Package: PCBN (via r-universe)

May 17, 2026

Title Inference of Pair-Copula Bayesian Networks

Version 0.1.1

Description Creates, fits and samples Pair-Copula Bayesian Networks (PCBN) under some restrictions on the underlying Directed Acyclic Graph (DAG), that is, no active cycles nor interfering v-structures, following Derumigny, Horsman and Kurowicka (2025) <doi:10.48550/arXiv.2510.03518>.

License GPL (>= 2)

Encoding UTF-8

Imports bnlearn, igraph, r2r, VineCopula

RoxygenNote 7.3.3

Suggests knitr, rmarkdown, testthat (>= 3.0.0), data.tree, Rgraphviz

URL https://github.com/AlexisDerumigny/PCBN

BugReports https://github.com/AlexisDerumigny/PCBN/issues

Config/testthat/edition 3

VignetteBuilder knitr

Config/pak/sysreqs libglpk-dev libxml2-dev

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

Date/Publication 2025-11-18 09:06:21 UTC

RemoteUrl https://github.com/alexisderumigny/pcbn

RemoteRef HEAD

RemoteSha 92bd580ea06067b169002734a393b2e1a0977c0e

Contents

active_cycles 2
B_sets_are_increasing 4
B_sets_cut_increments 5
B_sets_make_unique 6

complete_and_check_orders . . . 6
compute_sample_margin . . . 7
create_empty_DAG . . . 9
DAG_to_restrictedDAG . . . 9
default_envir . . . 10
dsep_set . . . 12
extend_orders . . . 13
find_all_orders . . . 14
find_all_orders_v . . . 15
find_B_sets . . . 16
find_cond_copula_specified . . . 17
find_interfering_v_from_B_sets . . . 18
fit_copulas . . . 19
has_interfering_vstrucs . . . 22
is_cond_copula_specified . . . 23
is_order_abiding_Bsets . . . 24
is_restrictedDAG . . . 25
logLik.PCBN . . . 26
new_PCBN . . . 27
path_hasConvergingConnections . . . 28
PCBN_PDF . . . 29
PCBN_sim . . . 30
plot.PCBN . . . 31
possible_candidates . . . 32
remove_CondInd . . . 33

Index **35**

`active_cycles` *Checks if a graph contains active cycles*

Description

Checks if a graph contains active cycles

Usage

```
active_cycles(DAG, early.stopping = FALSE)

has_active_cycles(DAG)

plot_active_cycles(DAG, active_cycles_list = NULL)
```

Arguments

DAG	Directed Acyclic
early.stopping	if TRUE, stop at the first active cycle that is found.
active_cycles_list	a list of active cycles as given by active_cycles. If this is NULL, the function active_cycles is run on DAG to find the active cycles to be displayed.

Value

active_cycles returns a list containing the active cycles. Each active cycle is a character vector of the name of the nodes involved in the active cycle. The first element of this vector is the converging node of the active cycle.

has_active_cycles returns TRUE if at least 1 active cycle is found. Otherwise, it returns FALSE.

plot_active_cycles is called for its side-effects only. It plots the active cycles if any, and else prints a message.

See Also

the helper functions path_hasConvergingConnections, path_hasChords that are used to find the active cycles.

is_restrictedDAG to check also whether the DAG contains interfering v-structures.

Examples

```
DAG = create_empty_DAG(4)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')
DAG = bnlearn::set.arc(DAG, 'U1', 'U4')
DAG = bnlearn::set.arc(DAG, 'U2', 'U4')
DAG = bnlearn::set.arc(DAG, 'U3', 'U4')

active_cycles(DAG)  # no active cycle

DAG = create_empty_DAG(4)
DAG = bnlearn::set.arc(DAG, 'U1', 'U2')
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U4')
DAG = bnlearn::set.arc(DAG, 'U3', 'U4')

active_cycles(DAG)  # 1 active cycle

DAG = create_empty_DAG(5)
DAG = bnlearn::set.arc(DAG, 'U1', 'U2')
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U4')
DAG = bnlearn::set.arc(DAG, 'U3', 'U4')
DAG = bnlearn::set.arc(DAG, 'U2', 'U5')
DAG = bnlearn::set.arc(DAG, 'U3', 'U5')
```

```
active_cycles(DAG)  # 2 active cycles
active_cycles(DAG, early.stopping = TRUE)  # The first active cycle

# Plotting the active cycles
plot_active_cycles(DAG)
# which is the same as
plot_active_cycles(DAG, active_cycles_list = active_cycles(DAG))

# We now fix the active cycles by adding the some arcs.
fixedDAG = fix_active_cycles(DAG)
# We can see that no active cycles is plotted anymore
plot_active_cycles(fixedDAG)
has_active_cycles(fixedDAG)
# This is because two edges have been added, as can be seen on:
plot(fixedDAG)
```

`B_sets_are_increasing` *Checks if the B-sets for a particular node form an increasing sequence.*

Description

Checks if the B-sets for a particular node form an increasing sequence.

Usage

```
B_sets_are_increasing(B_sets)
```

Arguments

`B_sets`	a boolean matrix with `(2 + length(children))` columns and `length(parents)` rows. They are assumed to be sorted in increasing order of row sums, i.e. by increasing order of set cardinality. Typically, this will be the output of `find_B_sets_v`.

Value

`TRUE` if the B-sets form an ordered sequence, otherwise returns `FALSE`.

Examples

```
B_sets = matrix(c(FALSE, FALSE, FALSE, FALSE,
                  TRUE , FALSE, FALSE, FALSE,
                  TRUE , TRUE , FALSE, FALSE,
                  TRUE , TRUE , TRUE ,  TRUE),
                nrow = 4, byrow = TRUE)

B_sets_are_increasing(B_sets)

B_sets = matrix(c(FALSE, FALSE, FALSE, FALSE,
```

```
                  TRUE , FALSE, TRUE , FALSE,
                  TRUE , TRUE , FALSE, FALSE,
                  TRUE , TRUE , TRUE ,  TRUE),
                nrow = 4, byrow = TRUE)

B_sets_are_increasing(B_sets)
```

`B_sets_cut_increments` *Find the decomposition of B-sets*

Description

Find the decomposition of B-sets

Usage

```
B_sets_cut_increments(B_sets)
```

Arguments

`B_sets`	matrix of B-sets, assumed to be increasing. This means `find_interfering_v_from_B_sets` should return `NULL` on this matrix. This can be the output of `find_B_sets_v` or of `B_sets_make_unique`.

Value

a list of vectors of characters. Each element of the list corresponds to one $DeltaBset = Bset[i]\ Bset[i-1]$.

Examples

```
B_sets = matrix(c(FALSE, FALSE, FALSE, FALSE,
                  TRUE , FALSE, FALSE, FALSE,
                  TRUE , TRUE , FALSE, FALSE,
                  TRUE , TRUE , TRUE ,  TRUE),
                nrow = 4, byrow = TRUE)

colnames(B_sets) <- c("U1", "U2", "U3", "U4")

B_sets_cut_increments(B_sets)
```

B_sets_make_unique *Compress a given collection of B-sets*

Description

Compress a given collection of B-sets

Usage

```
B_sets_make_unique(B_sets)
```

Arguments

B_sets	a boolean matrix with (2 + length(children)) columns and length(parents) rows. They are assumed to be sorted in increasing order of row sums, i.e. by increasing order of set cardinality. Typically, this will be the output of find_B_sets_v for some node v.

Value

a 'data.frame' made of the unique rows of 'B_sets'. An additional column 'nodes' is added at the start. It contains all the children of v corresponding to this B-set.

Examples

```
DAG = create_empty_DAG(5)
DAG = bnlearn::set.arc(DAG, 'U1', 'U4')
DAG = bnlearn::set.arc(DAG, 'U2', 'U4')
DAG = bnlearn::set.arc(DAG, 'U3', 'U4')
DAG = bnlearn::set.arc(DAG, 'U4', 'U5')
B_sets = find_B_sets_v(DAG, v = 'U4')

B_sets_make_unique(B_sets)
```

complete_and_check_orders *Complete an order and check whether these are valid orders on parents sets*

Description

Complete an order and check whether these are valid orders on parents sets

Usage

```
complete_and_check_orders(DAG, order_hash)
```

Arguments

DAG	the DAG
order_hash	the hashmaps of orders

Value

NULL. This function has only side-effects, and modifies order_hash. It stops if the orders are not valid orders on the parents sets.

Examples

```
DAG = create_empty_DAG(4)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')
DAG = bnlearn::set.arc(DAG, 'U3', 'U4')

order_hash = r2r::hashmap()
try({complete_and_check_orders(DAG, order_hash)})
# Error because the order of the parents on "U3" should be specified.

order_hash[['U3']] = c("U1", "U2")
complete_and_check_orders(DAG, order_hash)
r2r::keys(order_hash)
# We obtain "U3" and "U4" because they both have parents
```

---

compute_sample_margin *Computes a conditional margin during sampling*

Description

Computes a conditional margin during sampling

Usage

```
compute_sample_margin(
  object,
  data,
  v,
  cond_set,
  check_PCBN = TRUE,
  verbose = 1
)
```

Arguments

object	PCBN object to sample from. **This does not work if the PCBN does not abide by the B-sets. And in general, it does not work if the PCBN is outside of the class of restricted PCBNs.**
data	data frame of observations of size n
v	name of the node
cond_set	conditioning set. This is a vector containing the names of all the nodes in the conditioning set.
check_PCBN	check whether the given PCBN satisfies the restrictions. If this is set to FALSE, no checking is performed. This means that the error due to the a non-restricted PCBN object (if this is the case) will occur later in the computations (and may not be so clear - typically it is because of failing to find a given conditional copula).
verbose	if 0, don't print anything. If verbose >= 1, print information about the fitting procedure.

Value

a vector of size n of realizations $u_{i,v|cond\_set}$ for $i = 1, \ldots, n$.

Examples

```
DAG = create_empty_DAG(3)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')

order_hash = r2r::hashmap()
order_hash[['U3']] = c("U1", "U2")

fam = matrix(c(0, 0, 1,
               0, 0, 1,
               0, 0, 0), byrow = TRUE, ncol = 3)
tau = 0.2 * fam

my_PCBN = new_PCBN(
  DAG, order_hash,
  copula_mat = list(tau = tau, fam = fam))

# Initialize data frame
N = 100
nodes = bnlearn::nodes(my_PCBN$DAG)
data = data.frame(matrix(ncol = length(nodes), nrow = N))
colnames(data) <- nodes

data[, "U1"] = stats::runif(N)
data[, "U2"] = stats::runif(N)
u_1_given2 = compute_sample_margin(object = my_PCBN, data = data,
                                   v = "U1", cond_set = c("U2"))
```

```
identical(data[, "U1"], u_1_given2)
```

`create_empty_DAG` *Create empty DAG*

Description

This function creates a directed graph with a total of 'N_nodes' nodes and no arcs. The nodes are named 'U1', 'U2', etc.

Usage

```
create_empty_DAG(N_nodes)
```

Arguments

`N_nodes`	An integer equal to the number of nodes

Value

A `bnlearn` graph object with 'N_nodes' nodes and no arcs

See Also

[bnlearn::empty.graph()] which this function wraps.

Examples

```
create_empty_DAG(6)
create_empty_DAG(10)
```

`DAG_to_restrictedDAG` *Turns a general graph into a restricted graph.*

Description

Turns a general graph into a restricted graph.

Usage

```
DAG_to_restrictedDAG(DAG)

fix_active_cycles(DAG, active_cycles_list = NULL)

fix_interfering_vstructs(DAG, all_B_sets = NULL)
```

Arguments

DAG	a directed acyclic graph object, of class bn.
active_cycles_list, all_B_sets	respective outputs of the functions active_cycles and find_B_sets. If they are NULL, the respective function is called to compute them.

Value

Restricted DAG.

See Also

is_restrictedDAG to check whether a given DAG is restricted.

Examples

```
# DAG with an active cycle at node 5
DAG = create_empty_DAG(5)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U1', 'U2')
DAG = bnlearn::set.arc(DAG, 'U2', 'U4')
DAG = bnlearn::set.arc(DAG, 'U3', 'U5')
DAG = bnlearn::set.arc(DAG, 'U4', 'U5')

# Fixed graph has extra arcs 1 -> 5, 2 -> 5
fixed_DAG = DAG_to_restrictedDAG(DAG)


# DAG with an interfering v-structures node 3
DAG = create_empty_DAG(5)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U1', 'U4')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U5')
DAG = bnlearn::set.arc(DAG, 'U3', 'U4')
DAG = bnlearn::set.arc(DAG, 'U3', 'U5')

# Fixed graph has extra arc 1 -> 5
fixed_DAG = DAG_to_restrictedDAG(DAG)
```

---

default_envir *Fits the copula joining w and v given cond_set abiding by the conditional independencies of the graph*

Description

Fits the copula joining w and v given cond_set abiding by the conditional independencies of the graph

Usage

```
default_envir()

BiCopCondFit(
  data,
  DAG,
  v,
  w,
  cond_set,
  familyset,
  order_hash,
  e,
  verbose = 1,
  method
)

ComputeCondMargin(
  data,
  DAG,
  v,
  cond_set,
  familyset,
  order_hash,
  e,
  verbose = 1,
  method = method
)
```

Arguments

data	data frame
DAG	Directed Acylic Graph
v, w	nodes of the graph
cond_set	vector of nodes of DAG. They should all be parents of v. They should be ordered from the smallest to the biggest.
familyset	vector of copula families
order_hash	hashmap of parental orders
e	environment containing all the hashmaps
verbose	if 0, don't print anything. If verbose >= 1, print information about the fitting procedure.
method	indicates the estimation method ("mle" for maximum likelihood estimation or "itau" of inversion of Kendall's tau)

Value

default_envir returns an environment to be passed to BiCopCondFit or to ComputeCondMargin.

`BiCopCondFit` returns the fitted copula object of `v`, `w` given `cond_set`. `ComputeCondMargin` returns the fitted conditional margins of `v` given `cond_set`.

Both functions store all intermediary results in `e` to save computation time.

Examples

```
DAG = create_empty_DAG(3)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')

order_hash = r2r::hashmap()
order_hash[['U3']] = c("U1", "U2")

fam = matrix(c(0, 0, 1,
               0, 0, 1,
               0, 0, 0), byrow = TRUE, ncol = 3)

tau = 0.2 * fam

my_PCBN = new_PCBN(
  DAG, order_hash,
  copula_mat = list(tau = tau, fam = fam))

mydata = PCBN_sim(my_PCBN, N = 5)
e = default_envir()
ls(e)
C_13 = BiCopCondFit(data = mydata, DAG = DAG, v = "U1", w = "U3",
                    cond_set = c(), familyset = 1, order_hash = order_hash,
                    e = e, method = "mle")

C_23_1 = BiCopCondFit(data = mydata, DAG = DAG, v = "U2", w = "U3",
                      cond_set = "U1", familyset = 1, order_hash = order_hash,
                      e = e, method = "itau")

U_2_13 = ComputeCondMargin(data = mydata, DAG = DAG,
                           v = "U2", cond_set = c("U1", "U3"),
                           familyset = 1, order_hash = order_hash, e = e,
                           method = "mle")
```

`dsep_set` *D-separation of two nodes given a set in a DAG*

Description

D-separation of two nodes given a set in a DAG

Usage

```
dsep_set(DAG, X, Y, Z = NULL)
```

Arguments

DAG	Directed Acyclic Graph
X	node
Y	node
Z	set

Value

TRUE if the sets are d-separated and FALSE if not

Examples

```
DAG = create_empty_DAG(5)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')
DAG = bnlearn::set.arc(DAG, 'U1', 'U4')
DAG = bnlearn::set.arc(DAG, 'U2', 'U4')
DAG = bnlearn::set.arc(DAG, 'U3', 'U4')

dsep_set(DAG, 'U1', 'U5')
```

---

extend_orders *Fills in all possible orders for the next node for each possible order*

Description

Fills in all possible orders for the next node for each possible order

Usage

```
extend_orders(DAG, all_orders, node)
```

Arguments

DAG	Directed Acyclic Graph
all_orders	list of orders
node	node

Value

list of order hashmaps

Examples

```
DAG = create_empty_DAG(4)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')
DAG = bnlearn::set.arc(DAG, 'U1', 'U4')
DAG = bnlearn::set.arc(DAG, 'U2', 'U4')
DAG = bnlearn::set.arc(DAG, 'U3', 'U4')

# Start with empty order
order_hash = r2r::hashmap()

all_orders_3 = find_all_orders_v(DAG, v = "U3", order_hash = order_hash)
print(all_orders_3)

# Two possible choices for node 3, let's use the first
order_hash[['U3']] = all_orders_3[[1]]

extended_orders = extend_orders(DAG, list(order_hash), node = 'U4')
length(extended_orders)
# We can extend this order in 4 ways:
for (i in 1:length(extended_orders)){
  print(extended_orders[[i]][['U4']])
}
# We never pick U2 and U3 first, because their copula is not specified
```

`find_all_orders` *Finds all possible copula assignments given a DAG*

Description

Finds all possible copula assignments given a DAG

Usage

```
find_all_orders(DAG)
```

Arguments

`DAG`	Directed Acyclic Graph

Value

a list of hashmaps containing the possible orders

Examples

```
DAG = create_empty_DAG(4)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')
DAG = bnlearn::set.arc(DAG, 'U1', 'U4')
DAG = bnlearn::set.arc(DAG, 'U2', 'U4')
DAG = bnlearn::set.arc(DAG, 'U3', 'U4')
all_orders = find_all_orders(DAG)
length(all_orders)
# 8 orders
for (i in 1:length(all_orders)){
  cat("Order ", i, ": \n")
  cat("U3:", all_orders[[i]][['U3']])
  cat(" ; U4:", all_orders[[i]][['U4']], "\n")
}
```

`find_all_orders_v` *Finds all possible orders of node v given previous copula assignments*

Description

Finds all possible orders of node v given previous copula assignments

Usage

```
find_all_orders_v(DAG, v, order_hash)
```

Arguments

`DAG`	Directed Acyclic Graph
`v`	node
`order_hash`	hashmap of orders

Value

list of vectors containing all possible orders for v

Examples

```
DAG = create_empty_DAG(5)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')
DAG = bnlearn::set.arc(DAG, 'U1', 'U4')
DAG = bnlearn::set.arc(DAG, 'U2', 'U4')
DAG = bnlearn::set.arc(DAG, 'U3', 'U4')
DAG = bnlearn::set.arc(DAG, 'U1', 'U5')
DAG = bnlearn::set.arc(DAG, 'U2', 'U5')
```

```
DAG = bnlearn::set.arc(DAG, 'U3', 'U5')
DAG = bnlearn::set.arc(DAG, 'U4', 'U5')

# Start with empty order
order_hash = r2r::hashmap()

all_orders_3 = find_all_orders_v(DAG, v = "U3", order_hash = order_hash)
print(all_orders_3)

# Two possible choices for node 3, let's use the first
order_hash[['U3']] = all_orders_3[[1]]

all_orders_4 = find_all_orders_v(DAG, v = "U4", order_hash = order_hash)
print(all_orders_4)

# Four possible choices for node 4, let's use the third
order_hash[['U4']] = all_orders_4[[3]]

all_orders_5 = find_all_orders_v(DAG, v = "U5", order_hash = order_hash)
print(all_orders_5)

# Eight possible orders for node 5; let's use the fourth
order_hash[['U5']] = all_orders_5[[4]]
```

`find_B_sets` *Find all the B-sets of a given DAG*

Description

Find all the B-sets of a given DAG

Find the B sets for a given node v

Usage

```
find_B_sets(DAG)

find_B_sets_v(DAG, v)
```

Arguments

`DAG`	A bnlearn graph object
`v`	node at which we want to find the B-sets

Value

`find_B_sets` returns a list with three elements

- `B_sets` list of B-sets matrices for each node;
- `has_interfering_vstrucs` a boolean specifying if the graph contains interfering v-structures or not;
- `nodes_with_inter_vs` a list containing nodes forming the interfering v-structures.

`find_B_sets_v` returns a boolean matrix with `(2 + length(children))` columns and `length(parents)` rows. This is also true if `length(parents) == 0` and `length(parents) == 1`.

Examples

```
DAG = create_empty_DAG(6)
DAG = bnlearn::set.arc(DAG, 'U1', 'U5')
DAG = bnlearn::set.arc(DAG, 'U2', 'U5')
DAG = bnlearn::set.arc(DAG, 'U3', 'U5')
DAG = bnlearn::set.arc(DAG, 'U4', 'U5')

DAG = bnlearn::set.arc(DAG, 'U1', 'U6')
DAG = bnlearn::set.arc(DAG, 'U2', 'U6')
DAG = bnlearn::set.arc(DAG, 'U5', 'U6')

find_B_sets_v(DAG, v = 'U5')
B_sets = find_B_sets(DAG)
B_sets$B_sets
```

---

find_cond_copula_specified

Find among parents of a node, the one that has a conditional copula specified

---

Description

Find among parents of a node, the one that has a conditional copula specified

Usage

```
find_cond_copula_specified(DAG, order_hash, v, cond)
```

Arguments

`DAG`	Directed Acyclic Graph object corresponding to the model
`order_hash`	hashmap of orders of the parental sets
`v`	node in DAG
`cond`	vector of nodes in DAG. This must not be empty. It is assumed that conditionally independent nodes have already been removed by the function `remove_CondInd`. It is assumed to have been already sorted.

Value

a list with

- a node w such that the conditional copula $C_{w,v|cond[-v]}$ has been specified in the model. If no such node can be found, an error message is raised.
- the set cond[-v]

Examples

```
DAG = create_empty_DAG(3)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')

order_hash = r2r::hashmap()
order_hash[['U3']] = c("U1", "U2")

find_cond_copula_specified(DAG = DAG, order_hash = order_hash,
                           v = "U3", cond = c("U1"))
# returns "U1" because the copula c_{1,3} is known

find_cond_copula_specified(DAG = DAG, order_hash = order_hash,
                           v = "U3", cond = c("U1", "U2"))
# returns "U2" because the copula c_{2,3|1} is known
```

---

find_interfering_v_from_B_sets

Find all interfering v-structures for a given collection of B-sets

---

Description

Find all interfering v-structures for a given collection of B-sets

Usage

```
find_interfering_v_from_B_sets(B_sets)
```

Arguments

B_sets	a boolean matrix with (2 + length(children)) columns and length(parents) rows. They are assumed to be sorted in increasing order of row sums, i.e. by increasing order of set cardinality. Typically, this will be the output of find_B_sets_v for some node v.

Value

NULL if there is no interfering v-structures. Else, it returns a data.frame with 4 columns

- A: a list of children of v
- B: a list of children of v, disjoint from A
- parents.A.but.not.parents.B: a list of common parents of nodes of A, that are not parents of nodes of B
- parents.B.but.not.parents.A: a list of common parents of nodes of B, that are not parents of nodes of A

Each line correspond to 1 interfering v-structure.

Examples

```
DAG = create_empty_DAG(7)
DAG = bnlearn::set.arc(DAG, 'U1', 'U5')
DAG = bnlearn::set.arc(DAG, 'U2', 'U5')
DAG = bnlearn::set.arc(DAG, 'U3', 'U5')
DAG = bnlearn::set.arc(DAG, 'U4', 'U5')

DAG = bnlearn::set.arc(DAG, 'U1', 'U6')
DAG = bnlearn::set.arc(DAG, 'U5', 'U6')
DAG = bnlearn::set.arc(DAG, 'U2', 'U7')
DAG = bnlearn::set.arc(DAG, 'U5', 'U7')

B_sets = find_B_sets_v(DAG, v = 'U5')
find_interfering_v_from_B_sets(B_sets)

# Adding the missing arc
DAG = bnlearn::set.arc(DAG, 'U1', 'U7')
# Now no interfering v-structure
B_sets = find_B_sets_v(DAG, v = 'U5')
find_interfering_v_from_B_sets(B_sets)
```

---

fit_copulas *Fit the copulas of a PCBN given data*

Description

Fit the copulas of a PCBN given data

Fit all possible orders given a DAG

Usage

```
fit_copulas(
  data,
  DAG,
  order_hash,
  familyset = c(1, 3, 4, 5, 6),
  familyMatrix = NULL,
  e,
  verbose = 1,
  method = "mle"
)

fit_all_orders(
  data,
  DAG,
  familyset = c(1, 3, 4, 5, 6),
  e,
  score_metric = "BIC",
  verbose = 1
)
```

Arguments

data	data frame
DAG	Directed Acyclic Graph
order_hash	hashmap of parental orders
familyset	vector of copula families
familyMatrix	matrix of families, with named rows and columns. This should be used if the copula families are known/fixed. This overrides familyset.
e	environment containing all the hashmaps
verbose	if 0, don't print anything. If verbose >= 1, print information about the simulation procedure.
method	indicates the estimation method ("mle" for maximum likelihood estimation or "itau" of inversion of Kendall's tau).
score_metric	name of the metric used to choose the best order. Possible choices are logLik, AIC and BIC.

Value

fit_copulas returns the fitted PCBN, with an additional element called metrics which is a named vector of length 3 with names c("logLik", "BIC", "AIC"), where $AIC = -2 * logLik + 2 * nparameters$ and $BIC = -2 * logLik + log(n) * nparameters$, for a sample size n and nparameters is the number of parameters.

fit_all_orders returns a list containing:

- best_fit the PCBN which is the best according to the metric score_metric.

- `fitted_list` the list of all fitted PCBNs.
- `metrics` the matrix of metrics (logLik, BIC, AIC). Each row `i` of this matrix corresponds to a PCBN with a different set of parents' orderings, and corresponds to element `i` of `fitted_list`.

See Also

`BiCopCondFit` which this function wraps.

Examples

```
DAG = create_empty_DAG(3)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')

order_hash = r2r::hashmap()
order_hash[['U3']] = c("U1", "U2")

fam = matrix(c(0, 0, 1,
               0, 0, 1,
               0, 0, 0), byrow = TRUE, ncol = 3)

tau = 0.2 * fam

my_PCBN = new_PCBN(
  DAG, order_hash,
  copula_mat = list(tau = tau, fam = fam))

mydata = PCBN_sim(my_PCBN, N = 5)
e = default_envir()

result = fit_copulas(data = mydata, DAG = DAG,
                     order_hash = order_hash,
                     familyset = 1, e = e)

result_all_orders = fit_all_orders(data = mydata, DAG = DAG,
                                   familyset = 1, e = e)

# The two fitted PCBNs are:
print(result_all_orders$fitted_list[[1]])
print(result_all_orders$fitted_list[[2]])
# and the metrics matrix is:
print(result_all_orders$metrics)

# The PCBN corresponding to the true order U1 < U2 is usually better
# than the second one. This Will be more clear with a bigger sample size.
```

has_interfering_vstrucs *Checks if graph has interfering v-structures*

Description

Checks if graph has interfering v-structures

Usage

```
has_interfering_vstrucs(DAG, verbose = 0)
```

Arguments

DAG	Directed Acyclic Graph
verbose	if verbose is 0, do not print anything. If verbose is positive, print the name of the first node at which the interfering v-structure is found.

Value

TRUE if graph contains (at least) an interfering v-structure, and FALSE if it does not contain any interfering v-structure.

Examples

```
DAG = create_empty_DAG(5)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')

DAG = bnlearn::set.arc(DAG, 'U1', 'U4')
DAG = bnlearn::set.arc(DAG, 'U3', 'U4')
DAG = bnlearn::set.arc(DAG, 'U2', 'U5')
DAG = bnlearn::set.arc(DAG, 'U3', 'U5')

# There is one interfering v-structure
has_interfering_vstrucs(DAG, verbose = 1)

DAG = bnlearn::set.arc(DAG, 'U1', 'U5')
# Now no interfering v-structure
has_interfering_vstrucs(DAG)
```

`is_cond_copula_specified` *Checks if a given (conditional) copula has already been specified*

Description

Checks if a given (conditional) copula has already been specified

Usage

```
is_cond_copula_specified(DAG, order_hash, w, v, cond)
```

Arguments

`DAG`	Directed Acyclic Graph object corresponding to the model
`order_hash`	hashmap of orders of the parental sets
`w`	node in DAG
`v`	node in DAG
`cond`	vector of nodes in DAG. It is assumed to have been already sorted.

Value

`TRUE` if the conditional copula $C_{w,v|cond}$ has been specified in the model

Examples

```
DAG = create_empty_DAG(3)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')

order_hash = r2r::hashmap()
order_hash[['U3']] = c("U1", "U2")

is_cond_copula_specified(DAG = DAG, order_hash = order_hash,
                    w = "U1", v = "U3", cond = c())
# returns TRUE because the copula c_{1,3} is known

is_cond_copula_specified(DAG = DAG, order_hash = order_hash,
                    w = "U2", v = "U3", cond = c())
# returns FALSE because the copula c_{2,3} is not known

is_cond_copula_specified(DAG = DAG, order_hash = order_hash,
                    w = "U2", v = "U3", cond = c("U1"))
# returns TRUE because the copula c_{2,3 | 1} is known
```

`is_order_abiding_Bsets` *Check whether a certain order abides by the B-sets*

Description

Check whether a certain order abides by the B-sets

Usage

```
is_order_abiding_Bsets(DAG, order_hash)

is_order_abiding_Bsets_v(B_sets, orderParents)
```

Arguments

`DAG`	the considered DAG
`order_hash`	the hashmaps of parents ordering
`B_sets`	matrix of B-sets, assumed to be increasing. This can be the output of `find_B_sets_v` or of `B_sets_make_unique`.
`orderParents`	a vector of characters, interpreted as the ordered parents

Value

It returns 'TRUE' if the order abides by the B-sets, and 'FALSE' else.

Examples

```
DAG = create_empty_DAG(4)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')
DAG = bnlearn::set.arc(DAG, 'U3', 'U4')
DAG = bnlearn::set.arc(DAG, 'U1', 'U4')

order_hash = r2r::hashmap()
order_hash[['U3']] = c("U1", "U2")
order_hash[['U4']] = c("U1", "U3")
is_order_abiding_Bsets(DAG, order_hash)
order_hash[['U3']] = c("U2", "U1")
is_order_abiding_Bsets(DAG, order_hash)
```

`is_restrictedDAG` *Does a DAG satisfy the restrictions of no active cycle and no interfering v-structures*

Description

This functions checks whether the DAG is restricted, i.e. whether it has no active cycles nor any interfering v-structures.

Usage

```
is_restrictedDAG(DAG, verbose = 2, check_both = TRUE)
```

Arguments

`DAG`	the DAG object
`verbose`	if `verbose` is 2, details are printed. If `verbose` is 1, details are printed only if an active cycle or an interfering v-structure is found. If `verbose` is `0` the function does not print anything and only returns `TRUE` or `FALSE`.
`check_both`	if `TRUE`, both v-structures and active cycles are checked anyway. If `FALSE`, the function stops early if it already found any v-structures.

Value

`TRUE` if the PCBN satisfies both restrictions. `FALSE` if at least one of the restrictions is not satisfies.

See Also

`DAG_to_restrictedDAG` for one way of making the DAG to be restricted if it is not the case.

`active_cycles` to find all active cycles. `has_interfering_vstrucs` to check only for interfering v-structures.

Examples

```
DAG = create_empty_DAG(4)
DAG = bnlearn::set.arc(DAG, 'U1', 'U2')
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U4')
DAG = bnlearn::set.arc(DAG, 'U3', 'U4')

is_restrictedDAG(DAG)  # 1 active cycle
```

`logLik.PCBN` *Log-likelihood of a PCBN object*

Description

This function computes the log-likelihood of the PCBN model given a dataset of i.i.d. observations uniformly (or approximatively uniformly) distributed on $[0, 1]$. This is the same as the logarithm of the density of the PCBN at the observations.

Usage

```
## S3 method for class 'PCBN'
logLik(object, data_uniform, ...)
```

Arguments

`object`	the PCBN object
`data_uniform`	the dataset for which the log-likelihood is computed. It must have already been standardized to uniform margins.
`...`	other arguments, ignored for the moment

Value

the log-likelihood of the PCBN model for the given dataset

Examples

```
DAG = create_empty_DAG(3)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')

order_hash = r2r::hashmap()
order_hash[['U3']] = c("U1", "U2")

fam = matrix(c(0, 0, 1,
               0, 0, 1,
               0, 0, 0), byrow = TRUE, ncol = 3)
tau = 0.2 * fam

my_PCBN = new_PCBN(
  DAG, order_hash,
  copula_mat = list(tau = tau, fam = fam))

mydata = PCBN_sim(my_PCBN, N = 10)

logLik(my_PCBN, mydata)
```

new_PCBN *Initializes PCBN class*

Description

Initializes PCBN class

Usage

```
new_PCBN(DAG, order_hash, copula_mat, verbose = 0)
```

Arguments

DAG	the corresponding DAG (a 'bn' object)
order_hash	a hashmap of character vectors Each character vector corresponds to the order of the parents for the given node.
copula_mat	a list with at least two components: • fam the matrix of families • tau the matrix of Kendall's tau They both should be matrices of size d * d, where d is the number of nodes in the graph DAG.
verbose	If 0, no message is printed. If 1 (recommended), information is printed during the checking of the PCBN.

Value

the new PCBN

Examples

```
DAG = create_empty_DAG(3)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')

order_hash = r2r::hashmap()
order_hash[['U3']] = c("U1", "U2")

fam = matrix(c(0, 0, 1,
               0, 0, 1,
               0, 0, 0), byrow = TRUE, ncol = 3)
tau = 0.2 * fam

my_PCBN = new_PCBN(
  DAG, order_hash,
  copula_mat = list(tau = tau, fam = fam))
```

path_hasConvergingConnections *Checks a path for converging connections and chords.*

Description

Checks a path for converging connections and chords.

Usage

```
path_hasConvergingConnections(DAG, path)

path_hasChords(DAG, path)
```

Arguments

DAG	Directed Acyclic Graph.
path	character vector of nodes in the DAG forming a trail.

Value

path_hasConvergingConnections returns TRUE if the path contains a converging connection.

path_hasChords returns TRUE if the path contains a chord.

See Also

active_cycles which uses these two functions.

Examples

```
DAG = create_empty_DAG(4)
DAG = bnlearn::set.arc(DAG, 'U1', 'U2')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')
DAG = bnlearn::set.arc(DAG, 'U3', 'U4')
path_hasConvergingConnections(DAG, c('U1', 'U2', 'U3', 'U4') ) # FALSE
path_hasChords(DAG, c('U1', 'U2', 'U3', 'U4') ) # FALSE

DAG = bnlearn::set.arc(DAG, 'U1', 'U4')
path_hasConvergingConnections(DAG, c('U1', 'U2', 'U3', 'U4') ) # FALSE
path_hasChords(DAG, c('U1', 'U2', 'U3', 'U4') ) # TRUE: has a chord

DAG = create_empty_DAG(4)
DAG = bnlearn::set.arc(DAG, 'U1', 'U2')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')
DAG = bnlearn::set.arc(DAG, 'U4', 'U3')
path_hasConvergingConnections(DAG, c('U1', 'U2', 'U3', 'U4') )
# TRUE: has a converging connection
path_hasChords(DAG, c('U1', 'U2', 'U3', 'U4') ) # FALSE
```

PCBN_PDF *PDF of a PCBN model*

Description

This function computes the Probability Density Function of a PCBN model.

Usage

```
PCBN_PDF(PCBN, newdata)
```

Arguments

PCBN	PCBN object
newdata	new data on which the PDF should be computed

Details

This is a wrapper to logLik.PCBN.

Value

the probability density at newdata.

Examples

```
DAG = create_empty_DAG(3)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')

order_hash = r2r::hashmap()
order_hash[['U3']] = c("U1", "U2")

fam = matrix(c(0, 0, 1,
               0, 0, 1,
               0, 0, 0), byrow = TRUE, ncol = 3)
tau = 0.2 * fam

my_PCBN = new_PCBN(
  DAG, order_hash,
  copula_mat = list(tau = tau, fam = fam))

mydata = PCBN_sim(my_PCBN, N = 10)

PCBN_PDF(my_PCBN, mydata)
```

PCBN_sim *Simulate data from a specified PCBN*

Description

Simulate data from a specified PCBN

Usage

```
PCBN_sim(object, N, check_PCBN = TRUE, verbose = 1)
```

Arguments

object	PCBN object to simulate from. **This does not work if the PCBN does not abide by the B-sets. And in general, it does not work if the PCBN is outside of the class of restricted PCBNs.**
N	sample size
check_PCBN	check whether the given PCBN satisfies the restrictions. If this is set to FALSE, no checking is performed. This means that the error due to the a non-restricted PCBN object (if this is the case) will occur later in the computations (and may not be so clear - typically it is because of failing to find a given conditional copula). Nevertheless, even with 'check_PCBN = TRUE' if could be that some error happen later if the parental orderings are not compatible with each other.
verbose	if 0, don't print anything. If verbose >= 1, print information about the simulation procedure.

Value

a data frame of N samples

Examples

```
DAG = create_empty_DAG(3)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')

order_hash = r2r::hashmap()
order_hash[['U3']] = c("U1", "U2")

fam = matrix(c(0, 0, 1,
               0, 0, 1,
               0, 0, 0), byrow = TRUE, ncol = 3)

tau = 0.2 * fam

my_PCBN = new_PCBN(
  DAG, order_hash,
  copula_mat = list(tau = tau, fam = fam))
```

```
mydata = PCBN_sim(my_PCBN, N = 5)
```

`plot.PCBN` *Print and plot PCBN objects*

Description

Print and plot PCBN objects

Usage

```
## S3 method for class 'PCBN'
plot(x, ...)

## S3 method for class 'PCBN'
print(x, print.orders = "non-empty", ...)
```

Arguments

`x`	PCBN object
`...`	other arguments, unused
`print.orders`	if `"all"`, print all orders. If `"non-empty"`, this only prints the non-empty ones.

Value

No return value, both functions are called for side effects only.

Examples

```
DAG = create_empty_DAG(3)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')

order_hash = r2r::hashmap()
order_hash[['U3']] = c("U1", "U2")

fam = matrix(c(0, 0, 1,
               0, 0, 1,
               0, 0, 0), byrow = TRUE, ncol = 3)
tau = 0.2 * fam

my_PCBN = new_PCBN(
  DAG, order_hash,
  copula_mat = list(tau = tau, fam = fam))
print(my_PCBN)
plot(my_PCBN)
```

possible_candidates *Possible candidates to be added to a partial order*

Description

When given a partial order of a PCBN, one can complete it by adding one of the parents' node to the partial order. Some nodes can be added; they are then called "possible candidates".

Usage

```
possible_candidates(DAG, v, order_v, order_hash, B_minus_O)

possible_candidate_incoming_arc(DAG, w, v, order_v, order_hash)

possible_candidate_outgoing_arc(DAG, w, v, order_v, order_hash)
```

Arguments

DAG	Directed Acyclic Graph.
order_v	partial order for node v.
order_hash	hashmap of parental orders
B_minus_O	this is the current B-set, without the elements of order_v, i.e. this is the set of elements that could be considered possible candidates.
w, v	nodes in DAG. w is assumed to be a parent of v.

Details

possible_candidate_incoming_arc returns a node o such w is a parent of o, and w can be used as an incoming arc to v by the node o. If no such o can be found, w cannot be used as a potential candidate for the order of v by incoming arc. Then, the function possible_candidate_incoming_arc returns NULL.

In the same way, possible_candidate_outgoing_arc returns a node o such o is a parent of w, and w can be used as an outgoing arc to v by the node o.

Value

possible_candidates returns a vector of possible candidates, potentially empty. Both possible_candidate_incoming_ar
and possible_candidate_outgoing_arc return either a node o, or NULL if they could not find such a node.

See Also

dsep_set for checking whether two sets of nodes are d-separated by another set. find_B_sets to find the B-sets.

Examples

```
DAG = create_empty_DAG(4)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')
DAG = bnlearn::set.arc(DAG, 'U1', 'U4')
DAG = bnlearn::set.arc(DAG, 'U2', 'U4')
DAG = bnlearn::set.arc(DAG, 'U3', 'U4')

order_hash = r2r::hashmap()
order_hash[['U3']] = c("U1", "U2")

# Node of interest
v = "U4"

# If we start by 1, then the arc 1 -> 3 cannot be used as an incoming arc
# (it is actually an outgoing arc)
possible_candidate_incoming_arc(
  DAG = DAG, w = "U3", v = v, order_v = c("U1"), order_hash = order_hash)
possible_candidate_outgoing_arc(
  DAG = DAG, w = "U3", v = v, order_v = c("U1"), order_hash = order_hash)
possible_candidates(
  DAG = DAG, v = v, order_v = c("U1"), order_hash = order_hash, B_minus_O = "U2")
```

`remove_CondInd` *Remove elements from a conditioning set by using conditional independence*

Description

Remove elements from a conditioning set by using conditional independence

Usage

```
remove_CondInd(DAG, node, cond_set)
```

Arguments

`DAG`	Directed Acyclic Graph
`node`	node
`cond_set`	vector of nodes in conditioning set

Value

a vector containing the nodes that cannot be removed from the conditioning set.

Examples

```
DAG = create_empty_DAG(3)
DAG = bnlearn::set.arc(DAG, 'U1', 'U3')
DAG = bnlearn::set.arc(DAG, 'U2', 'U3')

remove_CondInd(DAG = DAG, node = "U1", cond_set = c("U2"))
remove_CondInd(DAG = DAG, node = "U3", cond_set = c("U1"))
```

Index

active_cycles, 2, *10*, *25*, *28*

B_sets_are_increasing, 4
B_sets_cut_increments, 5
B_sets_make_unique, *5*, 6, *24*
BiCopCondFit, *21*
BiCopCondFit *(*default_envir*)*, 10

complete_and_check_orders, 6
compute_sample_margin, 7
ComputeCondMargin *(*default_envir*)*, 10
create_empty_DAG, 9

DAG_to_restrictedDAG, 9, *25*
default_envir, 10
dsep_set, 12, *32*

extend_orders, 13

find_all_orders, 14
find_all_orders_v, 15
find_B_sets, *10*, 16, *32*
find_B_sets_v, *5*, *24*
find_B_sets_v *(*find_B_sets*)*, 16
find_cond_copula_specified, 17
find_interfering_v_from_B_sets, *5*, 18
fit_all_orders *(*fit_copulas*)*, 19
fit_copulas, 19
fix_active_cycles
        *(*DAG_to_restrictedDAG*)*, 9
fix_interfering_vstructs
        *(*DAG_to_restrictedDAG*)*, 9

has_active_cycles *(*active_cycles*)*, 2
has_interfering_vstrucs, 22, *25*

is_cond_copula_specified, 23
is_order_abiding_Bsets, 24
is_order_abiding_Bsets_v
        *(*is_order_abiding_Bsets*)*, 24
is_restrictedDAG, *3*, *10*, 25

logLik.PCBN, 26, *29*

new_PCBN, 27

path_hasChords, *3*
path_hasChords
        *(*path_hasConvergingConnections*)*, 28
path_hasConvergingConnections, *3*, 28
PCBN_PDF, 29
PCBN_sim, 30
plot.PCBN, 31
plot_active_cycles *(*active_cycles*)*, 2
possible_candidate_incoming_arc
        *(*possible_candidates*)*, 32
possible_candidate_outgoing_arc
        *(*possible_candidates*)*, 32
possible_candidates, 32
print.PCBN *(*plot.PCBN*)*, 31

remove_CondInd, *17*, 33