 if L2 at t is >= theta
actN_A1 <- node("A1",t=0:t_end, distr="rbern", prob=ifelse(L2[t] >= theta,1,0))

# Define two actions, indexed by fixed values of theta={0,1}
D_act <- D + action("A1_th0", nodes=actN_A1, theta=0)
D_act <- D_act + action("A1_th1", nodes=actN_A1, theta=1)

# Simulate 50 observations for per each action above
simfull(simcausal::A(D_act), n=50)
```

```
#-----------------------------------------------------------------------------------
# Dynamic actions indexed by time-varying parameter theta[t]
#-----------------------------------------------------------------------------------
# This defines an action node with threshold theta varying in time (note syntax theta[t])
actN_A1 <- node("A1",t=0:t_end, distr="rbern", prob=ifelse(L2[t] >= theta[t],1,0))

# Now define 3 actions that are indexed by various values of theta over time
D_act <- D + action("A1_th_const0", nodes=actN_A1, theta=rep(0,(t_end+1)))
D_act <- D_act + action("A1_th_var1", nodes=actN_A1, theta=c(0,0,0,1,1,1))
D_act <- D_act + action("A1_th_var2", nodes=actN_A1, theta=c(0,1,1,1,1,1))

# Simulate 50 observations for per each action above
simfull(simcausal::A(D_act), n=50)
```

`add.nodes` *Adding Node(s) to DAG*

Description

Adding nodes to a growing DAG object, as in `DAG + node()`. Use either syntax `DAG + node()` or `add.nodes(DAG = , nodes = node())`. Both give identical results, see the examples in the vignette and below for details.

Usage

```
add.nodes(DAG, nodes)

## S3 method for class 'DAG'
obj1 + obj2
```

Arguments

`DAG`	DAG object
`nodes`	A node or several nodes returned from a call to `node` function. If the node(s) under same name(s) already exist, the old node(s) get overwritten.
`obj1`	Object that belongs to either classes: `DAG`, `DAG.action` or `DAG.nodelist`.
`obj2`	Object that belongs to either classes: `DAG`, `DAG.action` or `DAG.nodelist`.

Value

An updated DAG object with new nodes

See Also

`node`

DAG.empty *Initialize an empty DAG object*

Description

Initialize an empty DAG object

Usage

```
DAG.empty()
```

Define_sVar *Class for defining and evaluating user-specified summary measures (exprs_list)*

Description

Evaluates and and stores arbitrary summary measure expressions. The expressions (exprs_list) are evaluated in the environment of the input data.frame.

Format

An R6 class object.

Details

Following fields are created during initialization

- nodes ...
- subset_regs ...
- sA_nms ...
- sW_nms ...
- Kmax ...

Methods

Public methods:

- `Define_sVar$new()`
- `Define_sVar$set.new.exprs()`
- `Define_sVar$eval.nodeforms()`
- `Define_sVar$eval.EFU()`
- `Define_sVar$df.names()`
- `Define_sVar$setnode.setenv()`

- Define_sVar$set.net()
- Define_sVar$clone()

Method `new()`**:**

Usage:

```
Define_sVar$new()
```

Method `set.new.exprs()`**:**

Usage:

```
Define_sVar$set.new.exprs(exprs_list)
```

Method `eval.nodeforms()`**:**

Usage:

```
Define_sVar$eval.nodeforms(cur.node, data.df)
```

Method `eval.EFU()`**:**

Usage:

```
Define_sVar$eval.EFU(cur.node, data.df)
```

Method `df.names()`**:**

Usage:

```
Define_sVar$df.names(data.df)
```

Method `setnode.setenv()`**:**

Usage:

```
Define_sVar$setnode.setenv(cur.node)
```

Method `set.net()`**:**

Usage:

```
Define_sVar$set.net(netind_cl)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Define_sVar$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

DF.to.long — *Convert Data from Wide to Long Format Using* reshape

Description

This utility function takes a simulated data.frame in wide format as an input and converts it into a long format (slower compared to DF.to.longDT).

Usage

```
DF.to.long(df_wide)
```

Arguments

df_wide	A data.frame in wide format

Details

Keeps all covariates that appear only once and at the first time-point constant (carry-forward).

All covariates that appear fewer than range(t) times are imputed with NA for missing time-points.

Observations with all NA's for all time-varying covariates are removed.

When removing NA's the time-varying covariates that are attributes (attnames) are not considered.

Value

A data.frame object in long format

See Also

DF.to.longDT - a faster version of DF.to.long that uses data.table package

Other data manipulation functions: DF.to.longDT(), doLTCF()

DF.to.longDT — *Faster Conversion of Data from Wide to Long Format Using* dcast.data.table

Description

Faster utility function for converting wide-format data.frame into a long format. Internally uses **data.table** package functions melt.data.table and dcast.data.table.

Usage

```
DF.to.longDT(df_wide, return_DF = TRUE)
```

Arguments

df_wide	A data.frame or data.table in wide format
return_DF	TRUE (default) to return a data.frame, FALSE returns a data.table

Details

Keeps all covariates that appear only once and at the first time-point constant (carry-forward).

All covariates that appear fewer than range(t) times are imputed with NA for missing time-points.

Observations with all NA's for all time-varying covariates are removed.

When removing NA's the time-varying covariates that are attributes (attnames) are not considered.

Value

A data.frame in long format

See Also

Other data manipulation functions: DF.to.long(), doLTCF()

---

distr.list *List All Custom Distribution Functions in* simcausal.

Description

List All Custom Distribution Functions in simcausal.

Usage

```
distr.list()
```

---

doLTCF *Missing Variable Imputation with Last Time Point Value Carried Forward (LTCF)*

Description

Forward imputation for missing variable values in simulated data after a particular end of the follow-up event. The end of follow-up event is defined by the node of type EOF=TRUE being equal to 1.

Usage

```
doLTCF(data, LTCF)
```

Arguments

data	Simulated data.frame in wide format
LTCF	Character string specifying the outcome node that is the indicator of the end of follow-up (observations with value of the outcome variable being 1 indicate that the end of follow-up has been reached). The outcome variable must be a binary node that was declared with EFU=TRUE.

Value

Modified data.frame, all time-varying missing variables after the EFU outcome specified in LTCF are forward imputed with their last available non-missing value.

Details

The default behavior of the sim function consists in setting all nodes that temporally follow an EFU node whose simulated value is 1 to missing (i.e., NA). The argument LTCF of the sim function can however be used to change this default behavior and impute some of these missing values with *last time point value carried forward* (LTCF). More specifically, only the missing values of time-varying nodes (i.e., those with non-missing t argument) that follow the end of follow-up event encoded by the EFU node specified by the LTCF argument will be imputed. One can use the function doLTCF to apply the *last time point value carried forward* (LTCF) imputation to an existing simulated dataset obtained from the function sim that was called with its default imputation setting (i.e., with no LTCF argument). Illustration of the use of the LTCF imputation functionality are provided in the package vignette.

The first example below shows the default data format of the sim function after an end of the follow-up event and how this behavior can be modified to generate data with LTCF imputation by either using the LTCF argument of the sim function or by calling the doLTCF function. The second example demonstrates how to use the doLTCF function to perform LTCF imputation on already existing data simulated with the sim function based on its default non-imputation behavior.

See Also

sim, simobs and simfull for simulating data with and without carry forward imputation.

Other data manipulation functions: DF.to.long(), DF.to.longDT()

Examples

```
t_end <- 10
lDAG <- DAG.empty()
lDAG <- lDAG +
node(name = "L2", t = 0, distr = "rconst", const = 0) +
node(name = "A1", t = 0, distr = "rconst", const = 0) +
node(name = "L2", t = 1:t_end, distr = "rbern",
  prob = ifelse(A1[t - 1]  ==  1, 0.1,
  ifelse(L2[t-1] == 1, 0.9,
        min(1,0.1 + t/t_end)))) +
node(name = "A1", t = 1:t_end, distr = "rbern",
  prob = ifelse(A1[t - 1]  ==  1, 1,
   ifelse(L2[0] == 0, 0.3,
```

```
  ifelse(L2[0] == 0, 0.1,
   ifelse(L2[0] == 1, 0.7, 0.5))))) +
node(name = "Y", t = 1:t_end, distr = "rbern",
  prob = plogis(-6.5 + 4 * L2[t] + 0.05 * sum(I(L2[0:t] == rep(0,(t + 1))))),
  EFU = TRUE)
lDAG <- set.DAG(lDAG)
#----------------------------------------------------------------------------------------
# EXAMPLE 1. No forward imputation.
#----------------------------------------------------------------------------------------
Odat.wide <- sim(DAG = lDAG, n = 1000, rndseed = 123)
Odat.wide[c(21,47), 1:18]
Odat.wideLTCF <- sim(DAG = lDAG, n = 1000, LTCF = "Y", rndseed = 123)
Odat.wideLTCF[c(21,47), 1:18]
#----------------------------------------------------------------------------------------
# EXAMPLE 2. With forward imputation.
#----------------------------------------------------------------------------------------
Odat.wideLTCF2 <- doLTCF(data = Odat.wide, LTCF = "Y")
Odat.wideLTCF2[c(21,47), 1:18]
# all.equal(Odat.wideLTCF, Odat.wideLTCF2)
```

`eval.target` *Evaluate the True Value of the Causal Target Parameter*

Description

This function estimates the true value of the previously set target parameter (`set.targetE` or `set.targetMSM`) using the DAG object and either 1) `data`: list of action-specific simulated `data.frames`; or 2) `actions`; or 3) when `data` and `actions` are missing, using all distinct actions previously defined on the `DAG` object.

Usage

```
eval.target(
  DAG,
  n,
  data,
  actions,
  rndseed = NULL,
  verbose = getOption("simcausal.verbose")
)
```

Arguments

`DAG`	DAG object with target parameter set via `set.targetE` or `set.targetMSM` functions
`n`	Number of observations to simulate (if simulating full data), this is overwritten by the number of observations in each data
`data`	List of action-specific `data.frames` generated with `sim` or `simfull`

actions	Character vector of action names which play the role of the data generating mechanism for simulated data when argument data is missing. Alternatively, actions can be a list of action DAGs pre-selected with A(DAG) function. When this argument is missing, full data is automatically sampled from all available actions in the DAG argument.
rndseed	Seed for the random number generator.
verbose	Set to TRUE to print messages on status and information to the console. Turn this off by default using options(simcausal.verbose=FALSE).

Details

For examples and additional details see documentation for set.targetE or set.targetMSM

Value

For targetE returns a vector of counterfactual means, ATE or ATR; for targetMSM returns a named list with the MSM model fit ("msm"), MSM model coefficients ("coef"), the mapping of the MSM summary terms S() to the actual variable names used in the data, ("S.msm.map"), and the long format full data that was used for fitting this MSM "df_long".

---

igraph.to.sparseAdjMat

Convert igraph Network Object into Sparse Adjacency Matrix

Description

Convert igraph network object into its sparse adjacency matrix representation using as_adjacency_matrix function from the igraph package.

Usage

```
igraph.to.sparseAdjMat(igraph_network)
```

Arguments

igraph_network	Network as an igraph object

Value

Sparase adjacency matrix returned by igraph::as_adjacency_matrix function. NOTE: for directed graphs the friend IDs pointing into vertex i are assumed to be listed in the column i (i.e, which(adjmat[,i]) are friends of i).

See Also

network; sparseAdjMat.to.NetInd; NetInd.to.sparseAdjMat; sparseAdjMat.to.igraph;

N *Subsetting/Indexing* DAG *Nodes*

Description

Subsetting/Indexing DAG Nodes

Usage

```
N(DAG)
```

Arguments

DAG	A DAG object that was defined using functions node and set.DAG.

Value

returns a list of nodes that can be indexed as a typical named list "[[]]".

Examples

```
D <- DAG.empty()
D <- D + node(name="W1", distr="rbern", prob=plogis(-0.5))
D <- D + node(name="W2", distr="rbern", prob=plogis(-0.5 + 0.5*W1))
D <- set.DAG(D)
#Returns all nodes from DAG D
N(D)
#Returns node W1 from DAG D
N(D)["W1"]
```

net.list *List All Custom Network Generator Functions in* simcausal.

Description

List All Custom Network Generator Functions in simcausal.

Usage

```
net.list()
```

NetInd.to.sparseAdjMat *Convert Network IDs Matrix into Sparse Adjacency Matrix*

Description

Convert simcausal network ID matrix (NetInd_k) into a network represented by a sparse adjacency matrix.

Usage

```
NetInd.to.sparseAdjMat(NetInd_k, nF)
```

Arguments

NetInd_k	Matrix of network IDs of dimension (n=nrow(sparseAdjMat),Kmax), where each row i consists of the network IDs (row number of friends) of observation i. Remainders are filled with NAs.
nF	Integer vector of length n specifying the number of friends for each observation.

Value

Network represented as a sparse adjacency matrix (S4 class object dgCMatrix from package Matrix). NOTE: The friend IDs for observation i will be listed in column i (i.e, which(sparseAdjMat[,i]) are friends of i).

See Also

network; sparseAdjMat.to.igraph; igraph.to.sparseAdjMat; sparseAdjMat.to.NetInd;

NetIndClass *R6 class for creating and storing a friend matrix (network IDs) for network data*

Description

This R6 class defines fields and methods for creating and storing NetInd_k, a matrix of friend indices (network IDs) of dim = (nobs x Kmax).

Format

An R6Class generator object

Details

- NetInd - Matrix of friend indices (network IDs) of `dim = (nobs x Kmax)` (Active Binding).
- nF - Vector of integers, where `nF[i]` is the integer number of friends (0 to `Kmax`) for observation `i`.
- nobs - Number of observations
- Kmax - Maximum number of friends for any observation.

Methods

`new(nobs, Kmax = 1)` Uses `nobs` and `Kmax` to instantiate an object of R6 class and pre-allocate memory for the future network ID matrix.

`makeNetInd.fromIDs(Net_str, IDs_str = NULL, sep = ' ')` Build the matrix of network IDs (`NetInd_k`) from IDs string vector, all friends of one observation `i` are located in a string Net_str[i], with two distinct friend IDs of `i` separated by character `sep`. If `IDs_str` is NULL it is assumed that the friends in Net_str are actual row numbers in `1:nobs`, otherwise IDs from Net_str will be used for looking up the observation row numbers in `IDs_str`.

`make.nF(NetInd_k = self$NetInd_k, nobs = self$nobs, Kmax = self$Kmax)` This method calculates the integer number of friends for each row of the network ID matrix (`self$NetInd_k`). The result is assigned to a field `self$nF` and is returned invisibly.

`mat.nF(nFnode)` `nFnode` - the character name for the number of friends variable that is assigned as a column name to a single column matrix in `self$nF`.

Methods

Public methods:

- `NetIndClass$new()`
- `NetIndClass$makeNetInd.fromIDs()`
- `NetIndClass$make.nF()`
- `NetIndClass$mat.nF()`
- `NetIndClass$clone()`

Method `new()`:

Usage:

```
NetIndClass$new(nobs, Kmax = 1)
```

Method `makeNetInd.fromIDs()`:

Usage:

```
NetIndClass$makeNetInd.fromIDs(Net_str, IDs_str = NULL, sep = " ")
```

Method `make.nF()`:

Usage:

```
NetIndClass$make.nF(
  NetInd_k = self$NetInd_k,
  nobs = self$nobs,
  Kmax = self$Kmax
)
```

Method `mat.nF()`**:**

Usage:

```
NetIndClass$mat.nF(nFnode)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
NetIndClass$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`network` *Define a Network Generator*

Description

Define a network generator by providing a function (using the argument `netfun`) which will simulate a network of connected friends for observations `i` in `1:n`. This network then serves as a backbone for defining and simulating from the structural equation models for dependent data. In particular, the network allows new nodes to be defined as functions of the previously simulated node values of `i`'s friends, across all observations `i`. Let `F_i` denote the set of friends of one observation `i` (observations in `F_i` are assumed to be "connected" to `i`) and refer to the union of these sets `F_i` as a "network" on `n` observations, denoted by `F`. A user-supplied network generating function `netfun` should be able to simulate such network `F` by returning a matrix of `n` rows, where each row `i` defines a friend set `F_i`, i.e., row `i` should be a vector of observations in `1:n` that are connected to `i` (friends of `i`), with the remainder filled by `NA`s. Each friend set `F_i` can contain up to `Kmax` unique indices `j` from `1:n`, except for `i` itself. `F_i` is also allowed to be empty (row `i` has only `NA`s), implying that `i` has no friends. The functionality is illustrated in the examples below. For additional information see Details. To learn how to use the `node` function for defining a node as a function of the friend node values, see Syntax and Network Summary Measures.

Usage

```
network(name, netfun, ..., params = list())
```

Arguments

`name`	Character string specifiying the name of the current network, may be used for adding new network that replaces the existing one (resample previous network)
`netfun`	Character name of the user-defined network generating function, can be any R function that returns a matrix of friend IDs of dimension `c(n, Kmax)`. The function must accept a named argument `n` that specifies the total sample size of the network. The matrix of network IDs should have `n` rows and `Kmax` columns, where each row `i` contains a vector of unique IDs in `1:n` that are `i`'s friends (observations that can influence `i`'s node distribution), except for `i` itself. Arguments to `netfun` can be either passed as named arguments to `network` function

itself or as a named list of parameters `params`. These network arguments can themselves be functions of the previously defined node names, allowing for network sampling itself to be dependent on the previously simulated node values, as shown in Example 2.

`...` Named arguments specifying distribution parameters that are accepted by the network sampling function in `netfun`. These parameters can be R expressions that are themselves formulas of the past node names.

`params` A list of additional named parameters to be passed on to the `netfun` function. The parameters have to be either constants or character strings of R expressions of the past node names.

Details

Without the network of friends, the `DAG` objects constructed by calling the `node` function can only specify structural equation models for independent and identically distributed data. That is, if no network is specified, for each observation `i` a node can be defined conditionally only on `i`'s own previously simulated node values. As a result, any two observations simulated under such data-generating model are always independent and identically distributed. Defining a network `F` allows one to define a new structural equation model where a node for each observation `i` can depend on its own simulated past, but also on the previously simulated node values of `i`'s friends (`F_i`). This is accomplished by allowing the data generating distribution for each observation `i`'s node to be defined conditionally on the past node values of `i`'s friends (observations in `F_i`). The network of friends can be used in subsequent calls to `node` function where new nodes (random variables) defined by the `node` function can depend on the node values of `i`'s friends (observations in the set `F_i`). During simulation it is assumed observations on `F_i` can simultaneously influence `i`.

Note that the current version of the package does not allow combining time-varying node indexing `Var[t]` and network node indexing `Var[[net_indx]]` for the same data generating distribution.

Each argument for the input network can be an evaluable R expression. All formulas are captured by delayed evaluation and are evaluated during the simulation. Formulas can refer to standard or user-specified R functions that must only apply to the values of previously defined nodes (i.e. node(s) that were called prior to `network()` function call).

Value

A list containing the network object(s) of type `DAG.net`, this will be utilized when data is simulated with `sim` function.

Syntax

The `network` function call that defines the network of friends can be added to a growing `DAG` object by using `'+'` syntax, much like a new `node` is added to a `DAG`. Subsequently defined nodes (`node` function calls) can employ the double square bracket subsetting syntax to reference previously simulated node values for specific friends in `F_i` simultaneously across all observations `i`. For example, `VarName[[net_indx]]` can be used inside the `node` formula to reference the node `VarName` values of `i`'s friends in `F_i[net_indx]`, simultaneously across all `i` in `1:n`.

The friend subsetting index `net_indx` can be any non-negative integer vector that takes values from 0 to `Kmax`, where 0 refers to the `VarName` node values of observation `i` itself (this is equivalent to just using `VarnName` in the `node` formula), `net_indx` value of 1 refers to node `VarName` values for

observations in F_i[1], across all i in 1:n (that is, the value of VarName of i's first friend F_i[1], if the friend exists and NA otherwise), and so on, up to net_indx value of Kmax, which would reference to the last friend node values of VarName, as defined by observations in F_i[Kmax] across all i. Note that net_indx can be a vector (e.g, net_indx=c(1:Kmax)), in which case the result of the query VarName[[c(1:Kmax)]] is a matrix of Kmax columns and n rows.

By default, VarName[[j]] evaluates to missing (NA) when observation i does not have a friend under F_i[j] (i.e., in the jth spot of i's friend set). This default behavior however can be changed to return 0 instead of NA, by passing an additional argument replaceNAw0 = TRUE to the corresponding node function.

Network Summary Measures

One can also define summary measures of the network covariates by specifying a node formula that applies an R function to the result of VarName[[net_indx]]. The rules for defining and applying such summary measures are identical to the rules for defining summary measures for time-varying nodes VarName[t_indx]. For example, use sum(VarName[[net_indx]]) to define a summary measure as a sum of VarName values of friends in F_i[net_indx], across all observations i in 1:n. Similarly, use mean(VarName[[net_indx]]) to define a summary measure as a mean of VarName values of friends in F_i[net_indx], across all i. For more details on defining such summary functions see the simcausal vignette.

See Also

igraph.to.sparseAdjMat; sparseAdjMat.to.NetInd; NetInd.to.sparseAdjMat; sparseAdjMat.to.igraph

Examples

```
#----------------------------------------------------------------------------------------------------
# EXAMPLE 1. USING igraph R PACKAGE TO SIMULATE NETWORKS
#----------------------------------------------------------------------------------------------------

#----------------------------------------------------------------------------------------------------
# Example of a network sampler, will be provided as "netfun" argument to network(, netfun=);
# Generates a random graph according to the G(n,m) Erdos-Renyi model using the igraph package;
# Returns (n,Kmax) matrix of net IDs (friends) by row;
# Row i contains the IDs (row numbers) of i's friends;
# i's friends are assumed connected to i and can influence i in equations defined by node())
# When i has less than Kmax friends, the remaining i row entries are filled with NAs;
# Argument m_pn: > 0
# a total number of edges in the network as a fraction (or multiplier) of n (sample size)
#----------------------------------------------------------------------------------------------------
gen.ER <- function(n, m_pn, ...) {
  m <- as.integer(m_pn*n)
  if (n<=10) m <- 20
  igraph.ER <- igraph::sample_gnm(n = n, m = m, directed = TRUE)
  sparse_AdjMat <- igraph.to.sparseAdjMat(igraph.ER)
  NetInd_out <- sparseAdjMat.to.NetInd(sparse_AdjMat)
  return(NetInd_out$NetInd_k)
}

D <- DAG.empty()
```

```
# Sample ER model network using igraph::sample_gnm with m_pn argument:
D <- D + network("ER.net", netfun = "gen.ER", m_pn = 50)
# W1 - categorical (6 categories, 1-6):
D <- D +
  node("W1", distr = "rcat.b1",
        probs = c(0.0494, 0.1823, 0.2806, 0.2680, 0.1651, 0.0546)) +
# W2 - binary infection status, positively correlated with W1:
  node("W2", distr = "rbern", prob = plogis(-0.2 + W1/3)) +
# W3 - binary confounder:
  node("W3", distr = "rbern", prob = 0.6)
# A[i] is a function W1[i] and the total of i's friends values W1, W2 and W3:
D <- D + node("A", distr = "rbern",
              prob = plogis(2 + -0.5 * W1 +
                            -0.1 * sum(W1[[1:Kmax]]) +
                            -0.4 * sum(W2[[1:Kmax]]) +
                            -0.7 * sum(W3[[1:Kmax]])),
              replaceNAw0 = TRUE)
# Y[i] is a function of netW3 (friends of i W3 values) and the total N of i's friends
# who are infected AND untreated:
D <- D + node("Y", distr = "rbern",
              prob = plogis(-1 + 2 * sum(W2[[1:Kmax]] * (1 - A[[1:Kmax]])) +
                            -2 * sum(W3[[1:Kmax]])
                            ),
              replaceNAw0 = TRUE)
# Can add N untreated friends to the above outcome Y equation: sum(1 - A[[1:Kmax]]):
D <- D + node("Y", distr = "rbern",
              prob = plogis(-1 + 1.5 * sum(W2[[1:Kmax]] * (1 - A[[1:Kmax]])) +
                            -2 * sum(W3[[1:Kmax]]) +
                            0.25 * sum(1 - A[[1:Kmax]])
                            ),
              replaceNAw0 = TRUE)
# Can add N infected friends at baseline to the above outcome Y equation: sum(W2[[1:Kmax]]):
D <- D + node("Y", distr = "rbern",
              prob = plogis(-1 + 1 * sum(W2[[1:Kmax]] * (1 - A[[1:Kmax]])) +
                            -2 * sum(W3[[1:Kmax]]) +
                            0.25 * sum(1 - A[[1:Kmax]]) +
                            0.25 * sum(W2[[1:Kmax]])
                            ),
              replaceNAw0 = TRUE)
Dset <- set.DAG(D, n.test = 100)
# Simulating data from the above sem:
datnet <- sim(Dset, n = 1000, rndseed = 543)
head(datnet)
# Obtaining the network object from simulated data:
net_object <- attributes(datnet)$netind_cl
# Max number of friends:
net_object$Kmax
# Network matrix
head(attributes(datnet)$netind_cl$NetInd)

#----------------------------------------------------------------------------------------------------
# EXAMPLE 2. USING CUSTOM NETWORK GENERATING FUNCTION
#----------------------------------------------------------------------------------------------------
```

```
#-----------------------------------------------------------------------------------------------------
# Example of a user-defined network sampler(s) function
# Arguments K, bslVar[i] (W1) & nF are evaluated in the environment of the simulated data then
# passed to genNET() function
  # - K: maximum number of friends for any unit
  # - bslVar[i]: used for contructing weights for the probability of selecting i as
  # someone else's friend (weighted sampling), when missing the sampling goes to uniform
  # - nF[i]: total number of friends that need to be sampled for observation i
#-----------------------------------------------------------------------------------------------------
genNET <- function(n, K, bslVar, nF, ...) {
  prob_F <- plogis(-4.5 + 2.5*c(1:K)/2) / sum(plogis(-4.5 + 2.5*c(1:K)/2))
  NetInd_k <- matrix(NA_integer_, nrow = n, ncol = K)
  nFriendTot <- rep(0L, n)
  for (index in (1:n)) {
    FriendSampSet <- setdiff(c(1:n), index)
    nFriendSamp <- max(nF[index] - nFriendTot[index], 0L)
    if (nFriendSamp > 0) {
      if (length(FriendSampSet) == 1)  {
        friends_i <- FriendSampSet
      } else {
        friends_i <- sort(sample(FriendSampSet, size = nFriendSamp,
                          prob = prob_F[bslVar[FriendSampSet] + 1]))
      }
      NetInd_k[index, ] <- c(as.integer(friends_i),
                             rep_len(NA_integer_, K - length(friends_i)))
      nFriendTot[index] <- nFriendTot[index] + nFriendSamp
    }
  }
  return(NetInd_k)
}

D <- DAG.empty()
D <- D +
# W1 - categorical or continuous confounder (5 categories, 0-4):
  node("W1", distr = "rcat.b0",
        probs = c(0.0494, 0.1823, 0.2806, 0.2680, 0.1651, 0.0546)) +
# W2 - binary infection status at t=0, positively correlated with W1:
  node("W2", distr = "rbern", prob = plogis(-0.2 + W1/3)) +
# W3 - binary confounder:
  node("W3", distr = "rbern", prob = 0.6)

# def.nF: total number of friends for each i (0-K), each def.nF[i] is influenced by categorical W1
K <- 10
set.seed(12345)
normprob <- function(x) x / sum(x)
p_nF_W1_mat <- apply(matrix(runif((K+1)*6), ncol = 6, nrow = (K+1)), 2, normprob)
colnames(p_nF_W1_mat) <- paste0("p_nF_W1_", c(0:5))
create_probs_nF <- function(W1) t(p_nF_W1_mat[,W1+1])
vecfun.add("create_probs_nF")
D <- D + node("def.nF", distr = "rcat.b0", probs = create_probs_nF(W1))

# Adding the network generator that depends on nF and categorical W1:
```

```
D <- D + network(name="net.custom", netfun = "genNET", K = K, bslVar = W1, nF = def.nF)
# Define A[i] is a function W1[i] as well as the total sum of i's friends values for W1, W2 and W3:
D <- D + node("A", distr = "rbern",
              prob = plogis(2 + -0.5 * W1 +
                            -0.1 * sum(W1[[1:Kmax]]) +
                            -0.4 * sum(W2[[1:Kmax]]) +
                            -0.7 * sum(W3[[1:Kmax]])),
              replaceNAw0 = TRUE)
# Y[i] is a the total N of i's friends who are infected AND untreated
# + a function of friends W3 values
D <- D + node("pYRisk", distr = "rconst",
              const = plogis(-1 + 2 * sum(W2[[1:Kmax]] * (1 - A[[1:Kmax]])) +
                               -1.5 * sum(W3[[1:Kmax]])),
              replaceNAw0 = TRUE)

D <- D + node("Y", distr = "rbern", prob = pYRisk)
Dset <- set.DAG(D, n.test = 100)

# Simulating data from the above sem:
datnet <- sim(Dset, n = 1000, rndseed = 543)
head(datnet, 10)
# Obtaining the network object from simulated data:
net_object <- attributes(datnet)$netind_cl
# Max number of friends:
net_object$Kmax
# Network matrix
head(attributes(datnet)$netind_cl$NetInd)
plotDAG(Dset)
```

---

node *Create Node Object(s)*

Description

Define a single DAG node and its distribution or define many repeated-measure/time-varying nodes by using argument `t`. The node distribution is allowed to vary as a function of time (`t`). Conditionaing on past nodes is accomplished by using the syntactic sugar, such as, `NodeName[t]`. After all the nodes have been added to the DAG, call `set.DAG`, a DAG object constructor, and `add.action`, an action (intervention) constructor.

Usage

```
node(name, t, distr, EFU, order, ..., params = list(), asis.params = list())
```

Arguments

`name`	Character node name or a vector of names when specifying a multivariate node. For time-dependent nodes the names will be automatically expanded to a scheme "name_t" for each t provided specified.

t	Node time-point(s). Allows specification of several time-points when t is a vector of positive integers, in which case the output will consist of a named list of length(t) nodes, corresponding to each value in t.
distr	Character name of the node distribution, can be a standard distribution R function, s.a. rnorm, rbinom, runif or user defined. The function must accept a named argument "n" to specify the total sample size. Distributional parameters (arguments) must be passed as either named arguments to node or as a named list of parameters "params".
EFU	End-of-Follow Up flag for designating a survival/censoring type node, only applies to Bernoulli nodes. When EFU=TRUE this node becomes an indicator for the end of follow-up event (censoring, end of study, death, etc). When simulated variable with this node distribution evaluates to value 1 subsequent nodes with higher temporal order values will be set to NA by default (or imputed with carry forward imputation, depending on the settings of the sim function). This can only be set to TRUE and should be omitted otherwise.
order	An optional integer parameter specifying the order in which these nodes will be sampled. The value of order has to start at 1 and be unique for each new node, can be specified as a range / vector and has to be of the same length as the argument t above. When order is left unspecified it will be automatically inferred based on the order in which the node(s) were added in relation to other nodes. See Examples and Details below.
...	Named arguments specifying distribution parameters that are accepted by the distr function. The parameters can be R expressions that are themselves formulas of the past node names.
params	A list of additional named parameters to be passed on to the distr function. The parameters have to be either constants or character strings of R expressions of the past node names.
asis.params	(ADVANCED USE) A list of additional named distributional parameters that will be evaluated "as is", inside the currently simulated data.frame + the calling environment, without any modifications to the R expression strings inside the asis.params list. There is no error-checking for existing node names and no parent node name extraction (the arrows from parents will not appear in plotDAG). Time varying nodes should be referenced by their names as they appear in the simulated data, as in "TVar_t". See details and examples 7 and 8 below.

Value

A list containing node object(s) (expanded to several nodes if t is an integer vector of length > 1)

Details

The combination of name and t must uniquely identify each node in the DAG. Use argument t to identify measurements of the same attribute (e.g. 'A1c') at various time points. The collection of all unique t values, across all nodes, should consist of non-negative integers (i.e., starting at 0).

The optional order argument can be specified, used for determining the sampling order of each node. When order not specified, it is automatically inferred based on the actual order in which the nodes were added to the DAG (earlier added nodes get lower order value and are sampled first)

All node calls that share the same generic name name must also share the same EFU value (if any is specified in at least one of them). A value of TRUE for the EFU indicates that if a simulated value for a measurement of the attribute represented by node is 1 then all the following nodes with that measurement (in terms of higher t values) in the DAG will be unobserved (i.e., their simulated value will be set to NA).

Node formulas (parameters of the distribution)

Each user-supplied argument to the node function is an evaluable R expression, their evaluation is delayed until the actual simulation time. These arguments can refer to standard or user-specified R functions that must only apply to the values of parent nodes, i.e. a subset of the node(s) with an order value strictly lower than that of the node characterized by the formula. Formulas must reference the parent nodes with unique name identifiers, employing the square bracket vector subsetting name[t] for referencing a parent node at a particular time point t (if any time-points were specified). The square bracket notation is used to index a generic name with the relevant time point as illustrated in the examples. When an input node is used to define several nodes (i.e., several measurement of the same attribute, t=0:5), the formula(s) specified in that node can apply to each node indexed by a given time point denoted by t. This generic expression t can then be referenced within a formula to simultaneously identify a different set of parent nodes for each time point as illustrated below. Note that the parents of each node represented by a given node object are implicitly defined by the nodes referenced in formulas of that node call.

Different types of evaluation for node function arguments

There is quite a bit of flexibility in the way in which the node function arguments can be evaluated. By default, the named arguments specified as expressions are first captured by delayed-evaluation and then modified by simcausal to enable the special types of functional syntax. For example, simcausal will over-ride the subsetting operators '[...]' (for time varying nodes) and '[[...]]' (for networks), implying that these operators can no longer be used in their typical R way. Furthermore, simcausal will over-ride the standard 'c' function, with its own definition. Similarly, it will over-ride any calls to sum and mean functions with their row-matrix counterpart functions rowSums and rowMeans. When programming with simcausal (such as passing node arguments inside a function, prior to defining the node), it may be helpful to instead pass such node arguments as character strings, rather than as R expressions. In this case one should use the argument params by adopting the following syntax node(...,params = list(mean="A+B")), which in this case is equivalent to: node(..., mean = A+B).

There are also instances when it might be desirable to retain the original behavior of all R expressions and functions and evaluate a particular node argument "as is". For example, the user may wish to retain the original R functionality of all its operators, including those of [...] and [[...]]. In this case the node argument (or a specific part of the node argument) should be wrapped in .() or eval(). Note that once the expression has been wrapped with .(...) (or eval(...)), the simcausal definitions of operators [...] and [[...]] no longer apply to these expressions and no error checking for "correctness" of these node arguments will be performed.

The forced-evaluation operator .() can be also used as part of an expression, which will prevent the typical simcausal evaluation on only that specific part of the expression. Example 8 below demonstrates the following use case for the expression .(coefAi[t]) * A[t-1], which will look for vector coefAi and then subset it by current value of t (and return a scalar), while A[t-1] will evaluate to the entire column vector of variable A for time point t-1. Such an expression will

multiply the entire time-varying vector `A[t-1]` by scalar value determined by current value of `t` and the previously defined vector `coefAi`.

Furthermore, even when a vector or a matrix is wrapped in .(...) it still will be automatically re-parsed into K column matrix with n rows. When this is not desired, for example, when defining a multivariate node distribution, the user may pass such vector or matrix node arguments as a character string in a list argument `asis.params`. See Example 7 and 8 below for additional details.

Multivariate random variables (multivariate nodes)

Starting from v.0.5, a single `node` call can be used for defining a multivariate (and possibly correlated) random vector. To define a random vector that has more than 1 dimension, use the argument `name` to specify a vector with names for each dimension, e.g.,

`node(c("X1","X2"), distr = "mvtnorm::rmvnorm", asis.params = list(mean = "c(0,1)", sigma = "matrix(c(1,0.75,0.75,1), ncol=2)"))`

will define a bi-variate (correlated) normally distributed node, the simulated data set will contain this bi-variately distributed random variable in columns "X1" and "X2". Note that the multivariate sampling distribution function (such as function `rmvnorm` from the package `mvtnorm`) must return a matrix of `n` rows (number of observations) and `length(name)` columns (dimensionality). See additional examples below.

Note that one can also define time-varying multivariate nodes, e.g.,

`node(c("X1","X2"), t=0:5, distr = "mvtnorm::rmvnorm", asis.params = list(mean = "c(0,1)"))`.

References

Sofrygin O, van der Laan MJ, Neugebauer R (2017). "simcausal R Package: Conducting Transparent and Reproducible Simulation Studies of Causal Effect Estimation with Complex Longitudinal Data." Journal of Statistical Software, 81(2), 1-47. doi: 10.18637/jss.v081.i02.

Examples

```
#----------------------------------------------------------------------------------------
# EXAMPLE 1A: Define some Bernoulli nodes, survival outcome Y and put it together in a
# DAG object
#----------------------------------------------------------------------------------------
W1node <- node(name = "W1", distr = "rbern",
prob = plogis(-0.5), order = 1)
W2node <- node(name = "W2", distr = "rbern",
prob = plogis(-0.5 + 0.5 * W1), order = 2)
Anode <- node(name = "A", distr = "rbern",
prob = plogis(-0.5 - 0.3 * W1 - 0.3 * W2), order = 3)
Ynode <- node(name = "Y", distr = "rbern",
prob = plogis(-0.1 + 1.2 * A + 0.3 * W1 + 0.3 * W2), order = 4)
D1Aset <- set.DAG(c(W1node,W2node,Anode,Ynode))

#----------------------------------------------------------------------------------------
# EXAMPLE 1B: Same as 1A using +node interface and no order argument
#----------------------------------------------------------------------------------------
D1B <- DAG.empty()
D1B <- D1B +
```

```
  node(name = "W1", distr = "rbern", prob = plogis(-0.5)) +
  node(name = "W2", distr = "rbern", prob = plogis(-0.5 + 0.5 * W1)) +
  node(name = "A", distr = "rbern", prob = plogis(-0.5 - 0.3 * W1 - 0.3 * W2)) +
  node(name = "Y", distr = "rbern", prob = plogis(-0.1 + 1.2 * A + 0.3 * W1 + 0.3 * W2))
D1Bset <- set.DAG(D1B)

#----------------------------------------------------------------------------------------
# EXAMPLE 1C: Same as 1A and 1B using add.nodes interface and no order argument
#----------------------------------------------------------------------------------------
D1C <- DAG.empty()
D1C <- add.nodes(D1C, node(name = "W1", distr = "rbern", prob = plogis(-0.5)))
D1C <- add.nodes(D1C, node(name = "W2", distr = "rbern",
prob = plogis(-0.5 + 0.5 * W1)))
D1C <- add.nodes(D1C, node(name = "A", distr = "rbern",
prob = plogis(-0.5 - 0.3 * W1 - 0.3 * W2)))
D1C <- add.nodes(D1C, node(name = "Y", distr = "rbern",
prob = plogis(-0.1 + 1.2 * A + 0.3 * W1 + 0.3 * W2)))
D1C <- set.DAG(D1C)

#----------------------------------------------------------------------------------------
# EXAMPLE 1D: Add a uniformly distributed node and redefine outcome Y as categorical
#----------------------------------------------------------------------------------------
D_unif <- DAG.empty()
D_unif <- D_unif +
 node("W1", distr = "rbern", prob = plogis(-0.5)) +
 node("W2", distr = "rbern", prob = plogis(-0.5 + 0.5 * W1)) +
 node("W3", distr = "runif", min = plogis(-0.5 + 0.7 * W1 + 0.3 * W2), max = 10) +
 node("An", distr = "rbern", prob = plogis(-0.5 - 0.3 * W1 - 0.3 * W2 - 0.2 * sin(W3)))
# Categorical syntax 1 (probabilities as values):
D_cat_1 <- D_unif + node("Y", distr = "rcat.b1", probs = {0.3; 0.4})
D_cat_1 <- set.DAG(D_cat_1)
# Categorical syntax 2 (probabilities as formulas):
D_cat_2 <- D_unif +
node("Y", distr = "rcat.b1",
probs={plogis(-0.1 + 1.2 * An + 0.3 * W1 + 0.3 * W2 + 0.2 * cos(W3));
   plogis(-0.5 + 0.7 * W1)})
D_cat_2 <- set.DAG(D_cat_2)

#----------------------------------------------------------------------------------------
# EXAMPLE 2A: Define Bernoulli nodes using R rbinom() function, defining prob argument
# for L2 as a function of node L1
#----------------------------------------------------------------------------------------
D <- DAG.empty()
D <- D +
node("L1", t = 0, distr = "rbinom", prob = 0.05, size = 1) +
node("L2", t = 0, distr = "rbinom", prob = ifelse(L1[0] == 1, 0.5, 0.1), size = 1)
Dset <- set.DAG(D)

#----------------------------------------------------------------------------------------
# EXAMPLE 2B: Equivalent to 2A, passing argument size to rbinom inside a named list
# params
#----------------------------------------------------------------------------------------
D <- DAG.empty()
```

```
D <- D +
node("L1", t = 0, distr = "rbinom", prob = 0.05, params = list(size = 1)) +
node("L2", t = 0, distr = "rbinom",
prob = ifelse(L1[0] == 1,0.5,0.1), params = list(size = 1))
Dset <- set.DAG(D)

#----------------------------------------------------------------------------------------
# EXAMPLE 2C: Equivalent to 2A and 2B, define Bernoulli nodes using a wrapper "rbern"
#----------------------------------------------------------------------------------------
D <- DAG.empty()
D <- D +
node("L1", t = 0, distr = "rbern", prob = 0.05) +
node("L2", t = 0, distr = "rbern", prob = ifelse(L1[0] == 1, 0.5, 0.1))
Dset <- set.DAG(D)

#----------------------------------------------------------------------------------------
# EXAMPLE 3: Define node with normal distribution using rnorm() R function
#----------------------------------------------------------------------------------------
D <- DAG.empty()
D <- D + node("L2", t = 0, distr = "rnorm", mean = 10, sd = 5)
Dset <- set.DAG(D)

#----------------------------------------------------------------------------------------
# EXAMPLE 4: Define 34 Bernoulli nodes, or 2 Bernoulli nodes over 17 time points,
#----------------------------------------------------------------------------------------
t_end <- 16
D <- DAG.empty()
D <- D +
node("L2", t = 0:t_end, distr = "rbinom",
prob = ifelse(t == t_end, 0.5, 0.1), size = 1) +
node("L1", t = 0:t_end, distr = "rbinom",
prob = ifelse(L2[0] == 1, 0.5, 0.1), size = 1)
Dset <- set.DAG(D)
sim(Dset, n=10)

#----------------------------------------------------------------------------------------
# EXAMPLE 5: Defining new distribution function 'rbern', defining and passing a custom
# vectorized node function 'customfun'
#----------------------------------------------------------------------------------------
rbern <- function(n, prob) { # defining a bernoulli wrapper based on R rbinom function
  rbinom(n = n, prob = prob, size = 1)
}
customfun <- function(arg, lambda) {
  res <- ifelse(arg == 1, lambda, 0.1)
  res
}
D <- DAG.empty()
D <- D +
node("W1", distr = "rbern", prob = 0.05) +
node("W2", distr = "rbern", prob = customfun(W1, 0.5)) +
node("W3", distr = "rbern", prob = ifelse(W1 == 1, 0.5, 0.1))
D1d <- set.DAG(D, vecfun = c("customfun"))
sim(D1d, n = 10, rndseed = 1)
```

```
#----------------------------------------------------------------------------------------
# EXAMPLE 6: Defining latent variables I and U.Y (will be hidden from simulated data)
#----------------------------------------------------------------------------------------
D <- DAG.empty()
D <- D +
  node("I",
    distr = "rcat.b1",
    probs = c(0.1, 0.2, 0.2, 0.2, 0.1, 0.1, 0.1)) +
  node("W1",
    distr = "rnorm",
    mean = ifelse(I == 1, 0, ifelse(I == 2, 3, 10)) + 0.6 * I,
    sd = 1) +
  node("W2",
    distr = "runif",
    min = 0.025*I, max = 0.7*I) +
  node("W3",
    distr = "rbern",
    prob = plogis(-0.5 + 0.7*W1 + 0.3*W2 - 0.2*I)) +
  node("A",
    distr = "rbern",
    prob = plogis(+4.2 - 0.5*W1 + 0.2*W2/2 + 0.2*W3)) +
node("U.Y", distr = "rnorm", mean = 0, sd = 1) +
  node("Y",
    distr = "rconst",
    const = -0.5 + 1.2*A + 0.1*W1 + 0.3*W2 + 0.2*W3 + 0.2*I + U.Y)
Dset1 <- set.DAG(D, latent.v = c("I", "U.Y"))
sim(Dset1, n = 10, rndseed = 1)

#----------------------------------------------------------------------------------------
# EXAMPLE 7: Multivariate random variables
#----------------------------------------------------------------------------------------
if (requireNamespace("mvtnorm", quietly = TRUE)) {
  D <- DAG.empty()
  # 2 dimensional normal (uncorrelated), using rmvnorm function from rmvnorm package:
  D <- D +
    node(c("X1","X2"), distr = "mvtnorm::rmvnorm",
      asis.params = list(mean = "c(0,1)")) +
  # Can define a degenerate (rconst) multivariate node:
    node(c("X1.copy", "X2.copy"), distr = "rconst", const = c(X1, X2))
  Dset1 <- set.DAG(D, verbose = TRUE)
  sim(Dset1, n = 10)
}

# On the other hand this syntax wont work,
# since simcausal will parse c(0,1) into a two column matrix:
## Not run:
D <- DAG.empty()
D <- D + node(c("X1","X2"), distr = "mvtnorm::rmvnorm", mean = c(0,1))
Dset1 <- set.DAG(D, verbose = TRUE)

## End(Not run)
```

```
if (requireNamespace("mvtnorm", quietly = TRUE)) {
  D <- DAG.empty()
  # Bivariate normal (correlation coef 0.75):
  D <- D +
    node(c("X1","X2"), distr = "mvtnorm::rmvnorm",
      asis.params = list(mean = "c(0,1)",
        sigma = "matrix(c(1,0.75,0.75,1), ncol=2)")) +
  # Can use any component of such multivariate nodes when defining new nodes:
    node("A", distr = "rconst", const = 1 - X1)
  Dset2 <- set.DAG(D, verbose = TRUE)
  sim(Dset2, n = 10)
}

# Time-varying multivar node (3 time-points, 2 dimensional normal)
# plus changing the mean over time, as as function of t:
if (requireNamespace("mvtnorm", quietly = TRUE)) {
  D <- DAG.empty()
  D <- D +
    node(c("X1", "X2"), t = 0:2, distr = "mvtnorm::rmvnorm",
      asis.params = list(
        mean = "c(0,1) + t",
        sigma = "matrix(rep(0.75,4), ncol=2)"))
  Dset5b <- set.DAG(D)
  sim(Dset5b, n = 10)
}

# Two ways to define the same bivariate uniform copula:
if (requireNamespace("copula", quietly = TRUE)) {
  D <- DAG.empty()
  D <- D +
  # with a warning since normalCopula() returns an object unknown to simcausal:
    node(c("X1","X2"), distr = "copula::rCopula",
      copula = eval(copula::normalCopula(0.75, dim = 2))) +
  # same, as above:
    node(c("X3","X4"), distr = "copula::rCopula",
      asis.params = list(copula = "copula::normalCopula(0.75, dim = 2)"))
  vecfun.add("qbinom")
  # Bivariate binomial derived from previous copula, with same correlation:
  D <- D +
    node(c("A.Bin1", "A.Bin2"), distr = "rconst",
      const = c(qbinom(X1, 10, 0.5), qbinom(X2, 15, 0.7)))
  Dset3 <- set.DAG(D)
  sim(Dset3, n = 10)
}

# Same as "A.Bin1" and "A.Bin2", but directly using rmvbin function in bindata package:
if (requireNamespace("bindata", quietly = TRUE)) {
  D <- DAG.empty()
  D <- D +
    node(c("B.Bin1","B.Bin2"), distr = "bindata::rmvbin",
      asis.params = list(
        margprob = "c(0.5, 0.5)",
        bincorr = "matrix(c(1,0.75,0.75,1), ncol=2)"))
```

```
  Dset4b <- set.DAG(D)
  sim(Dset4b, n = 10)
}

#----------------------------------------------------------------------------------------
# EXAMPLE 8: Combining simcausal non-standard evaluation with eval() forced evaluation
#----------------------------------------------------------------------------------------
coefAi <- 1:10
D <- DAG.empty()
D <- D +
  node("A", t = 1, distr = "rbern", prob = 0.7) +
  node("A", t = 2:10, distr = "rconst", const = eval(coefAi[t]) * A[t-1])
Dset8 <- set.DAG(D)
sim(Dset8, n = 10)

#----------------------------------------------------------------------------------------
# TWO equivalent ways of creating a multivariate node (combining nodes W1 and W2):
#----------------------------------------------------------------------------------------
D <- DAG.empty()
D <- D + node("W1", distr = "rbern", prob = 0.45)
D <- D + node("W2", distr = "rconst", const = 1)

# option 1:
D <- D + node(c("W1.copy1", "W2.copy1"), distr = "rconst", const = c(W1, W2))

# equivalent option 2:
create_mat <- function(W1, W2) cbind(W1, W2)
vecfun.add("create_mat")
D <- D + node(c("W1.copy2", "W2.copy2"), distr = "rconst", const = create_mat(W1, W2))

Dset <- set.DAG(D)
sim(Dset, n=10, rndseed=1)
```

`parents` *Show Node Parents Given DAG Object*

Description

Given a vector of node names, this function provides the name(s) of node parents that were obtained by parsing the node formulas.

Usage

```
parents(DAG, nodesChr)
```

Arguments

`DAG`	A DAG object that was specified by calling `set.DAG`
`nodesChr`	A vector of node names that are already defined in DAG

Value

A list with parent names for each node name in nodesChr

Examples

```
D <- DAG.empty()
D <- D + node(name="W1", distr="rbern", prob=plogis(-0.5))
D <- D + node(name="W2", distr="rbern", prob=plogis(-0.5 + 0.5*W1))
D <- D + node(name="A", distr="rbern", prob=plogis(-0.5 - 0.3*W1 - 0.3*W2))
D <- D + node(name="Y", distr="rbern", prob=plogis(-0.1 + 1.2*A + 0.3*W1 + 0.3*W2), EFU=TRUE)
D <- set.DAG(D)
parents(D, c("W2", "A", "Y"))
```

---

plotDAG *Plot DAG*

Description

Plot DAG object using functions from `igraph` package. The default setting is to keep the regular (observed) DAG nodes with shape set to "none", which can be over-ridden by the user. For latent (hidden) DAG nodes the default is to: 1) set the node color as grey; 2) enclose the node by a circle; and 3) all directed edges coming out of the latent node are plotted as dashed.

Usage

```
plotDAG(
  DAG,
  tmax = NULL,
  xjitter,
  yjitter,
  node.action.color,
  vertex_attrs = list(),
  edge_attrs = list(),
  excludeattrs,
  customvlabs,
  verbose = getOption("simcausal.verbose")
)
```

Arguments

`DAG`	A DAG object that was specified by calling `set.DAG`
`tmax`	Maximum time-point to plot for time-varying DAG objects
`xjitter`	Amount of random jitter for node x-axis plotting coordinates
`yjitter`	Amount of random jitter for node y-axis plotting coordinates
`node.action.color`	Color of the action node labels (only for action DAG of class DAG.action). If missing, defaults to red.

vertex_attrs	A named list of igraph graphical parameters for plotting DAG vertices. These parameters are passed on to add.vertices igraph function.
edge_attrs	A named list of igraph graphical parameters for plotting DAG edges. These parameters are passed on to add.edges igraph function.
excludeattrs	A character vector for DAG nodes that should be excluded from the plot
customvlabs	A named vector of custom DAG node labels (replaces node names from the DAG object).
verbose	Set to TRUE to print messages on status and information to the console. Turn this off by default using options(simcausal.verbose=FALSE).

References

Sofrygin O, van der Laan MJ, Neugebauer R (2017). "simcausal R Package: Conducting Transparent and Reproducible Simulation Studies of Causal Effect Estimation with Complex Longitudinal Data." Journal of Statistical Software, 81(2), 1-47. doi: 10.18637/jss.v081.i02.

---

plotSurvEst *(EXPERIMENTAL) Plot Discrete Survival Function(s)*

Description

Plot discrete survival curves from a list of discrete survival probabilities by calling plot with type='b'.

Usage

```
plotSurvEst(
  surv = list(),
  xindx = NULL,
  ylab = "",
  xlab = "t",
  ylim = c(0, 1),
  legend.xyloc = "topright",
  ...
)
```

Arguments

surv	A list of vectors, each containing action-specific discrete survival probabilities over time.
xindx	A vector of indices for subsetting the survival vectors in surv, if omitted all survival probabilities in each surv[[i]] are plotted.
ylab	An optional title for y axis, passed to plot.
xlab	An optional title for x axis, passed to plot.
ylim	Optional y limits for the plot, passed to plot.

legend.xyloc	Optional x and y co-ordinates to be used to position the legend. Can be specified by keyword or as a named list with (x,y), uses the same convention as in graphics::xy.coords.
...	Additional arguments passed to plot.

---

print.DAG *Print DAG Object*

Description

Print DAG Object

Usage

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

Arguments

x	A DAG object.
...	Other arguments to generic print.

---

print.DAG.action *Print Action Object*

Description

Print Action Object

Usage

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

Arguments

x	An object.
...	Other arguments to generic print.

print.DAG.node *Print DAG.node Object*

Description

Print DAG.node Object

Usage

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

Arguments

x	A Node object.
...	Other arguments to generic print.

rbern *Random Sample from Bernoulli Distribution*

Description

Wrapper for Bernoulli node distribution.

Usage

```
rbern(n, prob)
```

Arguments

n	Sample size.
prob	A vector of success probabilities.

Value

Binary vector of length n.

Examples

```
#----------------------------------------------------------------------------------------
# Specifying and simulating from a DAG with 3 Bernoulli nodes
#----------------------------------------------------------------------------------------
D <- DAG.empty()
D <- D + node("W1", distr="rbern", prob=0.05)
D <- D + node("W2", distr="rbern", prob=ifelse(W1==1,0.5,0.1))
D <- D + node("W3", distr="rbern", prob=ifelse(W1==1,0.5,0.1))
Dset <- set.DAG(D)
simdat <- sim(Dset, n=200, rndseed=1)
```

`rcat.factor` *Random Sample for a Categorical Factor*

Description

Matrix version of the categorical distribution. The argument `probs` can be a matrix of `n` rows, specifying individual (varying in sample) categorical probabilities. The number of categories generated is equal to `ncol(probs)+1`, the levels labeled as: `1,...,ncol(probs)+1`.

Usage

```
rcat.factor(n, probs)

rcategor(n, probs)
```

Arguments

`n`	Sample size.
`probs`	Either a vector or a matrix of success probabilities. When `probs` is a vector, `n` identically distributed random categorical variables are generated with categories: 1, 2, ..., length(probs)+1. When `probs` is a matrix, the categorical probabilities of the kth sample are determined by the kth row of `probs` matrix, i.e., `probs[k,]`.

Value

A factor of length `n` with levels: `1,2, ...,ncol(probs)+1`.

Functions

- `rcategor()`: (Deperecated) Random Sample of a Categorical Factor

See Also

`rcat.b1`, `rcat.b0`

Examples

```
#---------------------------------------------------------------------------------------
# Specifying and simulating from a DAG with one categorical node with constant
# probabilities
#---------------------------------------------------------------------------------------
D <- DAG.empty()
D <- D + node("race",t=0,distr="rcat.factor",probs=c(0.2,0.1,0.4,0.15,0.05,0.1))
Dset <- set.DAG(D)
simdat <- sim(Dset, n=200, rndseed=1)

#---------------------------------------------------------------------------------------
```

```
# Specifying and simulating from a DAG with a categorical node with varying
# probabilities (probabilities are determined by values sampled for nodes L0 and L1)
#-----------------------------------------------------------------------------------
D <- DAG.empty()
D <- D + node("L0", distr="rnorm", mean=10, sd=5)
D <- D + node("L1", distr="rnorm", mean=10, sd=5)
D <- D + node("L2", distr="rcat.factor", probs=c(abs(1/L0), abs(1/L1)))
Dset <- set.DAG(D)
simdat <- sim(Dset, n=200, rndseed=1)
```

`rcategor.int` *Random Sample from Base 1 (rcat.b1) or Base 0 (rcat.b0) Categorical (Integer) Distribution*

Description

Same as , but returning a vector of sampled integers with range 1, 2, ..., `ncol(probs)+1` for `rcat.b1` or range 0, 1, ..., `ncol(probs)` for `rcat.b0`. For sampling categorical factors see rcat.factor.

Usage

```
rcategor.int(n, probs)

rcat.b1(n, probs)

rcat.b0(n, probs)
```

Arguments

`n`	Sample size.
`probs`	Either a vector or a matrix of success probabilities. When probs is a vector, `n` identically distributed random categorical variables are generated. When `probs` is a matrix, the categorical probabilities of the `k`th sample are determined by the `k`th row of probs matrix, i.e., `probs[k,]`.

Value

An integer vector of length `n` with range either in `0,...,ncol(probs)` or in `1,...,ncol(probs)+1`.

Functions

- `rcategor.int()`: (Deperecated) Random Sample from Base 1 Categorical (Integer) Distribution
- `rcat.b1()`: Random Sample from Base 1 Categorical (Integer) Distribution
- `rcat.b0()`: Random Sample from Base 0 Categorical (Integer) Distribution

See Also

`rcat.factor`

rconst *Constant (Degenerate) Distribution (Returns its Own Argument* const*)*

Description

Wrapper for constant value (degenerate) distribution.

Usage

```
rconst(n, const)
```

Arguments

n	Sample size.
const	Either a vector with one constant value (replicated n times) or a vector of length n or a matrix with n rows (for a multivariate node).

Value

A vector of constants of length n.

Examples

```
#----------------------------------------------------------------------------------
# Specifying and simulating from a DAG with 1 Bernoulli and 2 constant nodes
#----------------------------------------------------------------------------------
D <- DAG.empty()
D <- D + node("W1", distr = "rbern", prob = 0.45)
D <- D + node("W2", distr = "rconst", const = 1)
D <- D + node("W3", distr = "rconst", const = ifelse(W1 == 1, 5, 10))

# TWO equivalent ways of creating a multivariate node (just repeating W1 and W2):
create_mat <- function(W1, W2) cbind(W1, W2)
vecfun.add("create_mat")

D <- D + node(c("W1.copy1", "W2.copy1"), distr = "rconst", const = c(W1, W2))
D <- D + node(c("W1.copy2", "W2.copy2"), distr = "rconst", const = create_mat(W1, W2))
Dset <- set.DAG(D)
sim(Dset, n=10, rndseed=1)
```

rdistr.template *Template for Writing Custom Distribution Functions*

Description

Template function for writing SimCausal custom distribution wrappers.

Usage

```
rdistr.template(n, arg1, arg2, ...)
```

Arguments

n	Sample size that needs to be generated
arg1	Argument 2
arg2	Argument 1
...	Additional optional parameters

Details

One of the named arguments must be 'n', this argument is passed on to the function automatically by the package and is assigned to the number of samples that needs to be generated from this distribution. Other arguments (in this example arg1 and arg2) must be declared by the user as arguments inside the node() function that uses this distribution, e.g., node("Node1", distr="distr.template", arg1 = ..., arg2 = ...). Both, arg1 and arg2, can be either numeric constants or formulas involving past node names. The constants get passed on to the distribution function unchanged. The formulas are evaluated inside the environment of the simulated data and are passed on to the distribution functions as vectors. The output of the distribution function is expected to be a vector of length n of the sampled covariates.

Value

A vector of length n

rnet.gnm *Call* igraph::sample_gnm *to Generate Random Graph Object According to the G(n,m) Erdos-Renyi Model*

Description

Call igraph::sample_gnm and convert the output to simcausal network matrix. The parameter m of igraph::sample_gnm is derived from n and m_pn as as.integer(m_pn*n)

Usage

```
rnet.gnm(n, m_pn)
```

Arguments

n	Size of the network graph (number of nodes).
m_pn	The total number of edges as a fraction of the sample size n.

Value

A matrix with n rows, each row lists the indices of friends connected to that particular observation.

See Also

rnet.gnp

---

rnet.gnp *Call* igraph::sample_gnp *to Generate Random Graph Object According to the G(n,p) Erdos-Renyi Model*

Description

Call igraph::sample_gnp and convert the output to simcausal network matrix.

Usage

```
rnet.gnp(n, p)
```

Arguments

n	Size of the network graph (number of nodes).
p	Same as igraph::sample_gnp: The probability for drawing an edge between two arbitrary vertices (G(n,p) graph).

Value

A matrix with n rows, each row lists the indices of friends connected to that particular observation.

See Also

rnet.gnm

rnet.SmWorld *Call* igraph::sample_smallworld *to Generate Random Graph Object from the Watts-Strogatz Small-World Model*

Description

Call igraph::sample_smallworld and convert the output to simcausal network matrix. The parameters are the same as those of igraph::sample_smallworld. The loop edges aren't allowed (loops = FALSE) and the multiple edges aren't allowed either multiple = FALSE.

Usage

```
rnet.SmWorld(n, dim, nei, p)
```

Arguments

n	Size of the network graph (the number of nodes).
dim	Same as in igraph::sample_smallworld: Integer constant, the dimension of the starting lattice.
nei	Same as in igraph::sample_smallworld: Integer constant, the neighborhood within which the vertices of the lattice will be connected.
p	Same as in igraph::sample_smallworld: Real constant between zero and one, the rewiring probability.

Value

A matrix with n rows, each row lists the indices of friends connected to that particular observation.

See Also

rnet.gnp, rnet.gnm

set.DAG *Create and Lock DAG Object*

Description

Check current DAG (created with node) for errors and consistency of its node distributions, set the observed data generating distribution. Attempts to simulates several observations to catch any errors in DAG definition. New nodes cannot be added after function set.DAG has been applied.

Usage

```
set.DAG(
  DAG,
  vecfun,
  latent.v,
  n.test = 10,
  verbose = getOption("simcausal.verbose")
)
```

Arguments

DAG	Named list of node objects that together will form a DAG. Temporal ordering of nodes is either determined by the order in which the nodes were added to the DAG (using +node(...) interface) or with an optional "order" argument to node().
vecfun	A character vector with names of the vectorized user-defined node formula functions. See examples and the vignette for more information.
latent.v	The names of the unobserved (latent) DAG node names. These variables will be hidden from the observed simulated data and will be marked differently on the DAG plot.
n.test	Non-negative simulation sample size used for testing the consistency of the DAG object. A larger n.test may be useful when simulating a network of a fixed size (see ?network) or when testing DAG for rare-event errors. A smaller n.test can be better for performance (faster validation of the DAG). Set n.test=0 to completely skip the simulation test (use at your own risk, since calling sim() on such un-tested DAG may lead to uninterpretable errors). Note that when using n.test=0, the plotDAG function *will not* draw *any* child-parent relationships, since the formula parsing is not performed.
verbose	Set to TRUE to print messages on status and information to the console. Turn this off by default using options(simcausal.verbose=FALSE).

Value

A DAG (S3) object, which is a list consisting of node object(s) sorted by their temporal order.

References

Sofrygin O, van der Laan MJ, Neugebauer R (2017). "simcausal R Package: Conducting Transparent and Reproducible Simulation Studies of Causal Effect Estimation with Complex Longitudinal Data." Journal of Statistical Software, 81(2), 1-47. doi: 10.18637/jss.v081.i02.

Examples

```
#----------------------------------------------------------------------------------
# EXAMPLE 1A: Define some Bernoulli nodes, survival outcome Y and put it together in a
# DAG object
#----------------------------------------------------------------------------------
W1node <- node(name = "W1", distr = "rbern",
```

```
prob = plogis(-0.5), order = 1)
W2node <- node(name = "W2", distr = "rbern",
prob = plogis(-0.5 + 0.5 * W1), order = 2)
Anode <- node(name = "A", distr = "rbern",
prob = plogis(-0.5 - 0.3 * W1 - 0.3 * W2), order = 3)
Ynode <- node(name = "Y", distr = "rbern",
prob = plogis(-0.1 + 1.2 * A + 0.3 * W1 + 0.3 * W2), order = 4)
D1Aset <- set.DAG(c(W1node,W2node,Anode,Ynode))

#----------------------------------------------------------------------------------------
# EXAMPLE 1B: Same as 1A using +node interface and no order argument
#----------------------------------------------------------------------------------------
D1B <- DAG.empty()
D1B <- D1B +
  node(name = "W1", distr = "rbern", prob = plogis(-0.5)) +
  node(name = "W2", distr = "rbern", prob = plogis(-0.5 + 0.5 * W1)) +
  node(name = "A", distr = "rbern", prob = plogis(-0.5 - 0.3 * W1 - 0.3 * W2)) +
  node(name = "Y", distr = "rbern", prob = plogis(-0.1 + 1.2 * A + 0.3 * W1 + 0.3 * W2))
D1Bset <- set.DAG(D1B)

#----------------------------------------------------------------------------------------
# EXAMPLE 1C: Same as 1A and 1B using add.nodes interface and no order argument
#----------------------------------------------------------------------------------------
D1C <- DAG.empty()
D1C <- add.nodes(D1C, node(name = "W1", distr = "rbern", prob = plogis(-0.5)))
D1C <- add.nodes(D1C, node(name = "W2", distr = "rbern",
prob = plogis(-0.5 + 0.5 * W1)))
D1C <- add.nodes(D1C, node(name = "A", distr = "rbern",
prob = plogis(-0.5 - 0.3 * W1 - 0.3 * W2)))
D1C <- add.nodes(D1C, node(name = "Y", distr = "rbern",
prob = plogis(-0.1 + 1.2 * A + 0.3 * W1 + 0.3 * W2)))
D1C <- set.DAG(D1C)

#----------------------------------------------------------------------------------------
# EXAMPLE 1D: Add a uniformly distributed node and redefine outcome Y as categorical
#----------------------------------------------------------------------------------------
D_unif <- DAG.empty()
D_unif <- D_unif +
 node("W1", distr = "rbern", prob = plogis(-0.5)) +
 node("W2", distr = "rbern", prob = plogis(-0.5 + 0.5 * W1)) +
 node("W3", distr = "runif", min = plogis(-0.5 + 0.7 * W1 + 0.3 * W2), max = 10) +
 node("An", distr = "rbern", prob = plogis(-0.5 - 0.3 * W1 - 0.3 * W2 - 0.2 * sin(W3)))
# Categorical syntax 1 (probabilities as values):
D_cat_1 <- D_unif + node("Y", distr = "rcat.b1", probs = {0.3; 0.4})
D_cat_1 <- set.DAG(D_cat_1)
# Categorical syntax 2 (probabilities as formulas):
D_cat_2 <- D_unif +
node("Y", distr = "rcat.b1",
probs={plogis(-0.1 + 1.2 * An + 0.3 * W1 + 0.3 * W2 + 0.2 * cos(W3));
   plogis(-0.5 + 0.7 * W1)})
D_cat_2 <- set.DAG(D_cat_2)

#----------------------------------------------------------------------------------------
```

```
# EXAMPLE 2A: Define Bernoulli nodes using R rbinom() function, defining prob argument
# for L2 as a function of node L1
#----------------------------------------------------------------------------------------
D <- DAG.empty()
D <- D +
node("L1", t = 0, distr = "rbinom", prob = 0.05, size = 1) +
node("L2", t = 0, distr = "rbinom", prob = ifelse(L1[0] == 1, 0.5, 0.1), size = 1)
Dset <- set.DAG(D)

#----------------------------------------------------------------------------------------
# EXAMPLE 2B: Equivalent to 2A, passing argument size to rbinom inside a named list
# params
#----------------------------------------------------------------------------------------
D <- DAG.empty()
D <- D +
node("L1", t = 0, distr = "rbinom", prob = 0.05, params = list(size = 1)) +
node("L2", t = 0, distr = "rbinom",
prob = ifelse(L1[0] == 1,0.5,0.1), params = list(size = 1))
Dset <- set.DAG(D)

#----------------------------------------------------------------------------------------
# EXAMPLE 2C: Equivalent to 2A and 2B, define Bernoulli nodes using a wrapper "rbern"
#----------------------------------------------------------------------------------------
D <- DAG.empty()
D <- D +
node("L1", t = 0, distr = "rbern", prob = 0.05) +
node("L2", t = 0, distr = "rbern", prob = ifelse(L1[0] == 1, 0.5, 0.1))
Dset <- set.DAG(D)

#----------------------------------------------------------------------------------------
# EXAMPLE 3: Define node with normal distribution using rnorm() R function
#----------------------------------------------------------------------------------------
D <- DAG.empty()
D <- D + node("L2", t = 0, distr = "rnorm", mean = 10, sd = 5)
Dset <- set.DAG(D)

#----------------------------------------------------------------------------------------
# EXAMPLE 4: Define 34 Bernoulli nodes, or 2 Bernoulli nodes over 17 time points,
#----------------------------------------------------------------------------------------
t_end <- 16
D <- DAG.empty()
D <- D +
node("L2", t = 0:t_end, distr = "rbinom",
prob = ifelse(t == t_end, 0.5, 0.1), size = 1) +
node("L1", t = 0:t_end, distr = "rbinom",
prob = ifelse(L2[0] == 1, 0.5, 0.1), size = 1)
Dset <- set.DAG(D)
sim(Dset, n=10)

#----------------------------------------------------------------------------------------
# EXAMPLE 5: Defining new distribution function 'rbern', defining and passing a custom
# vectorized node function 'customfun'
#----------------------------------------------------------------------------------------
```

```
rbern <- function(n, prob) { # defining a bernoulli wrapper based on R rbinom function
  rbinom(n = n, prob = prob, size = 1)
}
customfun <- function(arg, lambda) {
  res <- ifelse(arg == 1, lambda, 0.1)
  res
}
D <- DAG.empty()
D <- D +
node("W1", distr = "rbern", prob = 0.05) +
node("W2", distr = "rbern", prob = customfun(W1, 0.5)) +
node("W3", distr = "rbern", prob = ifelse(W1 == 1, 0.5, 0.1))
D1d <- set.DAG(D, vecfun = c("customfun"))
sim(D1d, n = 10, rndseed = 1)

#----------------------------------------------------------------------------------------
# EXAMPLE 6: Defining latent variables I and U.Y (will be hidden from simulated data)
#----------------------------------------------------------------------------------------
D <- DAG.empty()
D <- D +
  node("I",
    distr = "rcat.b1",
    probs = c(0.1, 0.2, 0.2, 0.2, 0.1, 0.1, 0.1)) +
  node("W1",
    distr = "rnorm",
    mean = ifelse(I == 1, 0, ifelse(I == 2, 3, 10)) + 0.6 * I,
    sd = 1) +
  node("W2",
    distr = "runif",
    min = 0.025*I, max = 0.7*I) +
  node("W3",
    distr = "rbern",
    prob = plogis(-0.5 + 0.7*W1 + 0.3*W2 - 0.2*I)) +
  node("A",
    distr = "rbern",
    prob = plogis(+4.2 - 0.5*W1 + 0.2*W2/2 + 0.2*W3)) +
node("U.Y", distr = "rnorm", mean = 0, sd = 1) +
  node("Y",
    distr = "rconst",
    const = -0.5 + 1.2*A + 0.1*W1 + 0.3*W2 + 0.2*W3 + 0.2*I + U.Y)
Dset1 <- set.DAG(D, latent.v = c("I", "U.Y"))
sim(Dset1, n = 10, rndseed = 1)

#----------------------------------------------------------------------------------------
# EXAMPLE 7: Multivariate random variables
#----------------------------------------------------------------------------------------
if (requireNamespace("mvtnorm", quietly = TRUE)) {
  D <- DAG.empty()
  # 2 dimensional normal (uncorrelated), using rmvnorm function from rmvnorm package:
  D <- D +
    node(c("X1","X2"), distr = "mvtnorm::rmvnorm",
      asis.params = list(mean = "c(0,1)")) +
  # Can define a degenerate (rconst) multivariate node:
```

```
    node(c("X1.copy", "X2.copy"), distr = "rconst", const = c(X1, X2))
  Dset1 <- set.DAG(D, verbose = TRUE)
  sim(Dset1, n = 10)
}

# On the other hand this syntax wont work,
# since simcausal will parse c(0,1) into a two column matrix:
## Not run:
D <- DAG.empty()
D <- D + node(c("X1","X2"), distr = "mvtnorm::rmvnorm", mean = c(0,1))
Dset1 <- set.DAG(D, verbose = TRUE)

## End(Not run)

if (requireNamespace("mvtnorm", quietly = TRUE)) {
  D <- DAG.empty()
  # Bivariate normal (correlation coef 0.75):
  D <- D +
    node(c("X1","X2"), distr = "mvtnorm::rmvnorm",
      asis.params = list(mean = "c(0,1)",
        sigma = "matrix(c(1,0.75,0.75,1), ncol=2)")) +
  # Can use any component of such multivariate nodes when defining new nodes:
    node("A", distr = "rconst", const = 1 - X1)
  Dset2 <- set.DAG(D, verbose = TRUE)
  sim(Dset2, n = 10)
}

# Time-varying multivar node (3 time-points, 2 dimensional normal)
# plus changing the mean over time, as as function of t:
if (requireNamespace("mvtnorm", quietly = TRUE)) {
  D <- DAG.empty()
  D <- D +
    node(c("X1", "X2"), t = 0:2, distr = "mvtnorm::rmvnorm",
      asis.params = list(
        mean = "c(0,1) + t",
        sigma = "matrix(rep(0.75,4), ncol=2)"))
  Dset5b <- set.DAG(D)
  sim(Dset5b, n = 10)
}

# Two ways to define the same bivariate uniform copula:
if (requireNamespace("copula", quietly = TRUE)) {
  D <- DAG.empty()
  D <- D +
  # with a warning since normalCopula() returns an object unknown to simcausal:
    node(c("X1","X2"), distr = "copula::rCopula",
      copula = eval(copula::normalCopula(0.75, dim = 2))) +
  # same, as above:
    node(c("X3","X4"), distr = "copula::rCopula",
      asis.params = list(copula = "copula::normalCopula(0.75, dim = 2)"))
  vecfun.add("qbinom")
  # Bivariate binomial derived from previous copula, with same correlation:
  D <- D +
```

```
    node(c("A.Bin1", "A.Bin2"), distr = "rconst",
      const = c(qbinom(X1, 10, 0.5), qbinom(X2, 15, 0.7)))
  Dset3 <- set.DAG(D)
  sim(Dset3, n = 10)
}

# Same as "A.Bin1" and "A.Bin2", but directly using rmvbin function in bindata package:
if (requireNamespace("bindata", quietly = TRUE)) {
  D <- DAG.empty()
  D <- D +
    node(c("B.Bin1","B.Bin2"), distr = "bindata::rmvbin",
      asis.params = list(
        margprob = "c(0.5, 0.5)",
        bincorr = "matrix(c(1,0.75,0.75,1), ncol=2)"))
  Dset4b <- set.DAG(D)
  sim(Dset4b, n = 10)
}

#----------------------------------------------------------------------------------------
# EXAMPLE 8: Combining simcausal non-standard evaluation with eval() forced evaluation
#----------------------------------------------------------------------------------------
coefAi <- 1:10
D <- DAG.empty()
D <- D +
  node("A", t = 1, distr = "rbern", prob = 0.7) +
  node("A", t = 2:10, distr = "rconst", const = eval(coefAi[t]) * A[t-1])
Dset8 <- set.DAG(D)
sim(Dset8, n = 10)

#----------------------------------------------------------------------------------------
# TWO equivalent ways of creating a multivariate node (combining nodes W1 and W2):
#----------------------------------------------------------------------------------------
D <- DAG.empty()
D <- D + node("W1", distr = "rbern", prob = 0.45)
D <- D + node("W2", distr = "rconst", const = 1)

# option 1:
D <- D + node(c("W1.copy1", "W2.copy1"), distr = "rconst", const = c(W1, W2))

# equivalent option 2:
create_mat <- function(W1, W2) cbind(W1, W2)
vecfun.add("create_mat")
D <- D + node(c("W1.copy2", "W2.copy2"), distr = "rconst", const = create_mat(W1, W2))

Dset <- set.DAG(D)
sim(Dset, n=10, rndseed=1)
```

---

set.targetE *Define Non-Parametric Causal Parameters*

Description

Set up the causal target parameter as a vector of expectations, ratio of expectations or contrast of expectations (average treatment effect) over the nodes of specified actions. These settings are then used to evaluate the true value of the causal target parameter by calling `eval.target` function.

Usage

```
set.targetE(DAG, outcome, t, param, ..., attr = list())
```

Arguments

`DAG`	Object specifying the directed acyclic graph (DAG) for the observed data
`outcome`	Name of the outcome node
`t`	Integer vector of time points to use for expectations, has to be omitted or NULL for non-time-varying DAGs.
`param`	A character vector `"ActionName1"`, specifying the action name for the expectation target parameter; `"ActionName1 / ActionName2"`, for the ratio of expectations of `outcome` nodes for actions `"ActionName1"` and `"ActionName2"`; `"ActionName1 - ActionName2"` for the contrast of expectations of `outcome` for actions `"ActionName1"` and `"ActionName2"`
`...`	Additional attributes (to be used in future versions)
`attr`	Additional attributes (to be used in future versions)

Value

A modified DAG object with the target parameter saved as part of the DAG, this DAG can now be passed as an argument to `eval.target` function for actual Monte-Carlo evaluation of the target parameter. See Examples.

References

Sofrygin O, van der Laan MJ, Neugebauer R (2017). "simcausal R Package: Conducting Transparent and Reproducible Simulation Studies of Causal Effect Estimation with Complex Longitudinal Data." Journal of Statistical Software, 81(2), 1-47. doi: 10.18637/jss.v081.i02.

Examples

```
#----------------------------------------------------------------------------------------
# EXAMPLE 1: DAG with single point treatment
#----------------------------------------------------------------------------------------
# Define a DAG with single-point treatment ("Anode")
D <- DAG.empty()
D <- D + node("W1", distr="rbern", prob=plogis(-0.5))
D <- D + node("W2", distr="rbern", prob=plogis(-0.5 + 0.5*W1))
D <- D + node("Anode", distr="rbern", prob=plogis(-0.5 - 0.3*W1 - 0.3*W2))
D <- D + node("Y", distr="rbern", prob=plogis(-0.1 + 1.2*Anode + 0.3*W1 + 0.3*W2),
EFU=TRUE)
D_WAY <- set.DAG(D)
```

```
# Defining interventions (actions)
# define action "A1" that sets the treatment node to constant 1
D_WAY <- D_WAY + action("A1", nodes=node("Anode",distr="rbern", prob=1))
# define another action "A0" that sets the treatment node to constant 0
D_WAY <- D_WAY + action("A0", nodes=node("Anode",distr="rbern", prob=0))
#----------------------------------------------------------------------------------------
# Defining and calculating causal parameters:
#----------------------------------------------------------------------------------------
# Counterfactual mean of node "Y" under action "A1"
D_WAY <- set.targetE(D_WAY, outcome="Y", param="A1")
eval.target(D_WAY, n=10000)

# Contrasts of means of "Y" under action "A1" minus action "A0"
D_WAY <- set.targetE(D_WAY, outcome="Y", param="A1-A0")
eval.target(D_WAY, n=10000)

# Ratios of "Y" under action "A1" over action "A0"
D_WAY <- set.targetE(D_WAY, outcome="Y", param="A1/A0")
eval.target(D_WAY, n=10000)

# Alternative parameter evaluation by passing already simulated full data to
# \code{eval.target}
X_dat1 <- simfull(A(D_WAY), n=10000)
D_WAY <- set.targetE(D_WAY, outcome="Y", param="A1/A0")
eval.target(D_WAY, data=X_dat1)

#----------------------------------------------------------------------------------------
# EXAMPLE 2: DAG with time-varying outcomes (survival outcome)
#----------------------------------------------------------------------------------------
# Define longitudinal data structure over 6 time-points t=(0:5)
t_end <- 5
D <- DAG.empty()
D <- D + node("L2", t=0, distr="rbern", prob=0.05)
D <- D + node("L1", t=0, distr="rbern", prob=ifelse(L2[0]==1,0.5,0.1))
D <- D + node("A1", t=0, distr="rbern", prob=ifelse(L1[0]==1 & L2[0]==0, 0.5,
ifelse(L1[0]==0 & L2[0]==0, 0.1,
ifelse(L1[0]==1 & L2[0]==1, 0.9, 0.5))))
D <- D + node("A2", t=0, distr="rbern", prob=0, order=4, EFU=TRUE)
D <- D + node("Y",  t=0, distr="rbern",
prob=plogis(-6.5 + L1[0] + 4*L2[0] + 0.05*I(L2[0]==0)),
EFU=TRUE)
D <- D + node("L2", t=1:t_end, distr="rbern", prob=ifelse(A1[t-1]==1, 0.1,
ifelse(L2[t-1]==1, 0.9,
min(1,0.1 + t/16))))
D <- D + node("A1", t=1:t_end, distr="rbern", prob=ifelse(A1[t-1]==1, 1,
ifelse(L1[0]==1 & L2[0]==0, 0.3,
ifelse(L1[0]==0 & L2[0]==0, 0.1,
ifelse(L1[0]==1 & L2[0]==1, 0.7,
0.5)))))
D <- D + node("A2", t=1:t_end, distr="rbern", prob=0, EFU=TRUE)
D <- D + node("Y",  t=1:t_end, distr="rbern",
prob=plogis(-6.5 + L1[0] + 4*L2[t] + 0.05*sum(I(L2[0:t]==rep(0,(t+1))))),
```

```
EFU=TRUE)
D <- set.DAG(D)

# Add two dynamic actions (indexed by values of the parameter theta={0,1})
# Define intervention nodes
act_t0_theta <- node("A1",t=0, distr="rbern", prob=ifelse(L2[0] >= theta,1,0))
act_tp_theta <- node("A1",t=1:t_end, distr="rbern",
prob=ifelse(A1[t-1]==1,1,ifelse(L2[t] >= theta,1,0)))
# Add two actions to current DAG object
D <- D + action("A1_th0", nodes=c(act_t0_theta, act_tp_theta), theta=0)
D <- D + action("A1_th1", nodes=c(act_t0_theta, act_tp_theta), theta=1)
#----------------------------------------------------------------------------------
# Defining and calculating the target parameter
#----------------------------------------------------------------------------------
# Counterfactual mean of node "Y" at time-point t=4 under action "A1_th0"
D <- set.targetE(D, outcome="Y", t=4, param="A1_th0")
eval.target(D, n=5000)

# Vector of counterfactual means of"Y" over all time points under action "A1_th1"
D <- set.targetE(D, outcome="Y", t=0:5, param="A1_th1")
eval.target(D, n=5000)

# Vector of counterfactual contrasts of "Y" over all time points
# for action "A1_th1" minus action "A1_th0"
D <- set.targetE(D, outcome="Y", t=0:5, param="A1_th1 - A1_th0")
eval.target(D, n=5000)

# Vector of counterfactual ratios of "Y" over all time points
# for action "A1_th0" over action "A1_th1"
D <- set.targetE(D, outcome="Y", t=0:5, param="A1_th0 / A1_th1")
eval.target(D, n=5000)
```

`set.targetMSM` *Define Causal Parameters with a Working Marginal Structural Model (MSM)*

Description

Set up the MSM causal target parameter for the current DAG object. These settings can be later used to evaluate the true value of the MSM parameter on the full (counterfactual) data by calling `eval.target` function.

Usage

```
set.targetMSM(
  DAG,
  outcome,
  t,
  formula,
```

```
  family = "quasibinomial",
  hazard,
  ...,
  attr = list()
)
```

Arguments

DAG	Object specifying the directed acyclic graph (DAG) for the observed data
outcome	Name of the outcome node
t	Vector of time points which are used for pooling the outcome
formula	MSM formula for modeling pooled outcome on the full data with glm regression. Left hand side should be equal to the outcome, right hand side can include baseline covariates, action-specific attribute names and time-dependent treatment summary measures. See Details.
family	Model family to use in the glm regression
hazard	When TRUE MSM fits the discrete hazard function for survival outcome (if outcome node had EOF=TRUE attribute)
...	Additional attributes (to be used in future versions)
attr	Additional attributes (to be used in future versions)

Details

Enclosing an MSM formula term inside S(), e.g., S(mean(A[0:t])), forces this term to be evaluated as a summary measure of time-indexed nodes in the full data environment. All such MSM terms are parsed and then evaluated inside the previously simulated full data environment, each S() term is then replaced with a vector name 'XMSMterms.i' that is a result of this evaluation.

Value

A modified DAG object with well-defined target parameter saved as part of the DAG, this DAG can now be passed as an argument to eval.target function for actual Monte-Carlo evaluation of the target parameter. See Examples.

References

Sofrygin O, van der Laan MJ, Neugebauer R (2017). "simcausal R Package: Conducting Transparent and Reproducible Simulation Studies of Causal Effect Estimation with Complex Longitudinal Data." Journal of Statistical Software, 81(2), 1-47. doi: 10.18637/jss.v081.i02.

Examples

```
#----------------------------------------------------------------------------------
# DAG with time-varying outcomes (survival outcome)
#----------------------------------------------------------------------------------
# Define longitudinal data structure over 6 time-points t=(0:5)
t_end <- 5
D <- DAG.empty()
```

```
D <- D + node("L2", t=0, distr="rbern", prob=0.05)
D <- D + node("L1", t=0, distr="rbern", prob=ifelse(L2[0]==1,0.5,0.1))
D <- D + node("A1", t=0, distr="rbern", prob=ifelse(L1[0]==1 & L2[0]==0, 0.5,
ifelse(L1[0]==0 & L2[0]==0, 0.1,
ifelse(L1[0]==1 & L2[0]==1, 0.9, 0.5))))
D <- D + node("A2", t=0, distr="rbern", prob=0, order=4, EFU=TRUE)
D <- D + node("Y",  t=0, distr="rbern",
prob=plogis(-6.5 + L1[0] + 4*L2[0] + 0.05*I(L2[0]==0)),
EFU=TRUE)
D <- D + node("L2", t=1:t_end, distr="rbern", prob=ifelse(A1[t-1]==1, 0.1,
ifelse(L2[t-1]==1, 0.9,
min(1,0.1 + t/16))))
D <- D + node("A1", t=1:t_end, distr="rbern", prob=ifelse(A1[t-1]==1, 1,
ifelse(L1[0]==1 & L2[0]==0, 0.3,
ifelse(L1[0]==0 & L2[0]==0, 0.1,
ifelse(L1[0]==1 & L2[0]==1, 0.7,
0.5)))))
D <- D + node("A2", t=1:t_end, distr="rbern", prob=0, EFU=TRUE)
D <- D + node( "Y",  t=1:t_end, distr="rbern",
prob=plogis(-6.5 + L1[0] + 4*L2[t] + 0.05*sum(I(L2[0:t]==rep(0,(t+1))))),
EFU=TRUE)
D <- set.DAG(D)

# Add two dynamic actions (indexed by values of the parameter theta={0,1})
# Define intervention nodes
act_t0_theta <- node("A1",t=0, distr="rbern", prob=ifelse(L2[0] >= theta,1,0))
act_tp_theta <- node("A1",t=1:t_end, distr="rbern",
prob=ifelse(A1[t-1]==1,1,ifelse(L2[t] >= theta,1,0)))
# Add two actions to current DAG object
D <- D + action("A1_th0", nodes=c(act_t0_theta, act_tp_theta), theta=0)
D <- D + action("A1_th1", nodes=c(act_t0_theta, act_tp_theta), theta=1)

#----------------------------------------------------------------------------------------
# MSM EXAMPLE 1: Modeling survival over time
#----------------------------------------------------------------------------------------
# Modeling pooled survival Y_t over time as a projection on the following working
# logistic model:
msm.form <- "Y ~ theta + t + I(theta*t)"
D <- set.targetMSM(D, outcome="Y", t=0:5, formula=msm.form, family="binomial",
hazard=FALSE)
MSMres <- eval.target(D, n=1000)
MSMres$coef

#----------------------------------------------------------------------------------------
# MSM EXAMPLE 2: Modeling survival over time with exposure-based summary measures
#----------------------------------------------------------------------------------------
# Now we want to model Y_t by adding a summary measure covariate defined as mean
# exposure A1 from time 0 to t;
# Enclosing any term inside S() forces its evaluation in the environment
# of the full (counterfactual) data.
msm.form_sum <- "Y ~ theta + t + I(theta*t) + S(mean(A1[0:t]))"
D <- set.targetMSM(D, outcome="Y", t=0:5, formula=msm.form_sum, family="binomial",
hazard=FALSE)
```

```
MSMres <- eval.target(D, n=1000)
MSMres$coef
```

`sim` *Simulate Observed or Full Data from* `DAG` *Object*

Description

This function simulates full data based on a list of intervention DAGs, returning a list of `data.frames`. See the vignette for examples and detailed description.

Usage

```
sim(
  DAG,
  actions,
  n,
  wide = TRUE,
  LTCF = NULL,
  rndseed = NULL,
  rndseed.reset.node = NULL,
  verbose = getOption("simcausal.verbose")
)
```

Arguments

`DAG`	A DAG objects that has been locked with set.DAG(DAG). Observed data from this DAG will be simulated if actions argument is omitted.
`actions`	Character vector of action names which will be extracted from the DAG object. Alternatively, this can be a list of action DAGs selected with `A(DAG)` function, in which case the argument `DAG` is unused. When `actions` is omitted, the function returns simulated observed data (see `simobs`).
`n`	Number of observations to sample.
`wide`	A logical, if TRUE the output data is generated in wide format, if FALSE, the output longitudinal data in generated in long format
`LTCF`	If forward imputation is desired for the missing variable values, this argument should be set to the name of the node that indicates the end of follow-up event.
`rndseed`	Seed for the random number generator.
`rndseed.reset.node`	When `rndseed` is specified, use this argument to specify the name of the `DAG` node at which the random number generator seed is reset back to `NULL` (simulation function will call `set.seed(NULL)`). Can be useful if one wishes to simulate data using the set seed `rndseed` only for the first K nodes of the DAG and use an entirely random sample when simulating the rest of the nodes starting at K+1 and on. The name of such (K+1)th order `DAG` node should be then specified with this argument.

verbose — Set to TRUE to print messages on status and information to the console. Turn this off by default using options(simcausal.verbose=FALSE).

Value

If actions argument is missing a simulated data.frame is returned, otherwise the function returns a named list of action-specific simulated data.frames with action names giving names to corresponding list items.

Forward Imputation

By default, when LTCF is left unspecified, all variables that follow after any end of follow-up (EFU) event are set to missing (NA). The end of follow-up event occurs when a binary node of type EFU=TRUE is equal to 1, indicating a failing or right-censoring event. To forward impute the values of the time-varying nodes after the occurrence of the EFU event, set the LTCF argument to a name of the EFU node representing this event. For additional details and examples see the vignette and doLTCF function.

References

Sofrygin O, van der Laan MJ, Neugebauer R (2017). "simcausal R Package: Conducting Transparent and Reproducible Simulation Studies of Causal Effect Estimation with Complex Longitudinal Data." Journal of Statistical Software, 81(2), 1-47. doi: 10.18637/jss.v081.i02.

See Also

simobs - a wrapper function for simulating observed data only; simfull - a wrapper function for simulating full data only; doLTCF - forward imputation of the missing values in already simulating data; DF.to.long, DF.to.longDT - converting longitudinal data from wide to long formats.

Other simulation functions: simfull(), simobs()

Examples

```
t_end <- 10
lDAG <- DAG.empty()
lDAG <- lDAG +
node(name = "L2", t = 0, distr = "rconst", const = 0) +
node(name = "A1", t = 0, distr = "rconst", const = 0) +
node(name = "L2", t = 1:t_end, distr = "rbern",
  prob = ifelse(A1[t - 1]  ==  1, 0.1,
  ifelse(L2[t-1] == 1, 0.9,
        min(1,0.1 + t/t_end)))) +
node(name = "A1", t = 1:t_end, distr = "rbern",
  prob = ifelse(A1[t - 1]  ==  1, 1,
   ifelse(L2[0] == 0, 0.3,
  ifelse(L2[0] == 0, 0.1,
   ifelse(L2[0] == 1, 0.7, 0.5))))) +
node(name = "Y", t = 1:t_end, distr = "rbern",
  prob = plogis(-6.5 + 4 * L2[t] + 0.05 * sum(I(L2[0:t] == rep(0,(t + 1))))),
  EFU = TRUE)
lDAG <- set.DAG(lDAG)
```

```
#----------------------------------------------------------------------------------------
# EXAMPLE 1. No forward imputation.
#----------------------------------------------------------------------------------------
Odat.wide <- sim(DAG = lDAG, n = 1000, rndseed = 123)
Odat.wide[c(21,47), 1:18]
Odat.wideLTCF <- sim(DAG = lDAG, n = 1000, LTCF = "Y", rndseed = 123)
Odat.wideLTCF[c(21,47), 1:18]
#----------------------------------------------------------------------------------------
# EXAMPLE 2. With forward imputation.
#----------------------------------------------------------------------------------------
Odat.wideLTCF2 <- doLTCF(data = Odat.wide, LTCF = "Y")
Odat.wideLTCF2[c(21,47), 1:18]
# all.equal(Odat.wideLTCF, Odat.wideLTCF2)
```

`simcausal` *Simulating Longitudinal Data with Causal Inference Applications*

Description

The **simcausal** R package is a tool for specification and simulation of complex longitudinal data structures that are based on structural equation models. The package provides a flexible tool for conducting transparent and reproducible simulation studies, with a particular emphasis on the types of data and interventions frequently encountered in typical causal inference problems, such as, observational data with time-dependent confounding, selection bias, and random monitoring processes. The package interface allows for concise expression of complex functional dependencies between a large number of nodes, where each node may represent a time-varying random variable. The package allows for specification and simulation of counterfactual data under various user-specified interventions (e.g., static, dynamic, deterministic, or stochastic). In particular, the interventions may represent exposures to treatment regimens, the occurrence or non-occurrence of right-censoring events, or of clinical monitoring events. **simcausal** enables the computation of a selected set of user-specified features of the distribution of the counterfactual data that represent common causal quantities of interest, such as, treatment-specific means, the average treatment effects and coefficients from working marginal structural models. For additional details and examples please see the package vignette and the function-specific documentation.

Documentation

- To see the package vignette use: `vignette("simcausal_vignette", package="simcausal")`
- To see all available package documentation use: `help(package = 'simcausal')`

Routines

The following routines will be generally invoked by a user, in the same order as presented below.

`DAG.empty` Initiates an empty `DAG` object that contains no nodes.

`node` Defines node(s) in the structural equation model and its conditional distribution(s) using a language of vector-like R expressions. A call to `node` can specify either a single node or multiple nodes at once.

`add.nodes` **or** `+node` Provide two equivalent ways of growing the structural equation model by adding new nodes and their conditional distributions. Sequentially define nodes in the `DAG` object, with each node representing the outcomes of one or more structural equation(s), altogether making-up the causal model of interest.

`set.DAG` Performs consistency checks and locks the `DAG` object so that no additional nodes can be subsequently added to the structural equation model.

`sim` **or** `simobs` Simulates iid observations of the complete node sequence defined by the `DAG` object. The output dataset is stored as a `data.frame` and is referred to as the *observed data*.

`add.action` **or** `+action` Provide two equivalent ways to define one or more actions. Each action modifies the conditional distribution for a subset of nodes in the original `DAG` object. The resulting data generating distribution is referred to as the post-intervention distribution. It is saved in the `DAG` object alongside the original structural equation model (`DAG` object).

`sim` **or** `simfull` Simulates independent observations from one or more post-intervention distribution(s). Produces a named list of `data.frames`, collectively referred to as the *full data*. The number of output `data.frames` is equal to the number of post-intervention distributions specified in the `actions` argument, where each `data.frame` object is an iid sample from a particular post-intervention distribution.

`set.targetE` **and** `set.targetMSM` Define two distinct types of target causal parameters. The function `set.targetE` defines causal parameters as the expected value(s) of `DAG` node(s) under one post-intervention distribution or the contrast of such expected value(s) from two post-intervention distributions. The function `set.targetMSM` defines causal parameters based on a user-specified **working** marginal structural model.

`eval.target` Evaluates the previously defined causal parameter using simulated full data

Data structures

The following most common types of output are produced by the package:

- *parameterized causal* `DAG` *model* - object that specifies the structural equation model, along with interventions and the causal target parameter of interest.
- *observed data* - data simulated from the (pre-intervention) distribution specified by the structural equation model.
- *full data* - data simulated from one or more post-intervention distributions defined by actions on the structural equation model.
- *causal target parameter* - the true value of the causal target parameter evaluated with full data.

Updates

Check for updates and report bugs at `https://github.com/osofr/simcausal`.

Author(s)

Maintainer: Fred Gruber `<fgruber@gmail.com>` [contributor]

Authors:

- Oleg Sofrygin `<oleg.sofrygin@gmail.com>`
- Mark J. van der Laan `<laan@berkeley.edu>`
- Romain Neugebauer `<Romain.S.Neugebauer@kp.org>`

References

Sofrygin O, van der Laan MJ, Neugebauer R (2017). "simcausal R Package: Conducting Transparent and Reproducible Simulation Studies of Causal Effect Estimation with Complex Longitudinal Data." Journal of Statistical Software, 81(2), 1-47. doi: 10.18637/jss.v081.i02.

See Also

Useful links:

- https://github.com/osofr/simcausal
- Report bugs at https://github.com/osofr/simcausal/issues

`simfull` *Simulate Full Data (From Action DAG(s))*

Description

This function simulates full data based on a list of intervention DAGs, returning a list of `data.frames`.

Usage

```
simfull(
  actions,
  n,
  wide = TRUE,
  LTCF = NULL,
  rndseed = NULL,
  rndseed.reset.node = NULL,
  verbose = getOption("simcausal.verbose")
)
```

Arguments

`actions`	Actions specifying the counterfactual DAG. This argument must be either an object of class DAG.action or a list of DAG.action objects.
`n`	Number of observations to sample.
`wide`	A logical, if TRUE the output data is generated in wide format, if FALSE, the output longitudinal data in generated in long format
`LTCF`	If forward imputation is desired for the missing variable values, this argument should be set to the name of the node that indicates the end of follow-up event. See the vignette, `sim` and `doLTCF` for additional details.
`rndseed`	Seed for the random number generator.

rndseed.reset.node
: When rndseed is specified, use this argument to specify the name of the DAG node at which the random number generator seed is reset back to NULL (simulation function will call set.seed(NULL)). Can be useful if one wishes to simulate data using the set seed rndseed only for the first K nodes of the DAG and use an entirely random sample when simulating the rest of the nodes starting at K+1 and on. The name of such (K+1)th order DAG node should be then specified with this argument.

verbose
: Set to TRUE to print messages on status and information to the console. Turn this off by default using options(simcausal.verbose=FALSE).

Value

A named list, each item is a data.frame corresponding to an action specified by the actions argument, action names are used for naming these list items.

See Also

simobs - a wrapper function for simulating observed data only; sim - a wrapper function for simulating both types of data; doLTCF for forward imputation of the missing values in already simulating data; DF.to.long, DF.to.longDT - converting longitudinal data from wide to long formats.

Other simulation functions: sim(), simobs()

---

simobs *Simulate Observed Data*

Description

This function simulates observed data from a DAG object.

Usage

```
simobs(
  DAG,
  n,
  wide = TRUE,
  LTCF = NULL,
  rndseed = NULL,
  rndseed.reset.node = NULL,
  verbose = getOption("simcausal.verbose")
)
```

Arguments

DAG
: A DAG objects that has been locked with set.DAG(DAG). Observed data from this DAG will be simulated.

n
: Number of observations to sample.

wide — A logical, if TRUE the output data is generated in wide format, if FALSE, the output longitudinal data in generated in long format

LTCF — If forward imputation is desired for the missing variable values, this argument should be set to the name of the node that indicates the end of follow-up event. See the vignette, sim and doLTCF for additional details.

rndseed — Seed for the random number generator.

rndseed.reset.node — When rndseed is specified, use this argument to specify the name of the DAG node at which the random number generator seed is reset back to NULL (simulation function will call set.seed(NULL)). Can be useful if one wishes to simulate data using the set seed rndseed only for the first K nodes of the DAG and use an entirely random sample when simulating the rest of the nodes starting at K+1 and on. The name of such (K+1)th order DAG node should be then specified with this argument.

verbose — Set to TRUE to print messages on status and information to the console. Turn this off by default using options(simcausal.verbose=FALSE).

Value

A data.frame where each column is sampled from the conditional distribution specified by the corresponding DAG object node.

See Also

simfull - a wrapper function for simulating full data only; sim - a wrapper function for simulating both types of data; doLTCF for forward imputation of the missing values in already simulating data; DF.to.long, DF.to.longDT - converting longitudinal data from wide to long formats.

Other simulation functions: sim(), simfull()

---

sparseAdjMat.to.igraph

Convert Network from Sparse Adjacency Matrix into igraph Object

---

Description

Uses graph_from_adjacency_matrix function from the igraph package to convert the network in sparse adjacency matrix format into igraph network object.

Usage

```
sparseAdjMat.to.igraph(sparseAdjMat, mode = "directed")
```

Arguments

sparseAdjMat	Network represented as a sparse adjacency matrix (S4 class object dgCMatrix from package Matrix). NOTE: for directed graphs the friend IDs pointing into vertex i are assumed to be listed in the column i (i.e, which(sparseAdjMat[,i]) are friends of i).
mode	Character scalar, passed on to igraph::graph_from_adjacency_matrix, specifies how igraph should interpret the supplied matrix. See ?igraph::graph_from_adjacency_matrix for details.

Value

A list containing the network object(s) of type DAG.net.

See Also

network; igraph.to.sparseAdjMat; sparseAdjMat.to.NetInd; NetInd.to.sparseAdjMat;

---

sparseAdjMat.to.NetInd

Convert Network from Sparse Adjacency Matrix into Network IDs Matrix

---

Description

Convert network represented by a sparse adjacency matrix into simcausal network IDs matrix (NetInd_k).

Usage

```
sparseAdjMat.to.NetInd(sparseAdjMat, trimKmax)
```

Arguments

sparseAdjMat	Network represented as a sparse adjacency matrix (S4 class object dgCMatrix from package Matrix). NOTE: The friends (row numbers) of observation i are assumed to be listed in column i (i.e, which(sparseAdjMat[,i]) are friends of i).
trimKmax	Trim the maximum number of friends to this integer value. If this argument is not missing, the conversion network matrix obtained from sparseAdjMat will be trimmed, so that each observation has at most trimKmax friends. The trimming initiates from the last column of the network ID matrix, removing columns until only trimKmax columns are left.

Value

A named list with 3 items: 1) NetInd_k; 2) nF; and 3) Kmax. 1) NetInd_k - matrix of network IDs of dimension (n=nrow(sparseAdjMat),Kmax), where each row i consists of the network IDs (friends) for observation i. Remainders are filled with NAs. 2) nF - integer vector of length n specifying the number of friends for each observation. 3) Kmax - integer constant specifying the maximum observed number of friends in input sparseAdjMat (this is the column dimension for the output matrix NetInd_k).

See Also

network; NetInd.to.sparseAdjMat; sparseAdjMat.to.igraph; igraph.to.sparseAdjMat;

---

vecfun.add *Add Custom Vectorized Functions*

Description

Add user-defined function names to a global list of custom vectorized functions. The functions in vecfun_names are intended for use inside the node formulas. Adding functions to this list will generally greatly expedite the simulation run time. Any node formula calling a function on this list will be evaluated "as is", the function should be written to accept arguments as either vectors of length n or as matrices with n rows. Adding function to this list will effects simulation from all DAG objects that call this function. See vignette for more details.

Usage

```
vecfun.add(vecfun_names)
```

Arguments

vecfun_names	A character vector of function names that will be treated as "vectorized" by the node formula R parser

Value

An old vector of user-defined vectorized function names

vecfun.all.print *Print Names of All Vectorized Functions*

Description

Print all vectorized function names (build-in and user-defined).

Usage

```
vecfun.all.print()
```

Value

A vector of build-in and user-defined vectorized function names

vecfun.print *Print Names of Custom Vectorized Functions*

Description

Print current user-defined vectorized function names.

Usage

```
vecfun.print()
```

Value

A vector of vectorized function names

vecfun.remove *Remove Custom Vectorized Functions*

Description

Remove user-defined function names from a global list of custom vectorized functions. See vignette for more details.

Usage

```
vecfun.remove(vecfun_names)
```

Arguments

vecfun_names	A character vector of function names that will be removed from the custom list

Value

An old vector of user-defined vectorized function names

`vecfun.reset` *Reset Custom Vectorized Function List*

Description

Reset a listing of user-defined vectorized functions.

Usage

```
vecfun.reset()
```

Value

An old vector of user-defined vectorized function names

Index