Package ‘pcalg’
August 16, 2020

Version     2.6-11
Date         2020-08-12
Title        Methods for Graphical Models and Causal Inference
Description  Functions for causal structure
              learning and causal inference using graphical models. The main algorithms
              for causal structure learning are PC (for observational data without hidden
              variables), FCI and RFCI (for observational data with hidden variables),
              and GIES (for a mix of data from observational studies
              (i.e. observational data) and data from experiments
              involving interventions (i.e. interventional data) without hidden
              variables). For causal inference the IDA algorithm, the Generalized
              Backdoor Criterion (GBC), the Generalized Adjustment Criterion (GAC)
              and some related functions are implemented. Functions for incorporating
              background knowledge are provided.
Maintainer    Markus Kalisch <kalisch@stat.math.ethz.ch>
Author        Markus Kalisch [aut, cre],
               Alain Hauser [aut],
               Martin Maechler [aut],
               Diego Colombo [ctb],
               Doris Entner [ctb],
               Patrik Hoyer [ctb],
               Antti Hyttinen [ctb],
               Jonas Peters [ctb],
               Nicoletta Andri [ctb],
               Emilia Perkovic [ctb],
               Preetam Nandy [ctb],
               Philipp Ruetimann [ctb],
               Daniel Stekhoven [ctb],
               Manuel Schuerch [ctb],
               Marco Eigenmann [ctb]
Depends       R (>= 3.0.2)
LinkingTo     Rcpp (>= 0.11.0), RcppArmadillo, BH
Imports       stats, graphics, utils, methods, abind, graph, RBGL, igraph,
              ggm, corpcor, robustbase, vcd, Rcpp, bdsmatrix, sfsmisc (>=
Suggests MASS, Matrix, Rgraphviz, mvtnorm, huge, ggplot2, dagitty

ByteCompile yes

NeedsCompilation yes

Encoding UTF-8

License GPL (>= 2)

URL http://pcalg.r-forge.r-project.org/

RoxygenNote 6.1.1

Repository CRAN

Date/Publication 2020-08-16 16:00:02 UTC

R topics documented:

addBgKnowledge ........................................... 4
adjustment ................................................. 5
ages ............................................................. 7
amatType ....................................................... 11
backdoor ....................................................... 14
beta.special .................................................. 18
beta.special.pcObj ......................................... 19
binCItest ..................................................... 20
checkTriple .................................................. 22
compareGraphs .............................................. 24
condIndFisherZ .............................................. 26
corGraph ...................................................... 28
dag2cpdag .................................................... 29
dag2essgraph ................................................. 30
dag2pag ......................................................... 32
disCItest ...................................................... 34
dreach ......................................................... 36
dsep ........................................................... 37
dsepTest ....................................................... 38
EssGraph-class ............................................... 39
fci ............................................................. 41
fciAlgo-class ................................................ 46
fciPlus ......................................................... 48
find.unsh.triple ........................................... 50
gac ............................................................ 51
gAlgo-class .................................................. 55
GaussL0penIntScore-class ................................. 56
GaussL0penObsScore-class ................................. 58
GaussParDAG-class ......................................... 60
gds ........................................................... 62
ges ........................................................... 64
<table>
<thead>
<tr>
<th>R topics documented</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>getGraph</td>
<td>68</td>
</tr>
<tr>
<td>getNextSet</td>
<td>69</td>
</tr>
<tr>
<td>gies</td>
<td>70</td>
</tr>
<tr>
<td>gmB</td>
<td>73</td>
</tr>
<tr>
<td>gmD</td>
<td>74</td>
</tr>
<tr>
<td>gmG</td>
<td>75</td>
</tr>
<tr>
<td>gmI</td>
<td>76</td>
</tr>
<tr>
<td>gmInt</td>
<td>77</td>
</tr>
<tr>
<td>gmL</td>
<td>79</td>
</tr>
<tr>
<td>ida</td>
<td>80</td>
</tr>
<tr>
<td>idaFast</td>
<td>85</td>
</tr>
<tr>
<td>iplotPC</td>
<td>86</td>
</tr>
<tr>
<td>isValidGraph</td>
<td>88</td>
</tr>
<tr>
<td>jointIda</td>
<td>89</td>
</tr>
<tr>
<td>legal.path</td>
<td>93</td>
</tr>
<tr>
<td>LINGAM</td>
<td>94</td>
</tr>
<tr>
<td>mat2targets</td>
<td>96</td>
</tr>
<tr>
<td>mcor</td>
<td>98</td>
</tr>
<tr>
<td>opt.target</td>
<td>99</td>
</tr>
<tr>
<td>optAdjSet</td>
<td>100</td>
</tr>
<tr>
<td>pag2mag</td>
<td>102</td>
</tr>
<tr>
<td>ParDAG-class</td>
<td>104</td>
</tr>
<tr>
<td>pc</td>
<td>105</td>
</tr>
<tr>
<td>pc.cons.intern</td>
<td>111</td>
</tr>
<tr>
<td>pcalg2dagitty</td>
<td>113</td>
</tr>
<tr>
<td>pcAlgo</td>
<td>114</td>
</tr>
<tr>
<td>pcAlgo-class</td>
<td>116</td>
</tr>
<tr>
<td>pcorOrder</td>
<td>118</td>
</tr>
<tr>
<td>pcSelect</td>
<td>119</td>
</tr>
<tr>
<td>pcSelect.presel</td>
<td>121</td>
</tr>
<tr>
<td>pdag2allDags</td>
<td>122</td>
</tr>
<tr>
<td>pdag2dag</td>
<td>124</td>
</tr>
<tr>
<td>pdsep</td>
<td>125</td>
</tr>
<tr>
<td>plotAG</td>
<td>128</td>
</tr>
<tr>
<td>plotSG</td>
<td>128</td>
</tr>
<tr>
<td>possAn</td>
<td>130</td>
</tr>
<tr>
<td>possDe</td>
<td>131</td>
</tr>
<tr>
<td>possibleDe</td>
<td>132</td>
</tr>
<tr>
<td>qreach</td>
<td>133</td>
</tr>
<tr>
<td>r.gauss.pardag</td>
<td>134</td>
</tr>
<tr>
<td>randDAG</td>
<td>136</td>
</tr>
<tr>
<td>randomDAG</td>
<td>138</td>
</tr>
<tr>
<td>rfcI</td>
<td>140</td>
</tr>
<tr>
<td>rmvDAG</td>
<td>144</td>
</tr>
<tr>
<td>rmvnorm.ivenet</td>
<td>145</td>
</tr>
<tr>
<td>Score-class</td>
<td>146</td>
</tr>
<tr>
<td>shd</td>
<td>148</td>
</tr>
<tr>
<td>showAmat</td>
<td>149</td>
</tr>
</tbody>
</table>
addBgKnowledge

Add background knowledge to a CPDAG or PDAG

Description
Add background knowledge x -> y to an adjacency matrix and complete the orientation rules from Meek (1995).

Usage
addBgKnowledge(gInput, x = c(), y = c(), verbose = FALSE, checkInput = TRUE)

Arguments
- gInput: graphNEL object or adjacency matrix of type amat.cpdag (see amatType)
- x, y: node labels of x or y in the adjacency matrix. x and y can be vectors representing several nodes (see details below).
- verbose: If TRUE, detailed output is provided.
- checkInput: If TRUE, the input adjacency matrix is carefully checked to see if it is a valid graph using function isValidGraph

Details
If the input is a graphNEL object, it will be converted into an adjacency matrix of type amat.cpdag. If x and y are given and if amat[y,x] != 0, this function adds orientation x -> y to the adjacency matrix amat and completes the orientation rules from Meek (1995). If x and y are not specified (or empty vectors) this function simply completes the orientation rules from Meek (1995). If x and y are vectors of length k, k>1, this function tries to add x[i] -> y[i] to the adjacency matrix amat and complete the orientation rules from Meek (1995) for every i in 1,...,k (see Algorithm 1 in Perkovic et. al, 2017).

Value
An adjacency matrix of type amat.cpdag of the maximally oriented pdag with added background knowledge x -> y or NULL, if the background knowledge is not consistent with any DAG represented by the PDAG with the adjacency matrix amat.
adjustment

Compute adjustment sets for covariate adjustment.

Description

This function is a wrapper for convenience to the function `adjustmentSet` from package `dagitty`.

Usage

```r
adjustment(amat, amat.type, x, y, set.type)
```
Arguments

amat      adjacency matrix of type amat.cpdag or amat.pag.
amat.type string specifying the type of graph of the adjacency matrix amat. It can be a
            DAG (type="dag"), a CPDAG (type="cpdag") or a maximally oriented PDAG
            (type="pdag") from Meek (1995); then the type of adjacency matrix is assumed
            to be amat.cpdag. It can also be a MAG (type = "mag") or a PAG (type="pag");
            then the type of the adjacency matrix is assumed to be amat.pag.
            
x          (integer) position of variable x in the adjacency matrix.
            
y          (integer) position of variable y in the adjacency matrix.
            
set.type  string specifying the type of adjustment set that should be computed. It can be
            "minimal", "all" and "canonical". See Details explanations.

Details

If set.type is "minimal", then only minimal sufficient adjustment sets are returned. If set.type is "all", all valid adjustment sets are returned. If set.type is "canonical", a single adjustment set is returned that consists of all (possible) ancestors of x and y, minus (possible) descendants of nodes on proper causal paths. This canonical adjustment set is always valid if any valid set exists at all.

Value

If adjustment sets exist, list of length at least one (list elements might be empty vectors, if the empty set is an adjustment set). If no adjustment set exists, an empty list is returned.

Author(s)

Emilija Perkovic and Markus Kalisch (<kalisch@stat.math.ethz.ch>)

References


See Also

gac for testing if a set satisfies the Generalized Adjustment Criterion.

Examples

```r
mFig1 <- matrix(c(0,1,1,0,0,0, 1,0,1,1,1,0, 0,0,0,0,0,1,
                  0,1,1,0,1,1, 0,1,0,1,0,1, 0,0,0,0,0,0), 6,6)
type <- "cpdag"
```
Estimate an APDAG within the Markov equivalence class of a DAG using AGES

**Description**

Estimate an APDAG (a particular PDAG) using the aggregative greedy equivalence search (AGES) algorithm, which uses the solution path of the greedy equivalence search (GES) algorithm of Chickering (2002).

**Usage**

```r
ages(data, lambda_min = 0.5 * log(nrow(data)), labels = NULL,
       fixedGaps = NULL, adaptive = c("none", "vstructures", "triples"),
       maxDegree = integer(0), verbose = FALSE, ...)
```

**Arguments**

- `data`: A `n * p` matrix (or data frame) containing the observational data.
- `lambda_min`: The smallest penalty parameter value used when computing the solution path of GES.
- `labels`: Node labels; if NULL the names of the columns of the data matrix (or the names in the data frame) are used. If these are not specified the sequence 1 to `p` is used.
fixedGaps logical symmetric matrix of dimension p*p. If entry \([i,j]\) is TRUE, the result is guaranteed to have no edge between nodes \(i\) and \(j\).

adaptive indicating whether constraints should be adapted to newly detected v-structures or unshielded triples (cf. details).

maxDegree Parameter used to limit the vertex degree of the estimated graph. Valid arguments:
1. Vector of length 0 (default): vertex degree is not limited.
2. Real number \(0 < r < 1\): degree of vertex \(v\) is limited to \(r \cdot n_v\), where \(n_v\) denotes the number of data points where \(v\) was not intervened.
3. Single integer: uniform bound of vertex degree for all vertices of the graph.
4. Integer vector of length \(p\): vector of individual bounds for the vertex degrees.

verbose If TRUE, detailed output is provided.

... Additional arguments for debugging purposes and fine tuning.

Details
This function tries to add orientations to the essential graph (CPDAG) found by ges (ran with lambda=lambda_min). It does it aggregating several CPDAGs present in the solution path of GES. Conceptually, AGES starts with the essential graph found by GES ran with lambda = lambda_min. Then, it checks for further (compatible) orientation information in other essential graphs present in the solution path of GES, i.e., in essential graphs outputted by GES for larger penalty parameters. With compatible we mean that the aggregation process is done such that the final APDAG is still within the Markov equivalence graph represented by the essential graph found by GES in the following sense: an APDAG can always be extended to a DAG without creating new v-structures. This DAG lies in the Markov equivalence class represented by the essential graph found by GES. The algorithm is explained in detail in Eigenmann, Nandy, and Maathuis (2017).

The arguments fixedgaps and adaptive work also with AGES. However, they have not been studied in Eigenmann, Nandy, and Maathuis (2017). Using the argument fixedGaps, one can make sure that certain edges will not be present in the resulting essential graph: if the entry \([i,j]\) of the matrix passed to fixedGaps is TRUE, there will be no edge between nodes \(i\) and \(j\). The argument adaptive can be used to relax the constraints encoded by fixedGaps according to a modification of GES called ARGES (adaptively restricted greedy equivalence search) which has been presented in Nandy, Hauser and Maathuis (2018):

- When adaptive = "vstructures" and the algorithm introduces a new v-structure \(a \rightarrow b \leftarrow c\) in the forward phase, then the edge \(a \rightarrow c\) is removed from the list of fixed gaps, meaning that the insertion of an edge between \(a\) and \(c\) becomes possible even if it was forbidden by the initial matrix passed to fixedGaps.

- When adaptive = "triples" and the algorithm introduces a new unshielded triple in the forward phase (i.e., a subgraph of three nodes \(a\), \(b\) and \(c\) where \(a\) and \(b\) as well as \(b\) and \(c\) are adjacent, but \(a\) and \(c\) are not), then the edge \(a \rightarrow c\) is removed from the list of fixed gaps.

With one of the adaptive modifications, the successive application of a skeleton estimation method and GES restricted to an estimated skeleton still gives a consistent estimator of the DAG, which is not the case without the adaptive modification.
For a detailed explanation of the GES function as well as its related object like essential graphs, we refer to the `ges` function.

Differences in the arguments with respect to GES: AGES uses data to initialize several scores taken as argument by GES. AGES modifies the forward and backward phases of GES performing single steps in either directions. For this reason, phase, iterate, and turning are not available arguments.

Value

`ages` returns a list with the following four components:

- `essgraph`: An object of class `EssGraph` containing an estimate of the equivalence class of the underlying DAG.
- `repr`: An object of a class derived from `ParDAG` containing a (random) representative of the estimated equivalence class.
- `CPDAGsList`: A list of p*p matrices containing all CPDAGs considered by AGES in the aggregation processes.
- `lambda`: A vector containing the penalty parameter used to obtain the list of CPDAGs mentioned above. GES returns the list of CPDAGs when used with this vector of penalty parameters if used with phases = c("forward", "backward") and iterate = FALSE.

Author(s)

Marco Eigenmann (<eigenmann@stat.math.ethz.ch>)

References


See Also

`ges`, `EssGraph`

Examples

```r
## Example 1: ages adds correct orientations: Bar --> V6 and Bar --> V8
set.seed(77)
p <- 8
n <- 5000
## true DAG:
```
vars <- c("Author", "Bar", "Ctrl", "Goal", paste0("V", 5:8))
gGtrue <- randomDAG(p, prob = 0.3, V = vars)
data = rmvDAG(n, gGtrue)

## Estimate the aggregated PDAG with ages
ages.fit <- ages(data = data)

## Estimate the essential graph with ges
## We specify the phases in order to have a fair comparison of the algorithms
## Without the phases specified it would be easy to find examples
## where each algorithm outperforms the other
score <- new("GaussL0penObsScore", data)
ges.fit <- ges(score, phase = c("forward", "backward"), iterate = FALSE)

## Plots
par(mfrow = c(1, 3))
plot(ges.fit$essgraph, main = "Estimated CPDAG with GES")
plot(ages.fit$essgraph, main = "Estimated APDAG with AGES")
plot(gGtrue, main = "TrueDAG")

## Example 2: ages adds correct orientations: Author --> Goal and Author --> V5
set.seed(50)
p <- 9
n <- 5000
## true DAG:
vars <- c("Author", "Bar", "Ctrl", "Goal", paste0("V", 5:9))
gGtrue <- randomDAG(p, prob = 0.5, V = vars)
data = rmvDAG(n, gGtrue)

## Estimate the aggregated PDAG with ages
ages.fit <- ages(data = data)

## Estimate the essential graph with ges
## We specify the phases in order to have a fair comparison of the algorithms
## Without the phases specified it would be easy to find examples
## where each algorithm outperforms the other
score <- new("GaussL0penObsScore", data)
ges.fit <- ges(score, phase = c("forward", "backward"), iterate = FALSE)

## Plots
par(mfrow = c(1, 3))
plot(ges.fit$essgraph, main = "Estimated CPDAG with GES")
plot(ages.fit$essgraph, main = "Estimated APDAG with AGES")
plot(gGtrue, main = "TrueDAG")
## Example 3: ges and ages return the same graph

data(gmG)

data <- gmG$x

## Estimate the aggregated PDAG with ages
ages.fit <- ages(data = data)

## Estimate the essential graph with ges
score <- new("GaussL0penObsScore", data)
ges.fit <- ges(score)

## Plots
par(mfrow=c(1,3))
plot(ges.fit$essgraph, main="Estimated CPDAG with GES")
plot(ages.fit$essgraph, main="Estimated APDAG with AGES")
plot(gmG8$g, main="TrueDAG")

---

amatType

### Description

Two types of adjacency matrices are used in package **pcalg**: Type `amat.cpdag` for DAGs and CPDAGs and type `amat.pag` for MAGs and PAGs. The required type of adjacency matrix is documented in the help files of the respective functions or classes. If in some functions more detailed information on the graph type is needed (i.e. DAG or CPDAG; MAG or PAG) this information will be passed in a separate argument (see e.g. `gac` and the examples below).

Note that you get (`extract`) such adjacency matrices as (S3) objects of class "amat" via the usual `as(.,"<class>")` coercion,

```r
as(from, "amat")
```

### Arguments

- **from**: an R object of class **pcAlgo**, as returned from `skeleton()` or `pc()` or an object of class **fciAlgo**, as from `fci()` (or `rfci`, `fciPlus`, and `dag2pag`), or an object of class "LINGAM" as returned from `lingam()`.
Details

Adjacency matrices are integer valued square matrices with zeros on the diagonal. They can have row- and columnnames; however, most functions will work on the (integer) column positions in the adjacency matrix.

**Coding for type amat.cpdag:**

0: No edge or tail
1: Arrowhead

Note that the edgemark-code refers to the row index (as opposed adjacency matrices of type mag or pag). E.g.:

- amat[a,b] = 0 and amat[b,a] = 1 implies a --> b.
- amat[a,b] = 1 and amat[b,a] = 0 implies a <-- b.
- amat[a,b] = 0 and amat[b,a] = 0 implies a b.
- amat[a,b] = 1 and amat[b,a] = 1 implies a --- b.

**Coding for type amat.pag:**

0: No edge
1: Circle
2: Arrowhead
3: Tail

Note that the edgemark-code refers to the column index (as opposed adjacency matrices of type dag or cpdag). E.g.:

- amat[a,b] = 2 and amat[b,a] = 3 implies a --> b.
- amat[a,b] = 3 and amat[b,a] = 2 implies a <-- b.
- amat[a,b] = 2 and amat[b,a] = 2 implies a <-> b.
- amat[a,b] = 1 and amat[b,a] = 3 implies a --o b.
- amat[a,b] = 0 and amat[b,a] = 0 implies a b.

See Also

E.g. *gac* for a function which takes an adjacency matrix as input; *fciAlgo* for a class which has an adjacency matrix in one slot.

*getGraph(x)* extracts the graph object from x, whereas as(*,*"amat") gets the corresponding adjacency matrix.

Examples

```r
# Function gac() takes an adjacency matrix of any kind as input. In addition to that, the precise type of graph (DAG/CPDAG/MAG/PAG) needs to be passed as a different argument
# Adjacency matrix of type 'amat.cpdag'
```
amatType

\[\text{m1} \leftarrow \text{matrix}(c(0,1,0,1,0,0, 0,0,1,0,1,0, 0,0,0,0,0,0, 0,0,0,0,0,0), 6,6)\]

## more detailed information on the graph type needed by gac()
\[\text{gac(m1, x=1, y=3, z=NULL, type = "dag")}\]

## Adjacency matrix of type 'amat.cpdag'
\[\text{m2} \leftarrow \text{matrix}(c(0,1,0,0,0,1,0,1,1,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0), 6,6)\]

## more detailed information on the graph type needed by gac()
\[\text{gac(m2, x=3, y=6, z=c(2,4), type = "cpdag")}\]

## Adjacency matrix of type 'amat.pag'
\[\text{m3} \leftarrow \text{matrix}(c(0,2,0,0, 3,0,3,3, 0,2,0,3, 0,2,2,0), 4,4)\]

## more detailed information on the graph type needed by gac()
\[\text{mg3} \leftarrow \text{gac(m3, x=2, y=4, z=NULL, type = "mag")}\]
\[\text{pg3} \leftarrow \text{gac(m3, x=2, y=4, z=NULL, type = "pag")}\]

############################################################
## as(*, "amat") returns an adjacency matrix incl. its type
############################################################

## Load predefined data
\[\text{data(gmG)}\]
\[\text{n <- nrow(gmG8$x)}\]
\[\text{V <- colnames(gmG8$x)}\]

## define sufficient statistics
\[\text{suffStat <- list(C = cor(gmG8$x), n = n)}\]

## estimate CPDAG
\[\text{skel.fit <- skeleton(suffStat, indepTest = gaussCItest,}\]
\[\text{alpha = 0.01, labels = V)}\]

## Extract the "amat" [and show nicely via 'print()' method]:
\[\text{as(skel.fit, "amat")}\]

############################################################
## Function fci() returns an adjacency matrix
## of type amat.pag as one slot.
############################################################

\[\text{set.seed(42)}\]
\[\text{p <- 7}\]

## generate and draw random DAG :
\[\text{myDAG} \leftarrow \text{randomDAG(p, prob = 0.4)}\]

## find skeleton and PAG using the FCI algorithm
\[\text{suffStat} \leftarrow \text{list(C = cov2cor(trueCov(myDAG)), n = 10^9)}\]
\[\text{res <- fci(suffStat, indepTest = gaussCItest,}\]
\[\text{alpha = 0.9999, p=p, doPdsep = FALSE)}\]
\[\text{str(res)}\]

## get the (adjacency) mat(rix) and nicely print() it:
\[\text{as(res, "amat")}\]

############################################################
## pcAlgo object
# Load predefined data
data(gmG)

n <- nrow(gmG$x)
V <- colnames(gmG$x)

# Define sufficient statistics
suffStat <- list(C = cor(gmG$x), n = n)

# Estimate CPDAG
skel.fit <- skeleton(suffStat, indepTest = gaussCItest,
                        alpha = 0.01, labels = V)

# Extract Adjacency Matrix - and print (via method 'print.amat'):
as(skel.fit, "amat")

pc.fit <- pc(suffStat, indepTest = gaussCItest,
                        alpha = 0.01, labels = V)
pc.fit # (using its own print() method 'print.pcAlgo')
as(pc.fit, "amat")

---

### backdoor

**Find Set Satisfying the Generalized Backdoor Criterion (GBC)**

#### Description

This function first checks if the total causal effect of one variable (x) onto another variable (y) is identifiable via the GBC, and if this is the case it explicitly gives a set of variables that satisfies the GBC with respect to x and y in the given graph.

#### Usage

```r
text = "pag", max.chordal = 10, verbose=FALSE)
```

#### Arguments

- `amat`: adjacency matrix of type `amat.cpdag` or `amat.pag`.
- `x,y`: (integer) position of variable X and Y, respectively, in the adjacency matrix.
- `type`: string specifying the type of graph of the adjacency matrix amat. It can be a DAG (type="dag"), a CPDAG (type="cpdag"); then the type of the adjacency matrix is assumed to be `amat.cpdag`. It can also be a MAG (type="mag"), or a PAG (type="pag"); then the type of the adjacency matrix is assumed to be `amat.pag`.
- `max.chordal`: only if type = "mag", is used in `pag2magAM` to determine paths too large to be checked for chordality.
- `verbose`: logical; if true, some output is produced during computation.
**Details**

This function is a generalization of Pearl’s backdoor criterion, see Pearl (1993), defined for directed acyclic graphs (DAGs), for single interventions and single outcome variable to more general types of graphs (CPDAGs, MAGs, and PAGs) that describe Markov equivalence classes of DAGs with and without latent variables but without selection variables. For more details see Maathuis and Colombo (2015).

The motivation to find a set W that satisfies the GBC with respect to \( x \) and \( y \) in the given graph relies on the result of the generalized backdoor adjustment:

*If a set of variables \( W \) satisfies the GBC relative to \( x \) and \( y \) in the given graph, then the causal effect of \( x \) on \( y \) is identifiable and is given by*

\[
P(Y|\text{do}(X = x)) = \sum_W P(Y|X, W) \cdot P(W).
\]

This result allows to write post-intervention densities (the one written using Pearl’s do-calculus) using only observational densities estimated from the data.

If the input graph is a DAG (type="dag"), this function reduces to Pearl's backdoor criterion for single interventions and single outcome variable, and the parents of \( x \) in the DAG satisfy the backdoor criterion unless \( y \) is a parent of \( x \).

If the input graph is a CPDAG \( C \) (type="cpdag"), a MAG \( M \) (type="mag"), or a PAG \( P \) (type="pag") (with both \( M \) and \( P \) not allowing selection variables), this function first checks if the total causal effect of \( x \) on \( y \) is identifiable via the GBC (see Maathuis and Colombo, 2015). If the effect is not identifiable in this way, the output is NA. Otherwise, an explicit set \( W \) that satisfies the GBC with respect to \( x \) and \( y \) in the given graph is found.

At this moment this function is not able to work with an RFCI-PAG.

It is important to note that there can be pair of nodes \( x \) and \( y \) for which there is no set \( W \) that satisfies the GBC, but the total causal effect might be identifiable via some other technique.

For the coding of the adjacency matrix see amatType.

**Value**

Either NA if the total causal effect is not identifiable via the GBC, or a set if the effect is identifiable via the GBC. Note that if the set \( W \) is equal to the empty set, the output is NULL.

**Author(s)**

Diego Colombo and Markus Kalisch (<kalisch@stat.math.ethz.ch>)

**References**


See Also

gac for the Generalized Adjustment Criterion (GAC), which is a generalization of GBC; pc for estimating a CPDAG, dag2pag and fci for estimating a PAG, and pag2magAM for estimating a MAG.

Examples

```r
# DAG
# Simulate the true DAG
suppressWarnings(RNGversion("3.5.0"))
set.seed(123)
p <- 7
myDAG <- randomDAG(p, prob = 0.2) ## true DAG
## Extract the adjacency matrix of the true DAG
true.amat <- (amat <- as(myDAG, "matrix")) != 0 # TRUE/FALSE <-> 1/0
print.table(1*true.amat, zero=".") # "visualization"

## Compute set satisfying the GBC:
backdoor(true.amat, 5, 7, type="dag")
```

```r
# CPDAG
# Example not identifiable
# Maathuis and Colombo (2015), Fig. 3a, p.1072
# create the graph
p <- 5
 amat <- rbind(c(.,.,1,1,1),
               c(.,.,1,1,1),
               c(.,.,.,1,.),
               c(.,.,.,.,1),
               c(.,.,.,.,.))
colnames(amat) <- rownames(amat) <- as.character(1:5)
V <- as.character(1:5)
edL <- vector("list",length=5)
names(edL) <- V
edL[[1]] <- list(edges=c(3,4,5), weights=c(1,1,1))
edL[[2]] <- list(edges=c(3,4,5), weights=c(1,1,1))
edL[[3]] <- list(edges=4, weights=c(1))
edL[[4]] <- list(edges=5, weights=c(1))
g <- new("graphNEL", nodes=V, edgeL=edL, edgemode="directed")
## estimate the true CPDAG
myCPDAG <- dag2cpdag(g)
## Extract the adjacency matrix of the true CPDAG
```

true.amat <- (as(myCPDAG, "matrix") != 0) # 1/0 <==> TRUE/FALSE

## The effect is not identifiable, in fact:
backdoor(true.amat, 3, 5, type="cpdag")

##################################################
#### Example identifiable
#### Maathuis and Colombo (2015), Fig. 3b, p.1072
##################################################

## create the graph
p <- 6
amat <- rbind(c(0,0,1,1,0,1), c(0,0,1,1,0,1), c(0,0,0,0,1,0),
              c(0,0,0,0,1,1), c(0,0,0,0,0,0), c(0,0,0,0,0,0))
colnames(amat) <- rownames(amat) <- as.character(1:6)
V <- as.character(1:6)
edL <- vector("list",length=6)
names(edL) <- V
edL[[1]] <- list(edges=c(3,4,6),weights=c(1,1,1))
edL[[2]] <- list(edges=c(3,4,6),weights=c(1,1,1))
edL[[3]] <- list(edges=5,weights=c(1))
edL[[4]] <- list(edges=c(5,6),weights=c(1,1))
g <- new("graphNEL", nodes=V, edgeL=edL, edgemode="directed")

## estimate the true CPDAG
myCPDAG <- dag2cpdag(g)
## Extract the adjacency matrix of the true CPDAG
true.amat <- as(myCPDAG, "matrix") != 0

## The effect is identifiable and the set satisfying GBC is:
backdoor(true.amat, 6, 3, type="cpdag")

##################################################################
#### PAG
##################################################################

##################################################
#### Example identifiable
#### Maathuis and Colombo (2015), Fig. 5a, p.1075
##################################################

## create the graph
p <- 7
amat <- t(matrix(c(0,0,1,1,0,0, 0,0,1,1,0,0, 0,0,0,1,0,1, 0,0,0,0,0,0, 0,0,0,0,0,1, 0,0,0,0,0,0, 0,0,0,0,0,0), 7, 7))
colnames(amat) <- rownames(amat) <- as.character(1:7)
V <- as.character(1:7)
edL <- vector("list",length=7)
names(edL) <- V
edL[[1]] <- list(edges=c(3,4),weights=c(1,1))
edL[[2]] <- list(edges=c(3,4),weights=c(1,1))
edL[[3]] <- list(edges=c(4,6), weights=c(1,1))
edL[[4]] <- list(edges=7, weights=c(1))
edL[[5]] <- list(edges=c(6,7), weights=c(1,1))
g <- new("graphNEL", nodes=V, edgeL=edL, edgemode="directed")
L <- 5

## compute the true covariance matrix of g
cov.mat <- trueCov(g)

## transform covariance matrix into a correlation matrix
true.corr <- cov2cor(cov.mat)
suffStat <- list(C=true.corr, n=10^9)
indepTest <- gaussCITest

## estimate the true PAG
true.pag <- dag2pag(suffStat, indepTest, g, L, alpha = 0.9999)@amat

## The effect is identifiable and the backdoor set is:
backdoor(true.pag, 3, 5, type="pag")

---

**beta.special**

*Compute set of intervention effects*

**Description**

This function is DEPRECATED! Use *ida* instead.

**Usage**

```r
beta.special(dat=NA, x.pos, y.pos, verbose=0, a=0.01, myDAG=NA,
             myplot=FALSE, perfect=FALSE, method="local", collTest=TRUE,
pcObj=NA, all.dags=NA, u2pd="rand")
```

**Arguments**

- **dat** | Data matrix
- **x.pos, y.pos** | integer column positions of *x* and *y* in *dat*
- **verbose** | 0=no comments, 2=detail on estimates
- **a** | Significance level of tests for finding CPDAG
- **myDAG** | Needed if true correlation matrix shall be computed
- **myplot** | Plot estimated graph
- **perfect** | True cor matrix is calculated from myDAG
- **method** | "local" - local (all combinations of parents in regr.); "global" - all DAGs
- **collTest** | True - Exclude orientations of undirected edges that introduce a new collider
- **pcObj** | Fit of PC Algorithm (CPDAG); if this is available, no new fit is done
all.dags  All DAGs in the format of function allDags; if this is available, no new function call allDags is done
u2pd    function for converting a UDAG to a PDAG; "rand": udag2pdag; "relaxed": udag2pdagRelaxed; "retry": udag2pdagSpecial.

Value
estimates of intervention effects

Author(s)
Markus Kalisch (<kalisch@stat.math.ethz.ch>)

See Also
pcAlgo, dag2cpdag; beta.special.pcObj for a fast version of beta.special(), using a precomputed pc-object.

beta.special.pcObj  Compute set of intervention effects in a fast way

Description
This function is DEPRECATED! Use ida or idaFast instead.

Usage
beta.special.pcObj(x.pos, y.pos, pcObj, mcov=NA, amat=NA, amatSkel=NA, t.amat=NA)

Arguments
  x.pos      Column of x in dat
  y.pos      Column of y in dat
  pcObj      Precomputed pc-object
  mcov       Covariance that was used in the pc-object fit
  amat, amatSkel, t.amat
              Matrices that can be precomputed, if needed (see code for details on how to precompute)

Value
estimates of intervention effects

Author(s)
Markus Kalisch (<kalisch@stat.math.ethz.ch>)
**Description**

$G^2$ test for (conditional) independence of binary variables $X$ and $Y$ given the (possibly empty) set of binary variables $S$.

binCItest() is a wrapper of gSquareBin(), to be easily used in skeleton, pc and fci.

**Usage**

\[
gSquareBin(x, y, S, dm, adaptDF = FALSE, n.min = 10*df, verbose = FALSE)  
\]

\[
binCItest(x, y, S, suffStat)  
\]

**Arguments**

- **x, y** (integer) position of variable $X$ and $Y$, respectively, in the adjacency matrix.
- **S** (integer) positions of zero or more conditioning variables in the adjacency matrix.
- **dm** data matrix (with \{0, 1\} entries).
- **adaptDF** logical specifying if the degrees of freedom should be lowered by one for each zero count. The value for the degrees of freedom cannot go below 1.
- **n.min** the smallest $n$ (number of observations, nrow(dm)) for which the $G^2$ test is computed; for smaller $n$, independence is assumed ($G^2 := 1$) with a warning. The default is $10m$, where $m$ is the degrees of freedom assuming no structural zeros, $2^{|S|}$.
- **verbose** logical or integer indicating that increased diagnostic output is to be provided.
- **suffStat** a list with two elements, "dm", and "adaptDF" corresponding to the above two arguments of gSquareBin().

**Details**

The $G^2$ statistic is used to test for (conditional) independence of X and Y given a set S (can be NULL). This function is a specialized version of gSquareDis which is for discrete variables with more than two levels.

**Value**

The p-value of the test.

**Author(s)**

Nicoletta Andri and Markus Kalisch (<kalisch@stat.math.ethz.ch>)
References


See Also

gSquareDis for a (conditional) independence test for discrete variables with more than two levels.
dsepTest, gaussCItest and disCItest for similar functions for a d-separation oracle, a conditional independence test for Gaussian variables and a conditional independence test for discrete variables, respectively.
skeleton, pc or fci which need a testing function such as binCItest.

Examples

n <- 100
set.seed(123)
## Simulate *independent data of {0,1}-variables:
x <- rbinom(n, 1, pr=1/2)
y <- rbinom(n, 1, pr=1/2)
z <- rbinom(n, 1, pr=1/2)
dat <- cbind(x,y,z)

binCItest(1,3,2, list(dm = dat, adaptDF = FALSE)) # 0.36, not signif.
binCItest(1,3,2, list(dm = dat, adaptDF = TRUE )) # the same, here

## Simulate data from a chain of 3 variables: x1 -> x2 -> x3
set.seed(12)
b0 <- 0
b1 <- 1
b2 <- 1
n <- 10000
x1 <- rbinom(n, size=1, prob=1/2) ## = sample(c(0,1), n, replace=TRUE)
## NB: plogis(u) := "expit(u)" := exp(u) / (1 + exp(u))
p2 <- plogis(b0 + b1*x1) ; x2 <- rbinom(n, 1, prob = p2) # {0,1}
p3 <- plogis(b0 + b2*x2) ; x3 <- rbinom(n, 1, prob = p2) # {0,1}

ftable(xtabs(~ x1+x2+x3))
dat <- cbind(x1,x2,x3)

## Test marginal and conditional independencies
gSquareBin(3,1,NULL,dat, verbose=TRUE)
gSquareBin(3,1, 2, dat)
gSquareBin(1,3, 2, dat) # the same
gSquareBin(1,3, 2, dat, adaptDF=TRUE, verbose = 2)
checkTriple

Check Consistency of Conditional Independence for a Triple of Nodes

Description

For each subset of nbrsA and nbrsC where a and c are conditionally independent, it is checked if b is in the conditioning set.

Usage

checkTriple(a, b, c, nbrsA, nbrsC, 
    sepsetA, sepsetC, 
    suffStat, indepTest, alpha, version.unf = c(NA, NA), 
    maj.rule = FALSE, verbose = FALSE)

Arguments

a, b, c (integer) positions in adjacency matrix for nodes a, b, and c, respectively.
nbrsA, nbrsC (integer) position in adjacency matrix for neighbors of a and c, respectively.
sepsetA vector containing Sepset(a,c).
sepsetC vector containing Sepset(c,a).
suffStat a list of sufficient statistics for independent tests; see, e.g., pc.
indepTest a function for the independence test, see, e.g., pc.
alpha significance level of test.
version.unf (integer) vector of length two:
    version.unf[1]: 1 - check for all separating subsets of nbrsA and nbrsC if b is in that set,
        2 - it also checks if there at all exists any sepset which is a subset of the neighbours (there might be none, although b is in the sepset, which indicates an ambiguous situation);
    version.unf[2]: 1 - do not consider the initial sepsets sepsetA and sepsetC (same as Tetrad),
        2 - consider if b is in sepsetA or sepsetC.
maj.rule logical indicating that the following majority rule is applied: if b is in less than 50% of the checked sepsets, we say that b is in no sepset. If b is in more than 50% of the checked sepsets, we say that b is in all sepsets. If b is in exactly 50% of the checked sepsets, the triple is considered 'ambiguous'.
verbose Logical asking for detailed output of intermediate steps.

Details

This function is used in the conservative versions of structure learning algorithms.
checkTriple

**Value**

- **decision**: Decision on possibly ambiguous triple, an integer code.
  1. b is in NO sepset (make v-structure);
  2. b is in ALL sepsets (make no v-structure);
  3. b is in SOME but not all sepsets (ambiguous triple)

- **vers**: Version (1 or 2) of the ambiguous triple (1=normal ambiguous triple that is b is in some sepsets; 2=triple coming from version.unf[1]==2, that is, a and c are indep given the initial sepset but there doesn’t exist a subset of the neighbours that d-separates them.)

- **sepsetA**: Updated version of sepsetA
- **sepsetC**: Updated version of sepsetC

**Author(s)**

Markus Kalisch (<kalisch@stat.math.ethz.ch>) and Diego Colombo.

**References**


**Examples**

```r
# Using Gaussian Data
## Load predefined data
data(gmG)
n <- nrow(gmG$x)
V <- colnames(gmG$x)

## define independence test (partial correlations), and test level
indepTest <- gaussCItest
alpha <- 0.01
## define sufficient statistics
suffStat <- list(C = cor(gmG$x), n = n)

## estimate CPDAG
pc.fit <- pc(suffStat, indepTest, alpha=alpha, labels = V, verbose = TRUE)

if (require(Rgraphviz)) {
  ## show estimated CPDAG
  par(mfrow=c(1,2))
  plot(pc.fit, main = "Estimated CPDAG")
  plot(gmG$g, main = "True DAG")
}

a <- 6
```
compareGraphs

Compare two graphs in terms of TPR, FPR and TDR

Description

Compares the true undirected graph with an estimated undirected graph in terms of True Positive Rate (TPR), False Positive Rate (FPR) and True Discovery Rate (TDR).

Usage

compareGraphs(gl, gt)

Arguments

- **gl**: Estimated graph (graph object)
- **gt**: True graph (graph object)

Details

If the input graph is directed, the directions are omitted. Special cases:

- If the true graph contains no edges, the tpr is defined to be zero.
- Similarly, if the true graph contains no gaps, the fpr is defined to be one.
- If there are no edges in the true graph and there are none in the estimated graph, tdr is one. If there are none in the true graph but there are some in the estimated graph, tdr is zero.

```r
b <- 1
c <- 8
ccheckTriple(a, b, c,
nbrsA = c(1, 5, 7),
nbrsC = c(1, 5),
sepsetA = pc.fit@sepset[[a]][[c]],
sepsetC = pc.fit@sepset[[c]][[a]],
suffStat = suffStat, indepTest = indepTest, alpha = alpha,
version.unf = c(2, 2),
verbose = TRUE) -> ct
str(ct)
# List of 4
# $ decision: int 2
# $ version: int 1
# $ SepsetA: int [1:2] 1 5
# $ SepsetC: int 1

ccheckTriple(a, b, c,
nbrsA = c(1, 5, 7),
nbrsC = c(1, 5),
sepsetA = pc.fit@sepset[[a]][[c]],
sepsetC = pc.fit@sepset[[c]][[a]],
version.unf = c(1, 1),
suffStat = suffStat, indepTest = indepTest, alpha = alpha) -> c2
stopifnot(identical(ct, c2)) # in this case, 'version.unf' had no effect
```
A named numeric vector with three numbers:

- **tpr**: True Positive Rate: Number of correctly found edges (in estimated graph) divided by number of true edges (in true graph)
- **fpr**: False Positive Rate: Number of incorrectly found edges divided by number of true gaps (in true graph)
- **tdr**: True Discovery Rate: Number of correctly found edges divided by number of found edges (both in estimated graph)

**Author(s)**

Markus Kalisch (<kalisch@stat.math.ethz.ch>) and Martin Maechler

**See Also**

`randomDAG` for generating a random DAG.

**Examples**

```r
## generate a graph with 4 nodes
V <- LETTERS[1:4]
edL2 <- vector("list", length=4)
names(edL2) <- V
edL2[[1]] <- list(edges= 2)
edL2[[2]] <- list(edges= c(1,3,4))
edL2[[3]] <- list(edges= c(2,4))
edL2[[4]] <- list(edges= c(2,3))
gt <- new("graphNEL", nodes=V, edgeL=edL2, edgemode="undirected")

## change graph
gl <- graph::addEdge("A","C", gt,1)

## compare the two graphs
if (require(Rgraphviz)) {
  par(mfrow=c(2,1))
  plot(gt) ; title("True graph")
  plot(gl) ; title("Estimated graph")
  (cg <- compareGraphs(gl,gt))
}
```
condIndFisherZ  

Test Conditional Independence of Gaussians via Fisher’s Z

Description

Using Fisher’s z-transformation of the partial correlation, test for zero partial correlation of sets of normally / Gaussian distributed random variables.

The `gaussCItest()` function, using `zStat()` to test for (conditional) independence between gaussian random variables, with an interface that can easily be used in `skeleton, pc` and `fci`.

Usage

```r
condIndFisherZ(x, y, S, C, n, cutoff, verbose= )
zStat (x, y, S, C, n)
gaussCItest (x, y, S, suffStat)
```

Arguments

- `x, y, S` (integer) position of variable `X`, `Y` and set of variables `S`, respectively, in the adjacency matrix. It is tested, whether `X` and `Y` are conditionally independent given the subset `S` of the remaining nodes.
- `C` Correlation matrix of nodes
- `n` Integer specifying the number of observations (“samples”) used to estimate the correlation matrix `C`.
- `cutoff` Numeric cutoff for significance level of individual partial correlation tests. Must be set to `qnorm(1 - alpha/2)` for a test significance level of `alpha`.
- `verbose` Logical indicating whether some intermediate output should be shown; currently not used.
- `suffStat` A list with two elements, "C" and "n", corresponding to the above arguments with the same name.

Details

For gaussian random variables and after performing Fisher’s z-transformation of the partial correlation, the test statistic `zStat()` is (asymptotically for large enough `n`) standard normally distributed.

Partial correlation is tested in a two-sided hypothesis test, i.e., basically, `condIndFisherZ(*) == abs(zStat(*)) > qnorm(1 -alpha/2)`. In a multivariate normal distribution, zero partial correlation is equivalent to conditional independence.

Value

`zStat()` gives a number

\[ Z = \sqrt{n - |S| - 3} \cdot \log((1 + r)/(1 - r))/2 \]

which is asymptotically normally distributed under the null hypothesis of correlation 0.
condIndFisherZ() returns a logical L indicating whether the "partial correlation of x and y given S is zero" could not be rejected on the given significance level. More intuitively and for multivariate normal data, this means: If TRUE then it seems plausible, that x and y are conditionally independent given S. If FALSE then there was strong evidence found against this conditional independence statement.

gaussCItest() returns the p-value of the test.

Author(s)

Markus Kalisch (<kalisch@stat.math.ethz.ch>) and Martin Maechler

References


See Also

*pcorOrder* for computing a partial correlation given the correlation matrix in a recursive way.

*dsepTest, disCItest and binCItest* for similar functions for a d-separation oracle, a conditional independence test for discrete variables and a conditional independence test for binary variables, respectively.

Examples

```r
set.seed(42)
## Generate four independent normal random variables
n <- 20
data <- matrix(rnorm(n*4),n,4)
## Compute corresponding correlation matrix
corMatrix <- cor(data)
## Test, whether variable 1 (col 1) and variable 2 (col 2) are
## independent given variable 3 (col 3) and variable 4 (col 4) on 0.05
## significance level
x <- 1
y <- 2
S <- c(3,4)
n <- 20
alpha <- 0.05
cutoff <- qnorm(1-alpha/2)
(b1 <- condIndFisherZ(x,y,S,corMatrix,n,cutoff))
# -> 1 and 2 seem to be conditionally independent given 3,4

## Now an example with conditional dependence
data <- matrix(rnorm(n*3),n,3)
data[,3] <- 2*data[,1]
corMatrix <- cor(data)
(b2 <- condIndFisherZ(1,3,2,corMatrix,n,cutoff))
# -> 1 and 3 seem to be conditionally dependent given 2

## simulate another dep.case: x -> y -> z
```
set.seed(29)
x <- rnorm(100)
y <- 3*x + rnorm(100)
z <- 2*y + rnorm(100)
dat <- cbind(x,y,z)

## analyze data
suffStat <- list(C = cor(dat), n = nrow(dat))
gaussCItest(1,3,NULL, suffStat) ## dependent [highly signif.]
gaussCItest(1,3, 2, suffStat) ## independent | S

---

corGraph | Computing the correlation graph

**Description**

Computes the correlation graph. This is the graph in which an edge is drawn between node i and node j, if the null hypothesis “Correlation between Xᵢ and Xⱼ is zero” can be rejected at the given significance level α(α).

**Usage**

corGraph(dm, alpha=0.05, Cmethod="pearson")

**Arguments**

dm | numeric matrix with rows as samples and columns as variables.
alpha | significance level for correlation test (numeric)
Cmethod | a character string indicating which correlation coefficient is to be used for the test. One of "pearson", "kendall", or "spearman", can be abbreviated.

**Value**

Undirected correlation graph, a graph object (package graph); getGraph for the “fitted” graph.

**Author(s)**

Markus Kalisch (<kalisch@stat.math.ethz.ch>) and Martin Maechler

**Examples**

```r
## create correlated samples
x1 <- rnorm(100)
x2 <- rnorm(100)
mat <- cbind(x1,x2, x3 = x1+x2)

if (require(Rgraphviz)) {
  ## `analyze the data''
  (g <- corGraph(mat)) # a 'graphNEL' graph, undirected
```
plot(g)  # ==> (1) and (2) are each linked to (3)

## use different significance level and different method
(g2 <- corGraph(mat, alpha=0.01, Cmethod="kendall"))
plot(g2)  ## same edges as 'g'

---

**Description**

Convert a DAG (Directed Acyclic Graph) to a Completed Partially Directed Acyclic Graph (CPDAG).

**Usage**

dag2cpdag(g)

**Arguments**

- **g**: an R object of class "graph" (package graph), representing a DAG.

**Details**

This function converts a DAG into its corresponding (unique) CPDAG as follows. Because every DAG in the Markov equivalence class described by a CPDAG shares the same skeleton and the same v-structures, this function takes the skeleton and the v-structures of the given DAG g. Afterwards it simply uses the 3 orientation rules of the PC algorithm (see references) to orient as many of the remaining undirected edges as possible.

The function is a simple wrapper function for dag2essgraph which is more powerful since it also allows the calculation of the Markov equivalence class in the presence of interventional data.

The output of this function is exactly the same as the one using

pc(suffStat, indepTest, alpha, labels)

using the true correlation matrix in the function gaussCItest with a large virtual sample size and a large alpha, but it is much faster.

**Value**

A graph object containing the CPDAG.

**Author(s)**

Markus Kalisch (<kalisch@stat.math.ethz.ch>) and Alain Hauser(<alain.hauser@bfh.ch>)
References


See Also
dag2essgraph, randomDAG, pc

Examples

```r
## A -> B <- C
am1 <- matrix(c(0,1,0, 0,0,0, 0,1,0), 3,3)
colnames(am1) <- rownames(am1) <- LETTERS[1:3]
g1 <- as(t(am1), "graphNEL") ## convert to graph
cpdag1 <- dag2cpdag(g1)
if(requireNamespace("Rgraphviz")) {
  par(mfrow = c(1,2))
  plot(g1)
  plot(cpdag1)
}

## A -> B -> C
am2 <- matrix(c(0,1,0, 0,0,1, 0,0,0), 3,3)
colnames(am2) <- rownames(am2) <- LETTERS[1:3]
g2 <- as(t(am2), "graphNEL") ## convert to graph
cpdag2 <- dag2cpdag(g2)
if(requireNamespace("Rgraphviz")) {
  par(mfrow = c(1,2))
  plot(g2)
  plot(cpdag2)
}
```

---

dag2essgraph

Convert a DAG to an Essential Graph

Description

Convert a DAG to an (interventional or observational) essential graph.

Usage

dag2essgraph(dag, targets = list(integer(0)))
Arguments

dag 
The DAG whose essential graph has to be calculated. Different representations are possible: dag can be an object of graphNEL (package graph) or an instance of a class derived from ParDAG.

targets 
List of intervention targets with respect to which the essential graph has to be calculated. An observational setting is represented by one single empty target (list(integer(0))).

Details

This function converts a DAG to its corresponding (interventional or observational) essential graph, using the algorithm of Hauser and Bühlmann (2012).

The essential graph is a partially directed graph that represents the (interventional or observational) Markov equivalence class of a DAG. It has the same skeleton as the DAG; a directed edge represents an arrow that has a common orientation in all representatives of the (interventional or observational) Markov equivalence class, whereas an undirected edge represents an arrow that has different orientations in different representatives of the equivalence class. In the observational case, the essential graph is also known as “CPDAG” (Spirtes et al., 2000).

In a purely observational setting (i.e., if targets = list(integer(0)), the function yields the same graph as dag2cpdag.

Value

Depending on the class of dag, the essential graph is returned as

- an instance of graphNEL, if dag is an instance of graphNEL,
- an instance of EssGraph, if dag is an instance of a class derived from ParDAG.

Author(s)

Alain Hauser (<alain.hauser@bfh.ch>)

References


See Also

dag2cpdag, Score, EssGraph
Examples

```r
p <- 10  # Number of random variables
s <- 0.4  # Sparseness of the DAG

## Generate a random DAG
set.seed(42)
require(graph)
dag <- randomDAG(p, s)
nodes(dag) <- sprintf("V%d", 1:p)

## Calculate observational essential graph
res.obs <- dag2essgraph(dag)

## Different argument classes
res2 <- dag2essgraph(as(dag, "GaussParDAG"))
str(res2)

## Calculate interventional essential graph for intervention targets
## {1} and {3}
res.int <- dag2essgraph(dag, as.list(c(1, 3)))
```

---

dag2pag

Convert a DAG with latent variables into a PAG

**Description**

Convert a DAG with latent variables into its corresponding (unique) Partial Ancestral Graph (PAG).

**Usage**

```r
dag2pag(suffStat, indepTest, graph, L, alpha, rules = rep(TRUE, 10),
verbose = FALSE)
```

**Arguments**

- `suffStat` the sufficient statistics, a list containing all necessary elements for the conditional independence decisions in the function `indepTest`.
- `indepTest` a function for testing conditional independence. The function is internally called as `indepTest(x, y, S, suffStat)`, and tests conditional independence of x and y given S. Here, x and y are variables, and S is a (possibly empty) vector of variables (all variables are denoted by their column numbers in the adjacency matrix). `suffStat` is a list containing all relevant elements for the conditional independence decisions. The return value of `indepTest()` is the p-value of the test for conditional independence.
- `graph` a DAG with p nodes, a graph object. The graph must be topological sorted (for example produced using `randomDAG`).
- `L` array containing the labels of the nodes in the graph corresponding to the latent variables.
alpha  significance level in \((0, 1)\) for the individual conditional independence tests.

rules  logical vector of length 10 indicating which rules should be used when directing edges. The order of the rules is taken from Zhang (2009).

verbose logical; if TRUE, detailed output is provided.

Details

This function converts a DAG (graph object) with latent variables into its corresponding (unique) PAG, an \texttt{fciAlgo} class object, using the ancestor information and conditional independence tests entailed in the true DAG. The output of this function is exactly the same as the one using

\[
\text{fci(suffStat, gaussCItest, p, alpha, rules = rep(TRUE, 10))}
\]

using the true correlation matrix in \texttt{gaussCItest()} with a large “virtual sample size” and a large alpha, but it is much faster, see the example.

Value

An object of \texttt{class fciAlgo}, containing the estimated graph (in the form of an adjacency matrix with various possible edge marks), the conditioning sets that lead to edge removals (sepset) and several other parameters.

Author(s)

Diego Colombo and Markus Kalisch \textless kalisch@stat.math.ethz.ch\textgreater.

References


See Also

\texttt{fci, pc}

Examples

```r
## create the graph
set.seed(78)
g <- randomDAG(10, prob = 0.25)
graph::nodes(g) # "1" "2" ... "10" % FIXME: should be kept in result!

## define nodes 2 and 6 to be latent variables
L <- c(2,6)

## compute the true covariance matrix of g
cov.mat <- trueCov(g)
## transform covariance matrix into a correlation matrix
true.corr <- cov2cor(cov.mat)
```
## Find PAG
## as dependence "oracle", we use the true correlation matrix in
gaussCItest() with a large "virtual sample size" and a large alpha:

```r
system.time(
  true.pag <- dag2pag(suffStat = list(C = true.corr, n = 10^9),
                      indepTest = gaussCItest,
                      graph=g, L=L, alpha = 0.9999) )
```

### ---- Find PAG using fci-function --------------------------

## From trueCov(g), delete rows and columns belonging to latent variable L
true.cov1 <- cov.mat[-L,-L]
## transform covariance matrix into a correlation matrix
true.corr1 <- cov2cor(true.cov1)
## Find PAG with FCI algorithm
## as dependence "oracle", we use the true correlation matrix in
gaussCItest() with a large "virtual sample size" and a large alpha:

```r
system.time(
  true.pag1 <- fci(suffStat = list(C = true.corr1, n = 10^9),
                    indepTest = gaussCItest,
                    p = ncol(true.corr1), alpha = 0.9999) )
```

## confirm that the outputs are equal
stopifnot(true.pag@amat == true.pag1@amat)

---

### disCItest

**G square Test for (Conditional) Independence of Discrete Variables**

**Description**

$G^2$ test for (conditional) independence of discrete (each with a finite number of “levels”) variables $X$ and $Y$ given the (possibly empty) set of discrete variables $S$.

disCItest() is a wrapper of gSquareDis(), to be easily used in skeleton, pc and fci.

**Usage**

```r
gSquareDis(x, y, S, dm, nlev, adaptDF = FALSE, n.min = 10*df, verbose = FALSE)
disCItest (x, y, S, suffStat)
```

**Arguments**

- `x, y` (integer) position of variable $X$ and $Y$, respectively, in the adjacency matrix.
- `S` (integer) positions of zero or more conditioning variables in the adjacency matrix.
- `dm` data matrix (rows: samples, columns: variables) with integer entries; the k levels for a given column must be coded by the integers 0,1,...,k-1. (see example)
- `nlev` optional vector with numbers of levels for each variable in dm.
adaptDF logical specifying if the degrees of freedom should be lowered by one for each zero count. The value for the degrees of freedom cannot go below 1.

n.min the smallest n (number of observations, nrow(dm)) for which the G^2 test is computed; for smaller n, independence is assumed (G^2 := 1) with a warning. The default is 10m, where m is the degrees of freedom assuming no structural zeros, here, the product of all the number of levels (nlev[x]-1) * (nlev[y]-1) * prod(nlev[S]).

verbose logical or integer indicating that increased diagnostic output is to be provided.

suffStat a list with three elements, "dm", "nlev", "adaptDF"; each corresponding to the above arguments of gSquareDis().

Details

The G^2 statistic is used to test for (conditional) independence of X and Y given a set S (can be NULL). If only binary variables are involved, gSquareBin is a specialized (a bit more efficient) alternative to gSquareDis().

Value

The p-value of the test.

Author(s)

Nicoletta Andri and Markus Kalisch (<kalisch@stat.math.ethz.ch>).

References


See Also

gSquareBin for a (conditional) independence test for binary variables.
dsepTest, gaussCItest and binCItest for similar functions for a d-separation oracle, a conditional independence test for gaussian variables and a conditional independence test for binary variables, respectively.

Examples

```r
## Simulate data
n <- 100
set.seed(123)
x <- sample(0:2, n, TRUE) ## three levels
y <- sample(0:3, n, TRUE) ## four levels
z <- sample(0:1, n, TRUE) ## two levels
dat <- cbind(x, y, z)

## Analyze data
gSquareDis(1, 3, S=2, dat, nlev = c(3, 4, 2)) # but nlev is optional:
```
dreach

Description

Let x and y be two distinct vertices in a mixed graph G. This function computes D-SEP(x,y,G),
which is defined as follows:

A node v is in D-SEP(x,y,G) iff v is not equal to x and there is a collider path between x and v in G
such that every vertex on this path is an ancestor of x or y in G.


Usage

dreach(x, y, amat, verbose = FALSE)

Arguments

x First argument of D-SEP, given as the column number of the node in the adjacency matrix.
y Second argument of D-SEP, given as the column number of the node in the adjacency matrix (y must be different from x).
amat Adjacency matrix of type amat.pag.
verbose Logical specifying details should be on output

Value

Vector of column positions indicating the nodes in D-SEP(x,y,G).

Author(s)

Diego Colombo and Markus Kalisch (<kalisch@stat.math.ethz.ch>)

References

MIT Press.

1060-1088.
**dsep**

*Test for d-separation in a DAG*

**Description**

This function tests for d-separation of nodes in a DAG.

**Usage**

```r
dsep(a, b, S=NULL, g, john.pairs = NULL)
```

**Arguments**

- `a` Label (sic!) of node A
- `b` Label (sic!) of node B
- `S` Labels (sic!) of set of nodes on which it is conditioned, maybe empty
- `g` The Directed Acyclic Graph (object of class "graph", see graph-class from the package RBGL)
- `john.pairs` The shortest path distance matrix for all pairs of nodes as computed (also by default) in johnson.all.pairs.sp from package RBGL.

**Details**

This function checks separation in the moralized graph as explained in Lauritzen (2004).

**Value**

TRUE if a and b are d-separated by S in G, otherwise FALSE.

**Author(s)**

Markus Kalisch (<kalisch@stat.math.ethz.ch>)

**References**

S.L. Lauritzen (2004), Graphical Models, *Oxford University Press*, Chapter 3.2.2

**See Also**

dsepTest for a wrapper of this function that can easily be included into skeleton, pc or fci
Examples

```r
# generate random DAG
p <- 8
set.seed(45)
myDAG <- randomDAG(p, prob = 0.3)
if (require(Rgraphviz)) {
  plot(myDAG)
}

# Examples for d-separation
dsep("1","7",NULL,myDAG)
dsep("4","5",NULL,myDAG)
dsep("4","5","2",myDAG)
dsep("4","5",c("2","3"),myDAG)

# Examples for d-connection
```

### dsepTest

Test for d-separation in a DAG

**Description**

Tests for d-separation of nodes in a DAG. `dsepTest()` is written to be easily used in `skeleton`, `pc`, `fci`.

**Usage**

`dsepTest(x, y, S=NULL, suffStat)`

**Arguments**

- `x,y` (integer) position of variable \(X\) and \(Y\), respectively, in the adjacency matrix.
- `S` (integer) positions of zero or more conditioning variables in the adjacency matrix.
- `suffStat` a list with two elements,
  - "g" Containing the Directed Acyclic Graph (object of class "graph", see `graph-class` from the package `graph`), and
  - "jp" Containing the shortest path distance matrix for all pairs of nodes as computed by `johnson.all.pairs.sp` from package `RBGL`.

**Details**

The function is based on `dsep`. For details on d-separation see the reference Lauritzen (2004).
Value

If x and y are d-separated by S in DAG G the result is 1, otherwise it is 0. This is analogous to the p-value of an ideal (without sampling error) conditional independence test on any distribution that is faithful to the DAG G.

Author(s)

Markus Kalisch (<kalisch@stat.math.ethz.ch>)

References

S.L. Lauritzen (2004), Graphical Models, Oxford University Press.

See Also

gaussCItest, disCItest and binCItest for similar functions for a conditional independence test for gaussian, discrete and binary variables, respectively.

Examples

```r
p <- 8
set.seed(45)
myDAG <- randomDAG(p, prob = 0.3)

if (require(Rgraphviz)) {
  ## plot the DAG
  plot(myDAG, main = "randomDAG(10, prob = 0.2)"
}

## define sufficient statistics (d-separation oracle)
suffStat <- list(g = myDAG, jp = RBGL::johnson.all.pairs.sp(myDAG))
dsepTest(1,6, S= NULL, suffStat) ## not d-separated
dsepTest(1,6, S= 3, suffStat) ## not d-separated by node 3
dsepTest(1,6, S= c(3,4), suffStat) ## d-separated by node 3 and 4
```

---

EssGraph-class

Class "EssGraph"

Description

This class represents an (observational or interventional) essential graph.
Details

An observational or interventional Markov equivalence class of DAGs can be uniquely represented by a partially directed graph, the essential graph. Its edges have the following interpretation:

1. a directed edge \( a \rightarrow b \) stands for an arrow that has the same orientation in all representatives of the Markov equivalence class;
2. an undirected edge \( a - b \) stands for an arrow that is oriented in one way in some representatives of the equivalence class and in the other way in other representatives of the equivalence class.

Extends

All reference classes extend and inherit methods from "envRefClass".

Constructor

new("EssGraph", nodes, in.edges, ...)

nodes Vector of node names; cf. also field .nodes.
in.edges A list of length \( p \) consisting of index vectors indicating the edges pointing into the nodes of the DAG.

Fields

.nodes: Vector of node names; defaults to as.character(1:p), where \( p \) denotes the number of nodes (variables) of the model.
.in.edges: A list of length \( p \) consisting of index vectors indicating the edges pointing into the nodes of the DAG.
targets List of mutually exclusive intervention targets with respect to which Markov equivalence is defined.
score: Object of class Score; used internally for score-based causal inference.

Class-Based Methods

Most class-based methods are only for internal use. Methods of interest for the user are:

repr(): Yields a representative causal model of the equivalence class, an object of a class derived from Score. Since the representative is not only characterized by the DAG, but also by appropriate parameters, the field score must be assigned for this method to work. The DAG is drawn at random; note that all representatives are statistically indistinguishable under a given set of intervention targets.

node.count(): Yields the number of nodes of the essential graph.
edge.count(): Yields the number of edges of the essential graph. Note that unoriented edges count as 2, whereas oriented edges count as 1 due to the internal representation.

Methods

plot signature(x = "EssGraph", y = "ANY"): plots the essential graph. In the plot, undirected and bidirected edges are equivalent.
fci

Author(s)

Alain Hauser (<alain.hauser@bfh.ch>)

See Also

ParDAG

Examples

showClass("EssGraph")

fci

Estimate a PAG by the FCI Algorithm

Description

Estimate a Partial Ancestral Graph (PAG) from observational data, using the FCI (Fast Causal Inference) algorithm.

Usage

fci(suffStat, indepTest, alpha, labels, p, skel.method = c("stable", "original", "stable.fast"), type = c("normal", "anytime", "adaptive"), fixedGaps = NULL, fixedEdges = NULL, NAdelete = TRUE, m.max = Inf, pdsep.max = Inf, rules = rep(TRUE, 10), doPdsep = TRUE, biCC = FALSE, conservative = FALSE, maj.rule = FALSE, numCores = 1, verbose = FALSE)

Arguments

suffStat sufficient statistics: A named list containing all necessary elements for the conditional independence decisions in the function indepTest.

indepTest a function for testing conditional independence. The function is internally called as indepTest(x,y,S,suffStat). and tests conditional independence of x and y given S. Here, x and y are variables, and S is a (possibly empty) vector of variables (all variables are denoted by their column numbers in the adjacency matrix). suffStat is a list with all relevant information, see above. The return value of indepTest() is the p-value of the test for conditional independence.

alpha numeric significance level (in (0, 1)) for the individual conditional independence tests.

labels (optional) character vector of variable (or “node”) names. Typically preferred to specifying p.

p (optional) number of variables (or nodes). May be specified if labels are not, in which case labels is set to 1:p.
skel.method character string specifying method; the default, "stable" provides an order-independent skeleton, see skeleton.

type character string specifying the version of the FCI algorithm to be used. By default, it is "normal", and so the normal FCI algorithm is called. If set to "anytime", the 'Anytime FCI' is called and m.max needs to be specified. If set to "adaptive", the 'Adaptive Anytime FCI' is called and m.max is not used. For more information, see Details.

fixedGaps logical matrix of dimension p*p. If entry [i,j] or [j,i] (or both) are TRUE, the edge i-j is removed before starting the algorithm. Therefore, this edge is guaranteed to be absent in the resulting graph.

fixedEdges logical matrix of dimension p*p. If entry [i,j] or [j,i] (or both) are TRUE, the edge i-j is never considered for removal. Therefore, this edge is guaranteed to be present in the resulting graph.

NAdelete If indepTest returns NA and this option is TRUE, the corresponding edge is deleted. If this option is FALSE, the edge is not deleted.

m.max Maximum size of the conditioning sets that are considered in the conditional independence tests.

pdsep.max Maximum size of Possible-D-SEP for which subsets are considered as conditioning sets in the conditional independence tests. If the nodes x and y are adjacent in the graph and the size of Possible-D-SEP(x\ y) is bigger than pdsep.max, the edge is simply left in the graph. Note that if pdsep.max is less than Inf, the final PAG may be a supergraph of the one computed with pdsep.max = Inf, because fewer tests may have been performed in the former.

rules Logical vector of length 10 indicating which rules should be used when directing edges. The order of the rules is taken from Zhang (2008).

doPdsep If TRUE, Possible-D-SEP is computed for all nodes, and all subsets of Possible-D-SEP are considered as conditioning sets in the conditional independence tests, if not defined otherwise in pdsep.max. If FALSE, Possible-D-SEP is not computed, so that the algorithm simplifies to the Modified PC algorithm of Spirtes, Glymour and Scheines (2000, p.84).

biCC If TRUE, only nodes on paths between nodes x and y are considered to be in Possible-D-SEP(x) when testing independence between x and y. Uses biconnected components, biConnComp from RBGL.

conservative Logical indicating if the unshielded triples should be checked for ambiguity the second time when v-structures are determined. For more information, see details.

maj.rule Logical indicating if the unshielded triples should be checked for ambiguity the second time when v-structures are determined using a majority rule idea, which is less strict than the standard conservative. For more information, see details.

numCores Specifies the number of cores to be used for parallel estimation of skeleton.

verbose If true, more detailed output is provided.

**Details**

This function is a generalization of the PC algorithm (see pc), in the sense that it allows arbitrarily many latent and selection variables. Under the assumption that the data are faithful to a DAG
that includes all latent and selection variables, the FCI algorithm (Fast Causal Inference algorithm) (Spirtes, Glymour and Scheines, 2000) estimates the Markov equivalence class of MAGs that describe the conditional independence relationships between the observed variables.

We estimate an equivalence class of maximal ancestral graphs (MAGs) instead of DAGs, since DAGs are not closed under marginalization and conditioning (Richardson and Spirtes, 2002).

An equivalence class of a MAG can be uniquely represented by a partial ancestral graph (PAG). A PAG contains the following types of edges: o-o, o-, o->, ->, <->, -. The bidirected edges come from hidden variables, and the undirected edges come from selection variables. The edges have the following interpretation: (i) there is an edge between x and y if and only if variables x and y are conditionally dependent given S for all sets S consisting of all selection variables and a subset of the observed variables; (ii) a tail on an edge means that this tail is present in all MAGs in the Markov equivalence class; (iii) an arrowhead on an edge means that this arrowhead is present in all MAGs in the Markov equivalence class; (iv) a o-edgemark means that there is at least one MAG in the Markov equivalence class where the edgemark is a tail, and at least one where the edgemark is an arrowhead. Information on the interpretation of edges in a MAG can be found in the references given below.

The first part of the FCI algorithm is analogous to the PC algorithm. It starts with a complete undirected graph and estimates an initial skeleton using skeleton(*,method="stable") which produces an initial order-independent skeleton, see skeleton for more details. All edges of this skeleton are of the form o-o. Due to the presence of hidden variables, it is no longer sufficient to consider only subsets of the neighborhoods of nodes x and y to decide whether the edge x o-o y should be removed. Therefore, the initial skeleton may contain some superfluous edges. These edges are removed in the next step of the algorithm which requires some orientations. Therefore, the v-structures are determined using the conservative method (see discussion on conservative below).

After the v-structures have been oriented, Possible-D-SEP sets for each node in the graph are computed at once. To decide whether edge x o-o y should be removed, one performs conditional independence tests of x and y given all possible subsets of Possible-D-SEP(x) and of Possible-D-SEP(y). The edge is removed if a conditional independence is found. This produces a fully order-independent final skeleton as explained in Colombo and Maathuis (2014). Subsequently, the v-structures are newly determined on the final skeleton (using information in sepset). Finally, as many as possible undetermined edge marks (o) are determined using (a subset of) the 10 orientation rules given by Zhang (2008).

The “Anytime FCI” algorithm was introduced by Spirtes (2001). It can be viewed as a modification of the FCI algorithm that only performs conditional independence tests up to and including order m.max when finding the initial skeleton, using the function skeleton, and the final skeleton, using the function pdsep. Thus, Anytime FCI performs fewer conditional independence tests than FCI. To use the Anytime algorithm, one sets type = "anytime" and needs to specify m.max, the maximum size of the conditioning sets.

The “Adaptive Anytime FCI” algorithm was introduced by Colombo et al (2012). The first part of the algorithm is identical to the normal FCI described above. But in the second part when the final skeleton is estimated using the function pdsep, the Adaptive Anytime FCI algorithm only performs conditional independence tests up to and including order m.max, where m.max is the maximum size of the conditioning sets that were considered to determine the initial skeleton using the function skeleton. Thus, m.max is chosen adaptively and does not have to be specified by the user.

Conservative versions of FCI, Anytime FCI, and Adaptive Anytime FCI are computed if conservative = TRUE is specified. After the final skeleton is computed, all potential v-structures a-b-c are checked
in the following way. We test whether \( a \) and \( c \) are independent conditioning on any subset of the neighbors of \( a \) or any subset of the neighbors of \( c \). When a subset makes \( a \) and \( c \) conditionally independent, we call it a separating set. If \( b \) is in no such separating set or in all such separating sets, no further action is taken and the normal version of the FCI, Anytime FCI, or Adaptive Anytime FCI algorithm is continued. If, however, \( b \) is in only some separating sets, the triple \( a-b-c \) is marked ‘ambiguous’. If \( a \) is independent of \( c \) given some \( S \) in the skeleton (i.e., the edge \( a-c \) dropped out), but \( a \) and \( c \) remain dependent given all subsets of neighbors of either \( a \) or \( c \), we will call all triples \( a-b-c \) ‘unambiguous’. This is because in the FCI algorithm, the true separating set might be outside the neighborhood of either \( a \) or \( c \). An ambiguous triple is not oriented as a v-structure. Furthermore, no further orientation rule that needs to know whether \( a-b-c \) is a v-structure or not is applied. Instead of using the conservative version, which is quite strict towards the v-structures, Colombo and Maathuis (2014) introduced a less strict version for the v-structures called majority rule. This adaptation can be called using \( \text{maj.bl} = \text{TRUE} \). In this case, the triple \( a-b-c \) is marked as ‘ambiguous’ if and only if \( b \) is in exactly 50 percent of such separating sets or no separating set was found. If \( b \) is in less than 50 percent of the separating sets it is set as a v-structure, and if in more than 50 percent it is set as a non v-structure (for more details see Colombo and Maathuis, 2014). Colombo and Maathuis (2014) showed that with both these modifications, the final skeleton and the decisions about the v-structures of the FCI algorithm are fully order-independent.

Note that the order-dependence issues on the 10 orientation rules are still present, see Colombo and Maathuis (2014) for more details.

Value

An object of class \text{fciAlgo} (see \text{fciAlgo}) containing the estimated graph (in the form of an adjacency matrix with various possible edge marks), the conditioning sets that lead to edge removals (sepset) and several other parameters.

Author(s)

Diego Colombo and Markus Kalisch (<kalisch@stat.math.ethz.ch>).

References


See Also

`fciPlus` for a more efficient variation of FCI; `skeleton` for estimating a skeleton using the PC algorithm; `pc` for estimating a CPDAG using the PC algorithm; `pdsep` for computing Possible-D-SEP for each node and testing and adapting the graph accordingly; `qreach` for a fast way of finding Possible-D-SEP for a given node.

`gaussCItest`, `disCItest`, `binCItest` and `dsepTest` as examples for `indepTest`.

Examples

```
# Example without latent variables
set.seed(42)
p <- 7
## generate and draw random DAG:
myDAG <- randomDAG(p, prob = 0.4)
## find skeleton and PAG using the FCI algorithm
suffStat <- list(C = cov2cor(trueCov(myDAG)), n = 10^9)
res <- fci(suffStat, indepTest=gaussCItest,
           alpha = 0.9999, p=p, doPdsep = FALSE)
```

```
# Example with hidden variables
# Zhang (2008), Fig. 6, p.1882

# create the graph g
p <- 4
L <- 1 # 'l' is latent
V <- c("Ghost", "Max","Urs","Anna","Eva")
edL <- setNames(vector("list", length=length(V)), V)
edL[[1]] <- list(edges=c(2,4),weights=c(1,1))
edL[[2]] <- list(edges=3,weights=c(1))
edL[[3]] <- list(edges=5,weights=c(1))
edL[[4]] <- list(edges=5,weights=c(1))
g <- new("graphNEL", nodes=V, edgeL=edL, edgemode="directed")

# compute the true covariance matrix of g
cov.mat <- trueCov(g)

# delete rows and columns belonging to latent variable L
true.cov <- cov.mat[-L,-L]
```
## transform covariance matrix into a correlation matrix
true.corr <- cov2cor(true.cov)

## The same, for the following three examples
indepTest <- gaussCItest
suffStat <- list(C = true.corr, n = 10^9)

## find PAG with FCI algorithm.
## As dependence "oracle", we use the true correlation matrix in
## gaussCItest() with a large "virtual sample size" and a large alpha:
normal.pag <- fci(suffStat, indepTest, alpha = 0.9999, labels = V[-L],
                  verbose=TRUE)

## find PAG with Anytime FCI algorithm with m.max = 1
## This means that only conditioning sets of size 0 and 1 are considered.
## As dependence "oracle", we use the true correlation matrix in the
## function gaussCItest with a large "virtual sample size" and a large
## alpha
anytime.pag <- fci(suffStat, indepTest, alpha = 0.9999, labels = V[-L],
                   type = "anytime", m.max = 1,
                   verbose=TRUE)

## find PAG with Adaptive Anytime FCI algorithm.
## This means that only conditioning sets up to size K are considered
## in estimating the final skeleton, where K is the maximal size of a
## conditioning set found while estimating the initial skeleton.
## As dependence "oracle", we use the true correlation matrix in the
## function gaussCItest with a large "virtual sample size" and a large
## alpha
adaptive.pag <- fci(suffStat, indepTest, alpha = 0.9999, labels = V[-L],
                    type = "adaptive",
                    verbose=TRUE)

## define PAG given in Zhang (2008), Fig. 6, p.1882
corr.pag <- rbind(c(0,1,1,0),
                  c(1,0,0,2),
                  c(1,0,0,2),
                  c(0,3,3,0))

## check if estimated and correct PAG are in agreement
all(corr.pag == normal.pag @ amat) # TRUE
all(corr.pag == anytime.pag @ amat) # FALSE
all(corr.pag == adaptive.pag@ amat) # TRUE
ij <- rbind(cbind(1:4,1:4), c(2,3), c(3,2))
all(corr.pag[ij] == anytime.pag @ amat[ij]) # TRUE
**fciAlgo-class**

**Class "fciAlgo" of FCI Algorithm Results**

**Description**

This class of objects is returned by functions `fci()`, `rfci()`, `fciPlus`, and `dag2pag` and represent the estimated PAG (and sometimes properties of the algorithm). Objects of this class have methods for the functions `plot`, `show` and `summary`.

**Usage**

```r
## S4 method for signature 'fciAlgo'
show(object)
## S3 method for class 'fciAlgo'
print(x, amat = FALSE, zero.print = ".", ...)
## S4 method for signature 'fciAlgo'
summary(object, amat = TRUE, zero.print = ".", ...)
## S4 method for signature 'fciAlgo,ANY'
plot(x, y, main = NULL, ...)
```

**Arguments**

- `x, object` a "fciAlgo" object.
- `amat` logical indicating if the adjacency matrix should be shown (printed) as well.
- `zero.print` string for printing 0 (‘zero’) entries in the adjacency matrix.
- `y` (generic `plot()` argument; unused).
- `main` main title, not yet supported.
- `...` optional further arguments (passed from and to methods).

**Slots**

The slots `call, n, max.ord, n.edgetests, sepset, and pMax` are inherited from class "gAlgo", see there.

In addition, "fciAlgo" has slots

- `amat`: adjacency matrix; for the coding of the adjacency matrix see `amatType`
- `allPdsep`: a list: the ith entry of this list contains Possible D-SEP of node number i.
- `n.edgetestsPDSEP`: the number of new conditional independence tests (i.e., tests that were not done in the first part of the algorithm) that were performed while checking subsets of Possible D-SEP.
- `max.ordPDSEP`: an integer: the maximum size of the conditioning sets used in the new conditional independence that were performed when checking subsets of Possible D-SEP.
Estimates a Partial Ancestral Graph (PAG) from observational data, using the FCI+ (Fast Causal Inference) Algorithm.

Usage

```r
fciPlus(suffStat, indepTest, alpha, labels, p, verbose=TRUE)
```
Arguments

suffStat  
sufficient statistics: A named list containing all necessary elements for the conditional independence decisions in the function indepTest.

indepTest  
a function for testing conditional independence. The function is internally called as indepTest(x, y, S, suffStat), and tests conditional independence of x and y given S. Here, x and y are variables, and S is a (possibly empty) vector of variables (all variables are denoted by their column numbers in the adjacency matrix). suffStat is a list with all relevant information, see above. The return value of indepTest() is the p-value of the test for conditional independence.

alpha  
numeric significance level (in (0, 1)) for the individual conditional independence tests.

labels  
(optional) character vector of variable (or “node”) names. Typically preferred to specifying p.

p  
(optional) number of variables (or nodes). May be specified if labels are not, in which case labels is set to 1:p.

verbose  
logical indicating if progress of the algorithm should be printed. The default is true, which used to be hard coded previously.

Details

A variation of FCI (Fast Causal Inference). For details, please see the references, and also fci.

Value

An object of class fciAlgo (see fciAlgo) containing the estimated graph (in the form of an adjacency matrix with various possible edge marks), the conditioning sets that lead to edge removals (sepset) and several other parameters.

Author(s)

Emilija Perkovic and Markus Kalisch (<kalisch@stat.math.ethz.ch>).

References


See Also

fci for estimating a PAG using the FCI algorithm.

Examples

```
### Example without latent variables
### generate a random DAG ( p = 7 )
```
set.seed(42)
p <- 7
myDAG <- randomDAG(p, prob = 0.4)

## find PAG using the FCI+ algorithm on "Oracle"
suffStat <- list(C = cov2cor(trueCov(myDAG)), n = 10^9)
m.fci <- fciPlus(suffStat, indepTest = gaussCItest,
                 alpha = 0.9999, p = p)
summary(m.fci)

## require("Rgraphviz")
sfsmisc::mult.fig(2, main="True DAG // fciPlus(.) "oracle" estimate")
plot(myDAG)
plot(m.fci)

---

#### find.unsh.triple

Find all Unshielded Triples in an Undirected Graph

**Description**

Find all unshielded triples in an undirected graph, \( q \), i.e., the ordered \((x, y, z)\) with \( x < z \) list of all the triples in the graph.

**Usage**

```r
find.unsh.triple(g, check=TRUE)
```

**Arguments**

- **g**: adjacency matrix of type `amat.cpdag` representing the skeleton; since a skeleton consists only of undirected edges, `g` must be symmetric.
- **check**: logical indicating that the symmetry of `g` should be checked.

**Details**

A triple of nodes \( x, y, \) and \( z \) is “unshielded”, if (all of these are true):

(i) \( x \) and \( y \) are connected;

(ii) \( y \) and \( z \) are connected;

(iii) \( x \) and \( z \) are **not** connected.

**Value**

- `unshTripl`: Matrix with 3 rows containing in each column an unshielded triple
- `unshVect`: Vector containing the unique number for each column in `unshTripl` (for internal use only)
Author(s)

Diego Colombo, Markus Kalisch (<kalisch@stat.math.ethz.ch>), and Martin Maechler

Examples

data(gmG)
if (require(Rgraphviz)) {
  ## show graph
  plot(gmG$g, main = "True DAG")
}

## prepare skeleton use in example
g <- wgtMatrix(gmG$g)  ## compute weight matrix
g <- 1*(g != 0)  # wgts --> 0/1; still lower triangular
print.table(g, zero.print=".")
skel <- g + t(g)  ## adjacency matrix of skeleton

## estimate unshielded triples -- there are 13 :
(uTr <- find.unsh.triple(skel))

---

gac

Test If Set Satisfies Generalized Adjustment Criterion (GAC)

Description

This function tests if z satisfies the Generalized Adjustment Criterion (GAC) relative to \((x, y)\) in the graph represented by adjacency matrix amat and interpreted as type (DAG, maximal PDAG, CPDAG, MAG, PAG). If yes, z can be used in covariate adjustment for estimating causal effects of x on y.

Usage

gac(amat, x, y, z, type = "pag")

Arguments

amat

adjacency matrix of type amat.cpdag or amat.pag

x, y, z

(integer) positions of variables in x, y or z in the adjacency matrix. x, y and z can be vectors representing several nodes.

-type

string specifying the type of graph of the adjacency matrix amat. It can be a DAG (type="dag"), a PDAG (type="pdag") or a CPDAG (type="cpdag"); then the type of the adjacency matrix is assumed to be amat.cpdag. It can also be a MAG (type="mag"), or a PAG (type="pag"); then the type of the adjacency matrix is assumed to be amat.pag.
Details

This work is a generalization of the work of Shpitser et al. (2012) (necessary and sufficient criterion in DAGs/ADMGs) and van der Zander et al. (2014) (necessary and sufficient criterion in MAGs). Moreover, it is a generalization of the Generalized Backdoor Criterion (GBC) of Maathuis and Colombo (2013): While GBC is sufficient but not necessary, GAC is both sufficient and necessary for DAGs, CPDAGs, MAGs and PAGs. For more details see Perkovic et al. (2015, 2017a, 2017b).

The motivation to find a set \( z \) that satisfies the GAC with respect to \((x, y)\) is the following:

A set of variables \( z \) satisfies the GAC relative to \((x, y)\) in the given graph, if and only if the causal effect of \( x \) on \( y \) is identifiable by covariate adjustment and is given by

\[
P(Y|\text{do}(X = x)) = \sum_Z P(Y|X,Z) \cdot P(Z),
\]

(for any joint distribution “compatible” with the graph; the formula is for discrete variables with straightforward modifications for continuous variables). This result allows to write post-intervention densities (the one written using Pearl’s do-calculus) using only observational densities estimated from the data.

For \( z \) to satisfy the GAC relative to \((x, y)\) and the graph, the following three conditions must hold:

(0) The graph is adjustment amenable relative to \((x, y)\).

(1) The intersection of \( z \) and the forbidden set (explained in Perkovic et al. (2015, 2017b) is empty.

(2) All proper definite status non-causal paths in the graph from \( x \) to \( y \) are blocked by \( z \).

It is important to note that there can be \( x \) and \( y \) for which there is no set \( Z \) that satisfies the GAC, but the total causal effect might be identifiable via some technique other than covariate adjustment.

For details on the GAC for DAGs, CPDAGs, PAGs see Perkovic et. al (2015,2017a). For details on the GAC for MAGs see van der Zander et. al (2014) and for details on the GAC for maximal PDAGs see Perkovic et. al (2017b).

For the coding of the adjacency matrix see amatType. The input matrix can either be of class matrix or of class amat.

Value

A list with three components:

- gac logical; TRUE if \( z \) satisfies the GAC relative to \((x, y)\) in the graph represented by amat and type
- res logical vector of length three indicating if each of the three conditions (0), (1) and (2) are true
- f node positions of nodes in the forbidden set (see Perkovic et al. (2015, 2017b)

Author(s)

Emilija Perkovic and Markus Kalisch (kalisch@stat.math.ethz.ch)
gac

References


See Also

backdoor for the Generalized Backdoor Criterion, pc for estimating a CPDAG and fci and fciPlus for estimating a PAG.

Examples

## We reproduce the four examples in Perkovic et. al (2015, 2017a)

### Example 4.1 in Perkovic et. al (2015), Example 2 in Perkovic et. al (2017a)

mFig1 <- matrix(c(0,1,1,0,0,0, 1,0,1,1,1,0, 0,0,0,0,0,1, 0,1,1,0,1,0, 0,0,0,0,0,0), 6,6)

type <- "cpdag"
x <- 3; y <- 6

## Z satisfies GAC :
gac(mFig1, x,y, z=c(2,4), type)
gac(mFig1, x,y, z=c(4,5), type)
gac(mFig1, x,y, z=c(4,2,1), type)
gac(mFig1, x,y, z=c(4,5,1), type)
gac(mFig1, x,y, z=c(4,2,5), type)
gac(mFig1, x,y, z=c(4,2,5,1), type)

### Z does not satisfy GAC :
gac(mFig1,x,y, z=2, type)
gac(mFig1,x,y, z=NULL, type)

### Example 4.2 in Perkovic et. al (2015), Example 3 in Perkovic et. al (2017a)

mFig3a <- matrix(c(0,1,0,0, 1,0,1,1, 0,1,0,1, 0,1,1,0), 4,4)
mFig3b <- matrix(c(0,2,0,0, 3,0,3,3, 0,2,0,3, 0,2,2,0), 4,4)
mFig3c <- matrix(c(0,3,0,0, 2,0,3,3, 0,2,0,3, 0,2,2,0), 4,4)
type <- "pag"
x <- 2; y <- 4
## Z does not satisfy GAC
gac(mFig3a,x,y, z=NULL, type) ## not amenable rel. to (X,Y)
gac(mFig3b,x,y, z=NULL, type) ## not amenable rel. to (X,Y)
## Z satisfies GAC
gac(mFig3c,x,y, z=NULL, type) ## amenable rel. to (X,Y)

##############################
## Example 4.3 in Perkovic et. al (2015), Example 4 in Perkovic et. al (2017a)
##############################
mFig4a <- matrix(c(0,0,1,0,0,0, 0,0,0,0,0,0, 2,2,0,3,3,2, 
0,0,2,0,2,2, 0,0,2,1,0,2, 0,0,1,3,3,0, 6,6)
mFig4b <- matrix(c(0,0,1,0,0,0, 0,0,0,0,0,0, 2,2,0,3,3,2, 
0,0,0,0,2,2, 0,0,2,3,0,2, 0,0,2,3,2,0, 6,6)
type <- "pag"
x <- 3; y <- 4
## both PAGs are amenable rel. to (X,Y)
## Z satisfies GAC in Fig. 4a
gac(mFig4a,x,y, z=6, type)
gac(mFig4a,x,y, z=c(1,6), type)
gac(mFig4a,x,y, z=c(2,6), type)
gac(mFig4a,x,y, z=c(1,2,6), type)
## no Z satisfies GAC in Fig. 4b
gac(mFig4b,x,y, z=NULL, type)
gac(mFig4b,x,y, z=6, type)
gac(mFig4b,x,y, z=c(5,6), type)

##############################
## Example 4.4 in Perkovic et. al (2015), Example 8 in Perkovic et. al (2017a)
##############################
mFig5a <- matrix(c(0,1,0,0,0, 1,0,1,0,0, 0,0,0,0,1, 0,0,1,0,0, 0,0,0,0), 5,5)
type <- "cpdag"
x <- c(1,5); y <- 4
## Z satisfies GAC
gac(mFig5a,x,y, z=c(2,3), type)
## Z does not satisfy GAC
gac(mFig5a,x,y, z=2, type)
mFig5b <- matrix(c(0,1,0,0,0,0, 2,0,2,3,0,3,0, 0,1,0,0,0,0, 0,2,0,3,0,0, 0,0,0,2,0,2,3, 0,2,0,2,0,0, 0,0,0,0,2,0,0), 7,7)
type <- "pag"
x<-c(2,7); y<-6
## Z satisfies GAC
gac(mFig5b,x,y, z=c(4,5), type)
gac(mFig5b,x,y, z=c(4,5,1), type)
gac(mFig5b,x,y, z=c(4,5,3), type)
gac(mFig5b,x,y, z=c(1,3,4,5), type)
## Z does not satisfy GAC
gac(mFig5b,x,y, z=NULL, type)

########################################################################
## Example 4.7 in Perkovic et. al (2017b)
```r
# Fig. 3a
mFig3a <- matrix(c(0,1,0,0, 1,0,1,1, 0,1,0,1, 0,1,1,0), 4,4)

# Fig. 3b
mFig3b <- matrix(c(0,1,0,0, 0,0,1,1, 0,0,0,1, 0,0,1,0), 4,4)

# Fig. 3c
mFig3c <- matrix(c(0,0,0,0, 1,0,1,0, 0,1,0,1, 0,1,1,0), 4,4)

type <- "pdag"
x <- 2; y <- 4

## Z does not satisfy GAC

# Not amenable rel. to (X,Y)
gac(mFig3a,x,y, z=NULL, type) # not amenable rel. to (X,Y)

gac(mFig3c,x,y, z=NULL, type) # amenable rel. to (X,Y), but no set can block X <- Y

## Z satisfies GAC

gac(mFig3b,x,y, z=NULL, type) # amenable rel. to (X,Y)
```

---

### Description

"gAlgo" is a "VIRTUAL" class, the common basis of classes "pcAlgo" and "fciAlgo".

We describe the common slots here; for more see the help pages of the specific classes.

### Slots

- **call**: a `call` object: the original function call.
- **n**: an "integer", the sample size used to estimate the graph.
- **max.ord**: an `integer`, the maximum size of the conditioning set used in the conditional independence tests of the (first part of the algorithm), in function `skeleton`.
- **n.edgetests**: the number of conditional independence tests performed by the (first part of the) algorithm.
- **sepset**: a `list`, the conditioning sets that led to edge deletions. The set that led to the removal of the edge `i -- j` is saved in either `sepset[[i]][[j]]` or in `sepset[[j]][[i]]`.
- **pMax**: a numeric square `matrix`, where the `(i,j)`th entry contains the maximal p-value of all conditional independence tests for edge `i -- j`.

### Author(s)

Martin Maechler

### See Also

"pcAlgo" and "fciAlgo".

### Examples

```r
showClass("gAlgo")
```
GaussL0penIntScore-class

Description

This class represents a score for causal inference from jointly interventional and observational Gaussian data; it is used in the causal inference functions \texttt{gies} and \texttt{simy}.

Details

The class implements an $\ell_0$-penalized Gaussian maximum likelihood estimator. The penalization is a constant (specified by the argument \texttt{lambda} in the constructor) times the number of parameters of the DAG model. By default, the constant $\lambda$ is chosen as $\log(n)/2$, which corresponds to the BIC score.

Extends

Class "\texttt{Score}"\texttt{, directly.}
All reference classes extend and inherit methods from "\texttt{envRefClass}".

Fields

The class GaussL0penIntScore has the same fields as \texttt{Score}. They need not be accessed by the user.

Constructor

\begin{verbatim}
new("GaussL0penIntScore",
data = matrix(1, 1, 1),
targets = list(integer(0)),
target.index = rep(as.integer(1), nrow(data)),
lambda = 0.5*log(nrow(data)),
intercept = FALSE,
use.cpp = TRUE,
...)
\end{verbatim}

data Data matrix with \(n\) rows and \(p\) columns. Each row corresponds to one realization, either interventional or observational.

targets List of mutually exclusive intervention targets that have been used for data generation.

target.index Vector of length \(n\); the \(i\)-th entry specifies the index of the intervention target in targets under which the \(i\)-th row of data was measured.

lambda Penalization constant (cf. details)

intercept Indicates whether an intercept is allowed in the linear structural equations, or, equivalently, if a mean different from zero is allowed for the observational distribution.
Indicates whether the calculation of the score should be done by the C++ library of the package, which speeds up calculation. This parameter should only be set to FALSE in the case of problems.

Methods

local.score(vertex, parents, ...) Calculates the local score of a vertex and its parents. Since this score has no obvious interpretation, it is rather for internal use.

global.score.int(edges, ...) Calculates the global score of a DAG, represented as a list of in-edges: for each vertex in the DAG, this list contains a vector of parents.

global.score(dag, ...) Calculates the global score of a DAG, represented as an object of a class derived from ParDAG.

local.mle(vertex, parents, ...) Calculates the local MLE of a vertex and its parents. The result is a vector of parameters encoded as follows:

- First element: variance of the Gaussian error term
- Second element: intercept
- Following elements: regression coefficients; one per parent vertex

global.mle(dag, ...) Calculates the global MLE of a DAG, represented by an object of a class derived from ParDAG. The result is a list of vectors, one per vertex, each in the same format as the result vector of local.mle.

Author(s)

Alain Hauser (<alain.hauser@bfh.ch>)

See Also
gies, simy, GaussL0penObsScore, Score

Examples

```
############################################################
## Using Gaussian Data
############################################################
## Load predefined data
data(gmInt)

## Define the score object
score <- new("GaussL0penIntScore", gmInt$x, gmInt$targets, gmInt$target.index)

## Score of the true underlying DAG
score$global.score(as(gmInt$g, "GaussParDAG"))

## Score of the DAG that has only one edge from 1 to 2
A <- matrix(0, ncol(gmInt$x), ncol(gmInt$x))
A[1, 2] <- 1
score$global.score(as(A, "GaussParDAG"))
## (Note: this is lower than the score of the true DAG.)
```
This class represents a score for causal inference from observational Gaussian data; it is used in the causal inference function `ges`.

The class implements an $\ell_0$-penalized Gaussian maximum likelihood estimator. The penalization is a constant (specified by the argument `lambda` in the constructor) times the number of parameters of the DAG model. By default, the constant $\lambda$ is chosen as $\log(n)/2$, which corresponds to the BIC score.

All reference classes extend and inherit methods from `envRefClass`.

The class `GaussL0penObsScore` has the same fields as `Score`. They need not be accessed by the user.

The constructor is:

```r
new("GaussL0penObsScore",
    data = matrix(1, 1, 1),
    lambda = 0.5*log(nrow(data)),
    intercept = TRUE,
    use.cpp = TRUE,
    ...
)
```

- `data` Data matrix with $n$ rows and $p$ columns. Each row corresponds to one observational realization.
- `lambda` Penalization constant (cf. details)
- `intercept` Indicates whether an intercept is allowed in the linear structural equations, or, equivalently, if a mean different from zero is allowed for the observational distribution.
- `use.cpp` Indicates whether the calculation of the score should be done by the C++ library of the package, which speeds up calculation. This parameter should only be set to `FALSE` in the case of problems.
Methods

local.score(vertex, parents, ...) Calculates the local score of a vertex and its parents. Since this score has no obvious interpretation, it is rather for internal use.

global.score.int(edges, ...) Calculates the global score of a DAG, represented as a list of in-edges: for each vertex in the DAG, this list contains a vector of parents.

global.score(dag, ...) Calculates the global score of a DAG, represented as an object of a class derived from ParDAG.

local.mle(vertex, parents, ...) Calculates the local MLE of a vertex and its parents. The result is a vector of parameters encoded as follows:
  • First element: variance of the Gaussian error term
  • Second element: intercept
  • Following elements: regression coefficients; one per parent vertex

global.mle(dag, ...) Calculates the global MLE of a DAG, represented by an object of a class derived from ParDAG. The result is a list of vectors, one per vertex, each in the same format as the result vector of local.mle.

Author(s)

Alain Hauser (<alain.hauser@bfh.ch>)

See Also

ges, GaussL0penIntScore, Score

Examples

#==============================================================================
## Using Gaussian Data
#==============================================================================
## Load predefined data
data(gmG)

## Define the score object
score <- new("GaussL0penObsScore", gmG$x)

## Score of the true underlying DAG
score$global.score(as(gmG$g, "GaussParDAG"))

## Score of the DAG that has only one edge from 1 to 2
A <- matrix(0, ncol(gmG$x), ncol(gmG$x))
A[1, 2] <- 1
score$global.score(as(A, "GaussParDAG"))
## (Note: this is lower than the score of the true DAG.)
GaussParDAG-class

Class "GaussParDAG" of Gaussian Causal Models

Description

The "GaussParDAG" class represents a Gaussian causal model.

Details

The class "GaussParDAG" is used to simulate observational and/or interventional data from Gaussian causal models as well as for parameter estimation (maximum-likelihood estimation) for a given DAG structure in the presence of a data set with jointly observational and interventional data.

A Gaussian causal model can be represented as a set of \( p \) linear structural equations with Gaussian noise variables. Those equations are fully specified by indicating the regression parameters, the intercept and the variance of the noise or error terms. More details can be found e.g. in Kalisch and Bühlmann (2007) or Hauser and Bühlmann (2012).

Extends

Class "ParDAG", directly.

All reference classes extend and inherit methods from "envRefClass".

Constructor

new("GaussParDAG", nodes, in.edges, params)

- nodes: Vector of node names; cf. also field .nodes.
- in.edges: A list of length \( p \) consisting of index vectors indicating the edges pointing into the nodes of the DAG.
- params: A list of length \( p \) consisting of parameter vectors modeling the conditional distribution of a node given its parents; cf. also field .params for the meaning of the parameters.

Fields

- .nodes: Vector of node names; defaults to as.character(1:p), where \( p \) denotes the number of nodes (variables) of the model.
- .in.edges: A list of length \( p \) consisting of index vectors indicating the edges pointing into the nodes of the DAG. The \( i \)-th entry lists the indices of the parents of the \( i \)-th node.
- .params: A list of length \( p \) consisting of parameter vectors modeling the conditional distribution of a node given its parents. The \( i \)-th entry models the conditional (normal) distribution of the \( i \)-th variable in the model given its parents. It is a vector of length \( k + 2 \), where \( k \) is the number of parents of node \( i \); the first entry encodes the error variance of node \( i \), the second entry the intercept, and the remaining entries the regression coefficients (see above). In most cases, it is easier to access the parameters via the wrapper functions err.var, intercept and weight.mat.
Class-Based Methods

set.err.var(value): Sets the error variances. The argument must be a vector of length \( p \), where \( p \) denotes the number of nodes in the model.

err.var(): Yields the vector of error variances.

intercept(): Yields the vector of intercepts.

set.intercept(value): Sets the intercepts. The argument must be a vector of length \( p \), where \( p \) denotes the number of nodes in the model.

weight.mat(target): Yields the (observational or interventional) weight matrix of the model. The weight matrix is an \( p \times p \) matrix whose \( i \)-th columns contains the regression coefficients of the \( i \)-th structural equation, if node \( i \) is not intervened (i.e., if \( i \) is not contained in the vector \( \text{target} \)), and is empty otherwise.

cov.mat(target, ivent.var): Yields the covariance matrix of the observational or an interventional distribution of the causal model. If \( \text{target} \) has length 0, the covariance matrix of the observational distribution is returned; otherwise \( \text{target} \) is a vector of the intervened nodes, and \( \text{ivent.var} \) is a vector of the same length indicating the variances of the intervention levels. Deterministic interventions with fix intervention levels would correspond to vanishing intervention variances; with non-zero intervention variances, stochastic interventions are considered in which intervention values are realizations of Gaussian variables (Korb et al., 2004).

The following methods are inherited (from the corresponding class): node.count ("ParDAG"), edge.count ("ParDAG"), simulate ("ParDAG")

Author(s)

Alain Hauser (<alain.hauser@bfh.ch>)

References


See Also

ParDAG

Examples

```r
set.seed(307)
myDAG <- r.gauss.pardag(p = 5, prob = 0.4)
(wm <- myDAG$weight.mat())
m <- as(myDAG, "matrix") # TRUE/FALSE adjacency matrix
```
symnum(m)
stopifnot(identical(unname(m),
                unname(wm != 0)))
myDAG$err.var()
myDAG$intercept()
myDAG$set.intercept(runif(5, min=3, max=4))
myDAG$intercept()
if (require(Rgraphviz)) plot(myDAG)

---

gds  

**Greedy DAG Search to Estimate Markov Equivalence Class of DAG**

**Description**

Estimate the observational or interventional essential graph representing the Markov equivalence class of a DAG by greedily optimizing a score function in the space of DAGs. In practice, greedy search should always be done in the space of equivalence classes instead of DAGs, giving the functions `gies` or `ges` the preference over `gds`.

**Usage**

```r

gds(score, labels = score$getNodes(), targets = score$getTargets(),
     fixedGaps = NULL, phase = c("forward", "backward", "turning"),
     iterate = length(phase) > 1, turning = TRUE, maxDegree = integer(0),
     verbose = FALSE, ...)
```

**Arguments**

- `score`: An instance of a class derived from `Score`.
- `labels`: Node labels; by default, they are determined from the scoring object.
- `targets`: A list of intervention targets (cf. details). A list of vectors, each vector listing the vertices of one intervention target.
- `fixedGaps`: Logical symmetric matrix of dimension p*p. If entry [i,j] is TRUE, the result is guaranteed to have no edge between nodes i and j.
- `phase`: Character vector listing the phases that should be used; possible values: `forward`, `backward`, and `turning` (cf. details).
- `iterate`: Logical indicating whether the phases listed in the argument `phase` should be iterated more than once (iterate = TRUE) or not.
- `turning`: Setting `turning = TRUE` is equivalent to setting `phases = c("forward","backward")` and `iterate = FALSE`; the use of the argument `turning` is deprecated.
- `maxDegree`: Parameter used to limit the vertex degree of the estimated graph. Valid arguments:
  1. Vector of length 0 (default): vertex degree is not limited.
  2. Real number r, 0 < r < 1: degree of vertex v is limited to r · n_v, where n_v denotes the number of data points where v was not intervened.
3. Single integer: uniform bound of vertex degree for all vertices of the graph.
4. Integer vector of length \( p \): vector of individual bounds for the vertex degrees.

verbose if \( \text{TRUE} \), detailed output is provided.

\[ \ldots \] additional arguments for debugging purposes and fine tuning.

Details

This function estimates the observational or interventional Markov equivalence class of a DAG based on a data sample with interventional data originating from various interventions and possibly observational data. The intervention targets used for data generation must be specified by the argument `targets` as a list of (integer) vectors listing the intervened vertices; observational data is specified by an empty set, i.e., a vector of the form `integer(0)`. As an example, if data contains observational samples as well as samples originating from an intervention at vertices 1 and 4, the intervention targets must be specified as `list(integer(0), as.integer(1), as.integer(c(1,4)))`.

An interventional Markov equivalence class of DAGs can be uniquely represented by a partially directed graph called interventional essential graph. Its edges have the following interpretation:

1. A directed edge \( a \rightarrow b \) stands for an arrow that has the same orientation in all representatives of the interventional Markov equivalence class;
2. An undirected edge \( a – b \) stands for an arrow that is oriented in one way in some representatives of the equivalence class and in the other way in other representatives of the equivalence class.

Note that when plotting the object, undirected and bidirected edges are equivalent.

Greedy DAG search (GDS) maximizes a score function (typically the BIC, passed to the function via the argument `score`) of a DAG in three phases, starting from the empty DAG:

**Forward phase** In the forward phase, GDS adds single arrows to the DAG as long as this augments the score.

**Backward phase** In the backward phase, the algorithm removes arrows from the DAG as long as this augments the score.

**Turning phase** In the turning phase, the algorithm reverts arrows of the DAG as long as this augments the score.

The phases that are actually run are specified with the argument `phase`. GDS cycles through the specified phases until no augmentation of the score is possible any more if `iterate = \text{TRUE}`. In the end, `gds` returns the (interventional or observational) essential graph of the last visited DAG.

It is well-known that a greedy search in the space of DAGs instead of essential graphs is more prone to be stuck in local optima of the score function and hence expected to yield worse estimation results than GIES (function `gies`) or GES (function `ges`) (Chickering, 2002; Hauser and Bühlmann, 2012). The function `gds` is therefore not of practical use, but can be used to compare causal inference algorithms to an elementary and straightforward approach.

Value

`gds` returns a list with the following two components:

`essgraph` An object of class `EssGraph` containing an estimate of the equivalence class of the underlying DAG.
An object of a class derived from `ParDAG` containing a (random) representative of the estimated equivalence class.

**Author(s)**

Alain Hauser (<alain.hauser@bfh.ch>)

**References**


**See Also**

`gies, ges, Score, EssGraph`

**Examples**

```r
## Load predefined data
data(gmInt)

## Define the score (BIC)
score <- new("GaussL0penIntScore", gmInt$x, gmInt$targets, gmInt$target.index)

## Estimate the essential graph
gds.fit <- gds(score)

## Plot the estimated essential graph and the true DAG
if (require(Rgraphviz)) {
  par(mfrow=c(1,2))
  plot(gds.fit$essgraph, main = "Estimated ess. graph")
  plot(gmInt$g, main = "True DAG")
}
```

### Description

Estimate the observational essential graph representing the Markov equivalence class of a DAG using the greedy equivalence search (GES) algorithm of Chickering (2002).
Usage

ges(score, labels = score$getNodes(),
      fixedGaps = NULL, adaptive = c("none", "vstructures", "triples"),
      phase = c("forward", "backward", "turning"), iterate = length(phase) > 1,
      turning = NULL, maxDegree = integer(0), verbose = FALSE, ...)

Arguments

score An instance of a class derived from Score which only accounts for observational data. If the dataset is high-dimensional (p>=n) ges might not be able to terminate.

labels Node labels; by default, they are determined from the scoring object.

fixedGaps logical symmetric matrix of dimension p*p. If entry [i,j] is TRUE, the result is guaranteed to have no edge between nodes i and j.

adaptive indicating whether constraints should be adapted to newly detected v-structures or unshielded triples (cf. details).

phase Character vector listing the phases that should be used; possible values: forward, backward, and turning (cf. details).

iterate Logical indicating whether the phases listed in the argument phase should be iterated more than once (iterate = TRUE) or not.

turning Setting turning = TRUE is equivalent to setting phases = c("forward", "backward") and iterate = FALSE; the use of the argument turning is deprecated.

maxDegree Parameter used to limit the vertex degree of the estimated graph. Valid arguments:

1. Vector of length 0 (default): vertex degree is not limited.
2. Real number r, 0 < r < 1: degree of vertex v is limited to r · n_v, where n_v denotes the number of data points where v was not intervened.
3. Single integer: uniform bound of vertex degree for all vertices of the graph.
4. Integer vector of length p: vector of individual bounds for the vertex degrees.

verbose If TRUE, detailed output is provided.

... Additional arguments for debugging purposes and fine tuning.

Details

Under the assumption that the distribution of the observed variables is faithful to a DAG, this function estimates the Markov equivalence class of the DAG. It does not estimate the DAG itself, because this is typically impossible (even with an infinite amount of data): different DAGs (forming a Markov equivalence class) can describe the same conditional independence relationships and be statistically indistinguishable from observational data alone.

All DAGs in an equivalence class have the same skeleton (i.e., the same adjacency information) and the same v-structures (i.e., the same induced subgraphs of the form a → b ← c). However, the direction of some edges may be undetermined, in the sense that they point one way in one DAG in the equivalence class, while they point the other way in another DAG in the equivalence class.
An equivalence class can be uniquely represented by a partially directed graph called (observational) essential graph or CPDAG (completed partially directed acyclic graph). Its edges have the following interpretation:

1. a directed edge $a \rightarrow b$ stands for an arrow that has the same orientation in all representatives of the Markov equivalence class;
2. an undirected edge $a - b$ stands for an arrow that is oriented in one way in some representatives of the equivalence class and in the other way in other representatives of the equivalence class.

Note that when plotting the object, undirected and bidirected edges are equivalent.

GES (greedy equivalence search) is a score-based algorithm that greedily maximizes a score function (typically the BIC, passed to the function via the argument score) in the space of (observational) essential graphs in three phases, starting from the empty graph:

**Forward phase** In the forward phase, GES moves through the space of essential graphs in steps that correspond to the addition of a single edge in the space of DAGs; the phase is aborted as soon as the score cannot be augmented any more.

**Backward phase** In the backward phase, the algorithm performs moves that correspond to the removal of a single edge in the space of DAGs until the score cannot be augmented any more.

**Turning phase** In the turning phase, the algorithm performs moves that correspond to the reversal of a single arrow in the space of DAGs until the score cannot be augmented any more.

GES cycles through these three phases until no augmentation of the score is possible any more if `iterate = TRUE`. Note that the turning phase was not part of the original implementation of Chickering (2002), but was introduced by Hauser and Bühlmann (2012) and shown to improve the overall estimation performance. The original algorithm of Chickering (2002) is reproduced with `phase = c("forward","backward")` and `iterate = FALSE`.

GES has the same purpose as the PC algorithm (see `pc`). While the PC algorithm is based on conditional independence tests (requiring the choice of an independence test and a significance level, see `pc`), the GES algorithm is a score-based method (requiring the choice of a score function) and does not depend on conditional independence tests. Since GES always operates in the space of essential graphs, it returns a valid essential graph (or CPDAG) in any case.

Using the argument `fixedGaps`, one can make sure that certain edges will not be present in the resulting essential graph: if the entry $[i,j]$ of the matrix passed to `fixedGaps` is `TRUE`, there will be no edge between nodes $i$ and $j$. Using this argument can speed up the execution of GIES and allows the user to account for previous knowledge or other constraints. The argument `adaptive` can be used to relax the constraints encoded by `fixedGaps` according to a modification of GES called ARGES (adaptively restricted greedy equivalence search) which has been presented in Nandy, Hauser and Maathuis (2015):

- When `adaptive = "vstructures"` and the algorithm introduces a new v-structure $a \rightarrow b \leftarrow c$ in the forward phase, then the edge $a - c$ is removed from the list of fixed gaps, meaning that the insertion of an edge between $a$ and $c$ becomes possible even if it was forbidden by the initial matrix passed to `fixedGaps`.
- When `adaptive = "triples"` and the algorithm introduces a new unshielded triple in the forward phase (i.e., a subgraph of three nodes $a$, $b$ and $c$ where $a$ and $b$ as well as $b$ and $c$ are adjacent, but $a$ and $c$ are not), then the edge $a - c$ is removed from the list of fixed gaps.
With one of the adaptive modifications, the successive application of a skeleton estimation method and GES restricted to an estimated skeleton still gives a consistent estimator of the DAG, which is not the case without the adaptive modification.

**Value**

`ges` returns a list with the following two components:

- `essgraph`: An object of class `EssGraph` containing an estimate of the equivalence class of the underlying DAG.
- `repr`: An object of a class derived from `ParDAG` containing a (random) representative of the estimated equivalence class.

**Author(s)**

Alain Hauser (<alain.hauser@bfh.ch>)

**References**


**See Also**

`pc`, `Score`, `EssGraph`

**Examples**

```r
## Load predefined data
data(gmG)

## Define the score (BIC)
score <- new("GaussL0penObsScore", gmG8$x)

ges.fit <- ges(score)

## Plot the estimated essential graph and the true DAG
if (require(Rgraphviz)) {
  par(mfrow=c(1,2))
  plot(ges.fit$essgraph, main = "Estimated CPDAG")
  plot(gmG8$g, main = "True DAG")
} else {
  # alternative:
```
getGraph

Get the "graph" Part or Aspect of R Object

Description
Get the graph part or “aspect” of an R object, notably from our pc(), skeleton(), fci(), etc, results.

Usage
getGraph(x)

Arguments
x potentially any R object which can be interpreted as a graph (with nodes and edges).

Value
a graph object, i.e., one inheriting from (the virtual) class "graph", package graph.

Methods
signature(x = "ANY") the default method just tries as(x,"graph"), so works when a coerce (S4) method is defined for x.
signature(x = "pcAlgo") and
signature(x = "fciAlgo") extract the graph part explicitly.
signature(x = "matrix") interpret x as adjacency matrix and return the corresponding "graphAM" object.
For sparseMatrix methods, see the ‘Note’.

Note
For large graphs, it may be attractive to work with sparse matrices from the Matrix package. If desired, you can activate this by

require(Matrix)
setMethod("getGraph", "sparseMatrix", function(x) as(x, "graphNEL"))
setMethod("getGraph", "Matrix", function(x) as(x, "graphAM"))

Author(s)
Martin Maechler
getNextSet

See Also

fci, etc. The graph class description in package graph.

Examples

```r
A <- rbind(c(0,1,0,0,1),
           c(0,0,0,1,1),
           c(1,0,0,1,0),
           c(1,0,0,0,1),
           c(0,0,0,1,0))
sum(A) # 9
getGraph(A) ## a graph with 5 nodes and 'sum(A)' edges
```

### Description

Iteration through a list of all combinations of choose(n,k)

### Usage

```r
getNextSet(n,k,set)
```

### Arguments

- `n`: Number of elements to choose from (integer)
- `k`: Size of chosen set (integer)
- `set`: Previous set in list (numeric vector)

### Details

The initial set is 1:k. Last index varies quickest. Using the dynamic creation of sets reduces the memory demands dramatically for large sets. If complete lists of combination sets have to be produced and memory is no problem, the function `combn` from package combinat is an alternative.

### Value

List with two elements:

- `nextSet`: Next set in list (numeric vector)
- `wasLast`: Logical indicating whether the end of the specified sequence is reached.

### Author(s)

Markus Kalisch <kalisch@stat.math.ethz.ch> and Martin Maechler
See Also

This function is used in skeleton.

Examples

```r
## start from first set (1,2) and get the next set of size 2 out of 1:5
## notice that res$wasLast is FALSE :
str(r <- getNextSet(5,2,c(1,2)))

## input is the last set; notice that res$wasLast now is TRUE:
str(r2 <- getNextSet(5,2,c(4,5)))

## Show all sets of size k out of 1:n :
## (if you really want this in practice, use something like combn() !)
n <- 5
k <- 3
currentSet <- 1:k
(res <- rbind(currentSet, deparse.level = 0))
repeat {
  newEl <- getNextSet(n,k,currentSet)
  if (newEl$wasLast)
    break
  ## otherwise continue:
  currentSet <- newEl$nextSet
  res <- rbind(res, currentSet, deparse.level = 0)
}
res
stopifnot(choose(n,k) == nrow(res)) ## must be identical
```

### gies

Estimate Interventional Markov Equivalence Class of a DAG by GIES

Description

Estimate the interventional essential graph representing the Markov equivalence class of a DAG using the greedy interventional equivalence search (GIES) algorithm of Hauser and Bühlmann (2012).

Usage

```r
gies(score, labels = score$getNodes(), targets = score$getTargets(),
      fixedGaps = NULL, adaptive = c("none", "vstructures", "triples"),
      phase = c("forward", "backward", "turning"), iterate = length(phase) > 1,
      turning = NULL, maxDegree = integer(0), verbose = FALSE, ...)
```

Arguments

- `score`: An R object inheriting from `Score`.
- `labels`: Node labels; by default, they are determined from the scoring object.
targets: A list of intervention targets (cf. details). A list of vectors, each vector listing the vertices of one intervention target.

fixedGaps: Logical symmetric matrix of dimension p*p. If entry [i,j] is TRUE, the result is guaranteed to have no edge between nodes i and j.

adaptive: indicating whether constraints should be adapted to newly detected v-structures or unshielded triples (cf. details).

phase: Character vector listing the phases that should be used; possible values: forward, backward, and turning (cf. details).

iterate: Logical indicating whether the phases listed in the argument phase should be iterated more than once (iterate = TRUE) or not.

turning: Setting turning = TRUE is equivalent to setting phases = c("forward","backward") and iterate = FALSE; the use of the argument turning is deprecated.

maxDegree: Parameter used to limit the vertex degree of the estimated graph. Possible values:

1. Vector of length 0 (default): vertex degree is not limited.
2. Real number $r$, $0 < r < 1$: degree of vertex v is limited to $r \cdot n_v$, where $n_v$ denotes the number of data points where v was not intervened.
3. Single integer: uniform bound of vertex degree for all vertices of the graph.
4. Integer vector of length p: vector of individual bounds for the vertex degrees.

verbose: If TRUE, detailed output is provided.

Additional arguments for debugging purposes and fine tuning.

Details

This function estimates the interventional Markov equivalence class of a DAG based on a data sample with interventional data originating from various interventions and possibly observational data. The intervention targets used for data generation must be specified by the argument targets as a list of (integer) vectors listing the intervened vertices; observational data is specified by an empty set, i.e. a vector of the form integer(0). As an example, if data contains observational samples as well as samples originating from an intervention at vertices 1 and 4, the intervention targets must be specified as list(integer(0),as.integer(1),as.integer(c(1,4))).

An interventional Markov equivalence class of DAGs can be uniquely represented by a partially directed graph called interventional essential graph. Its edges have the following interpretation:

1. a directed edge $a \to b$ stands for an arrow that has the same orientation in all representatives of the interventional Markov equivalence class;
2. an undirected edge $a \leftrightarrow b$ stands for an arrow that is oriented in one way in some representatives of the equivalence class and in the other way in other representatives of the equivalence class.

Note that when plotting the object, undirected and bidirected edges are equivalent.

GIES (greedy interventional equivalence search) is a score-based algorithm that greedily maximizes a score function (typically the BIC, passed to the function via the argument score) in the space of interventional essential graphs in three phases, starting from the empty graph:
**Forward phase** In the forward phase, GIES moves through the space of interventional essential graphs in steps that correspond to the addition of a single edge in the space of DAGs; the phase is aborted as soon as the score cannot be augmented any more.

**Backward phase** In the backward phase, the algorithm performs moves that correspond to the removal of a single edge in the space of DAGs until the score cannot be augmented any more.

**Turning phase** In the turning phase, the algorithm performs moves that correspond to the reversal of a single arrow in the space of DAGs until the score cannot be augmented any more.

The phases that are actually run are specified with the argument `phase`. GIES cycles through the specified phases until no augmentation of the score is possible any more if `iterate = TRUE`. GIES is an interventional extension of the GES (greedy equivalence search) algorithm of Chickering (2002) which is limited to observational data and hence operates on the space of observational instead of interventional Markov equivalence classes.

Using the argument `fixedGaps`, one can make sure that certain edges will not be present in the resulting essential graph: if the entry `[i,j]` of the matrix passed to `fixedGaps` is `TRUE`, there will be no edge between nodes `i` and `j`. Using this argument can speed up the execution of GIES and allows the user to account for previous knowledge or other constraints. The argument `adaptive` can be used to relax the constraints encoded by `fixedGaps` as follows:

- When `adaptive = "vstructures"` and the algorithm introduces a new v-structure `a → b ← c` in the forward phase, then the edge `a − c` is removed from the list of fixed gaps, meaning that the insertion of an edge between `a` and `c` becomes possible even if it was forbidden by the initial matrix passed to `fixedGaps`.
- When `adaptive = "triples"` and the algorithm introduces a new unshielded triple in the forward phase (i.e., a subgraph of three nodes `a`, `b`, and `c` where `a` and `b` as well as `b` and `c` are adjacent, but `a` and `c` are not), then the edge `a − c` is removed from the list of fixed gaps.

This modifications of the forward phase of GIES are inspired by the analog modifications in the forward phase of GES, which makes the successive application of a skeleton estimation method and GES restricted to an estimated skeleton a consistent estimator of the DAG (cf. Nandy, Hauser and Maathuis, 2015).

**Value**

gies returns a list with the following two components:

- **essgraph**: An object of class `EssGraph` containing an estimate of the equivalence class of the underlying DAG.
- **repr**: An object of a class derived from `ParDAG` containing a (random) representative of the estimated equivalence class.

**Author(s)**

Alain Hauser (<alain.hauser@bfh.ch>)

**References**


See Also
ges, Score, EssGraph

Examples

```r
## Load predefined data
data(gmInt)

## Define the score (BIC)
score <- new("GaussL0penIntScore", gmInt$x, gmInt$targets, gmInt$target.index)

## Estimate the essential graph
gies.fit <- gies(score)

## Plot the estimated essential graph and the true DAG
if (require(Rgraphviz)) {
  par(mfrow=c(1,2))
  plot(gies.fit$essgraph, main = "Estimated ess. graph")
  plot(gmInt$g, main = "True DAG")
}
```

---

**gmB**

*Graphical Model 5-Dim Binary Example Data*

**Description**

This data set contains a matrix containing information on five binary variables (coded as 0/1) and the corresponding DAG model.

**Usage**

data(gmB)

**Format**

The format is a list of two components

- **x**: Int [1:5000, 1:5] 0 1 1 0 1 1 0 1 1 1 0 1 1 1 1 1 1...
- **g**: Formal class 'graphNEL' [package "graph"] with 6 slots
  - ..@ nodes : chr [1:5] "1" "2" "3" "4" "5"...
  - ..@ edgeL :List of 5
  - ......

---

**gmB**

*GM 5-Dim Binary Example Data*
Details

The data was generated using Tetrad in the following way. A random DAG on five nodes was generated; binary variables were assigned to each node; then conditional probability tables corresponding to the structure of the generated DAG were constructed. Finally, 5000 samples were drawn using the conditional probability tables.

Examples

data(gmB)
## maybe str(gmB) ; plot(gmB) ...

---

### gmD

**Graphical Model Discrete 5-Dim Example Data**

Description

This data set contains a matrix containing information on five discrete variables (levels are coded as numbers) and the corresponding DAG model.

Usage

data(gmD)

Format

A list of two components

- **x**: a data frame with 5 columns X1 .. X5 each coding a discrete variable (aka factor) with intervals [1:10000, 1:5] 2 2 1 1 2 2 0 2 0 ...
- **g**: Formal class `graphNEL` [package “graph”] with 6 slots
  - ..@ nodes : chr [1:5] “1” “2” “3” “4” ...
  - ..@ edgeL :List of 5

where x is the data matrix and g is the DAG from which the data were generated.

Details

The data was generated using Tetrad in the following way. A random DAG on five nodes was generated; discrete variables were assigned to each node (with 3, 2, 3, 4 and 2 levels); then conditional probability tables corresponding to the structure of the generated DAG were constructed. Finally, 10000 samples were drawn using the conditional probability tables.
Examples

```r
data(gmD)
str(gmD, max=1)
if(require("Rgraphviz"))
  plot(gmD$g, main = "gmD $ g --- the DAG of the gmD (10’000 x 5 discrete data)"
## >>> 1 --> 3 <-- 2 --> 4 --> 5
str(gmD$x)
## The number of unique values of each variable:
sapply(gmD$x, function(v) nlevels(as.factor(v)))
## X1 X2 X3 X4 X5
##  3  2  3  4  2
lapply(gmD$x, table) ## the (marginal) empirical distributions
## $X1
##     0  1  2
## 1933 3059 5008
##
## $X2
##     0  1
##  8008 1992
##
## $X3
## .....```

---

**gmG**

*Graphical Model 8-Dimensional Gaussian Example Data*

---

Description

These two data sets contain a matrix containing information on eight gaussian variables and the corresponding DAG model.

Usage

```r
data(gmG)
```

Format

gmG and gmG8 are each a list of two components

- **x**: a numeric matrix 5000 × 8.
- **g**: a graph, i.e., of formal `class "graphNEL"` from package `graph` with 6 slots
  - ..@ nodes : chr [1:8] "1" "2" "3" "4" ...
  - ..@ edgeL :List of 8
  - ........
Details

The data was generated as indicated below. First, a random DAG model was generated, then 5000 samples were drawn from “almost” this model, for gmG: In the previous version, the data generation wgtMatrix had the non-zero weights in reversed order for each node. On the other hand, for gmG8, the correct weights were used in all cases.

Source

The data set is identical to the one generated by

```r
## Used to generate "gmG"
set.seed(40)
p <- 8
n <- 5000
## true DAG:
vars <- c("Author", "Bar", "Ctrl", "Goal", paste0("V",5:8))
gGtrue <- randomDAG(p, prob = 0.3, V = vars)
gmG <- list(x = rmvDAG(n, gGtrue, back.compatible=TRUE), g = gGtrue)
gmG8 <- list(x = rmvDAG(n, gGtrue), g = gGtrue)
```

Examples

data(gmG)
str(gmG, max=3)
stopifnot(identical(gmG $ g, gmG8 $ g))
if(dev.interactive()) { ## to save time in tests
  round(as(gmG $ g, "Matrix"), 2) # weight ("adjacency") matrix
  plot(gmG $ g)
  pairs(gmG$x, gap = 0,
       panel=function(...)
          smoothScatter(..., add=TRUE))
}

Description

This data set contains a matrix containing information on seven gaussian variables and the corresponding DAG model.

Usage

data(gmI)
gmInt

Format
The two gmI+ objects are each a list of two components x, an \( n \times 7 \) numeric matrix, and g, a DAG, a graph generated by randomDAG.

See gmG for more

Details
The data was generated as indicated below. First, a random DAG was generated, then samples were drawn from this model, strictly speaking for gmI7 only.

Source
The data sets are identical to those generated by

```r
## Used to generate "gmI"
set.seed(123)
p <- 7
myDAG <- randomDAG(p, prob = 0.2) ## true DAG
gmI <- list(x = rmvDAG(10000, myDAG, back.compatible=TRUE), g = myDAG)
gmI7 <- list(x = rmvDAG(8000, myDAG), g = myDAG)
```

Examples
```r
data(gmI)
str(gmI, max=3)
stopifnot(identical(gmI $ g, gmI7 $ g))
if(dev.interactive()) { ## to save time in tests
  round(as(gmI $ g, "Matrix"), 2) # weight ("adjacency") matrix
  plot(gmI $ g)
  pairs(gmI$x, gap = 0,
        panel=function(...) smoothScatter(..., add=TRUE))
}
```

Description
This data set contains a matrix with an ensemble of observational and interventional data from eight Gaussian variables. The corresponding (data generating) DAG model is also stored.

Usage
data(gmInt)
The format is a list of four components:

- **x**: Matrix with 5000 rows (one row a measurement) and 8 columns (corresponding to the 8 variables)
- **targets**: List of (mutually exclusive) intervention targets. In this example, the three entries `integer(0)`, `3`, and `5` indicate that the data set consists of observational data, interventional data originating from an intervention at vertex `3`, and interventional data originating from an intervention at vertex `5`.
- **target.index**: Vector with 5000 elements. Each entry maps a row of `x` to the corresponding intervention target. Example: `gmInt$target.index[3322] == 2` means that `x[3322,]` was simulated from an intervention at `gmInt$targets[[2]]`, i.e. at vertex `3`.
- **g**: Formal class 'graphNEL' [package "graph"] with 6 slots, representing the true DAG from which observational and interventional data was sampled.

The data was generated as indicated below. First, a random DAG model was generated, then 5000 samples were drawn from this model: 3000 observational ones, and 1000 each from an intervention at vertex `3` and `5`, respectively (see `gmInt$target.index`).

The data set is identical to the one generated by

```r
set.seed(40)
p <- 8
n <- 5000
gGtrue <- randomDAG(p, prob = 0.3)
pardag <- as(gGtrue, "GaussParDAG")
pardag$set.err.var(rep(1, p))
targets <- list(integer(0), 3, 5)
target.index <- c(rep(1, 0.6*n), rep(2, n/5), rep(3, n/5))
x1 <- rmvnorm.ivent(0.6*n, pardag)
x2 <- rmvnorm.ivent(n/5, pardag, targets[[2]], matrix(rnorm(n/5, mean = 4, sd = 0.02), ncol = 1))
x3 <- rmvnorm.ivent(n/5, pardag, targets[[3]], matrix(rnorm(n/5, mean = 4, sd = 0.02), ncol = 1))
gmInt <- list(x = rbind(x1, x2, x3),
               targets = targets,
               target.index = target.index,
               g = gGtrue)
```

Examples

```r
data(gmInt)
str(gmInt, max = 3)
pairs(gmInt$x, gap = 0, pch = ".")
```
Description

This data set contains a matrix containing information on four gaussian variables and the corresponding DAG model containing four observed and one latent variable.

Usage

data(gmL)

Format

The format is a list of 2 components

x: $ x: num [1:10000, 1:4] 0.924 -0.189 1.016 0.363 0.497 ... ..- attr(*, "dimnames")=List of 2 .. ..$ : NULL .. ..$ : chr [1:4] "2" "3" "4" "5"

G: $ g:Formal class 'graphNEL' [package "graph"] with 6 slots .. ..@ nodes : chr [1:5] "1" "2" "3" "4" ... ..@ edgeL :List of 5 .......

Details

The data was generated as indicated below. First, a random DAG model was generated with five nodes; then 10000 samples were drawn from this model; finally, variable one was declared to be latent and the corresponding column was deleted from the simulated data set.

Source

## Used to generate "gmL"
set.seed(47)
p <- 5
n <- 10000
gGtrue <- randomDAG(p, prob = 0.3) ## true DAG
myX <- rmvDAG(n, gGtrue)
colnames(myX) <- as.character(1:5)
gmL <- list(x = myX[-1], g = gGtrue)

Examples

data(gmL)
str(gmL, max=3)

## the graph:
gmL$g
graph::nodes(gmL$g); str(graph::edges(gmL$g))
if(require("Rgraphviz"))
plot(gmL$g, main = "gmL $ g -- latent variable example data")
pairs(gmL$x) # the data

**ida**

*Estimate Multiset of Possible Joint Total Causal Effects*

**Description**

ida() estimates the multiset of possible joint total causal effects of variables (X) onto variables (Y) from observational data via adjustment.

**Usage**

ida(x.pos, y.pos, mcov, graphEst, method = c("local","optimal","global"),
y.notparent = FALSE, verbose = FALSE, all.dags = NA, type = c("cpdag", "pdag"))

**Arguments**

- **x.pos, x**: Positions of variables X in the covariance matrix.
- **y.pos, y**: Positions of variables Y in the covariance matrix.
- **mcov**: Covariance matrix that was used to estimate graphEst.
- **graphEst**: Estimated CPDAG or PDAG. The CPDAG is typically from pc(): If the result of pc is pc.fit, the estimated CPDAG can be obtained by pc.fit@graph. A PDAG can be obtained from the CPDAG by adding background knowledge using addBgKnowledge().
- **method**: Character string specifying the method with default "local".
  - "global": The algorithm considers all undirected edges in the CPDAG or PDAG, using the possible parents as adjustment sets to estimate all possible causal effects. It is hence slow and can only be applied to singleton X.
  - "local": The algorithm only considers edges in the neighborhood of X in the CPDAG or PDAG, using the possible parents as adjustment sets to estimate the unique possible causal effects. It is hence much faster than "global" and can only be applied to singleton X.
  - "optimal": The algorithm considers only those edges necessary to compute the possible optimal valid adjustment sets, using these as adjustment sets to estimate the unique possible causal effects. It is hence faster than the "global" option but also slower than "local". It provides more efficient estimates than both alternatives but is also more sensitive to faulty graph estimates. Can be applied to sets X.

See details below.

- **y.notparent**: Logical; for singleton X and Y. If true, any edge between X and Y is assumed to be of the form X->Y. Not implemented for the method="optimal"
- **verbose**: If TRUE, details on the regressions are printed.
All DAGs in the equivalence class represented by the CPDAG or PDAG can be precomputed by `pdag2allDags()` and passed via this argument. In that case, `pdag2allDags(.)` is not called internally. This option is only relevant when using `method="global"`.

**type**

Type of graph "graphEst"; can be of type "cpdag" or "pdag" (e.g. a CPDAG with background knowledge from Meek, 1995)

**Details**

It is assumed that we have observational data from a multivariate Gaussian distribution faithful to the true (but unknown) underlying causal DAG (without hidden variables). Under these assumptions, this function estimates the multiset of possible total joint effects of \( X \) on \( Y \). Here the total joint effect of \( X = (X_1, X_2) \) on \( Y \) is defined via Pearl’s do-calculus as the vector 

\[
\begin{align*}
&\left\{ E[Y|do(X_1 = x_1 + 1, X_2 = x_2)] - E[Y|do(X_1 = x_1, X_2 = x_2 + 1)],
&\quad E[Y|do(X_1 = x_1, X_2 = x_2)] - E[Y|do(X_1 = x_1, X_2 = x_2 + 1)] \right\},
\end{align*}
\]

with a similar definition for more than two variables. These values are equal to the partial derivatives (evaluated at \( x_1, x_2 \)) of \( E[Y|do(X = x_1', X_2 = x_2')] \) with respect to \( x_1' \) and \( x_2' \). Moreover, under the Gaussian assumption, these partial derivatives do not depend on the values at which they are evaluated.

We estimate a set of possible joint total causal effects instead of the unique joint total causal effect, since it is typically impossible to identify the latter when the true underlying causal DAG is unknown (even with an infinite amount of data). Conceptually, the method works as follows. First, we estimate the equivalence class of DAGs that describe the conditional independence relationships in the data, using the function `pc` (see the help file of this function). For each DAG \( G \) in the equivalence class, we apply Pearl’s do-calculus to estimate the total causal effect of \( X \) on \( Y \). This can be done via a simple linear regression adjusting for a valid adjustment set.

For example, if \( X \) and \( Y \) are singleton and \( Y \) is not a parent of \( X \), we can take the regression coefficient of \( X \) in the regression \( lm(Y \sim X + pa(X,G)) \), where \( pa(X,G) \) denotes the parents of \( X \) in the DAG \( G \); if \( Y \) is a parent of \( X \) in \( G \), we can set the estimated causal effect to zero.

If the equivalence class contains \( k \) DAGs, this will yield \( k \) estimated total causal effects. Since we do not know which DAG is the true causal DAG, we do not know which estimated possible total joint causal effect of \( X \) on \( Y \) is the correct one. Therefore, we return the entire multiset of \( k \) estimated effects (it is a multiset rather than a set because it can contain duplicate values).

One can take summary measures of the multiset. For example, the minimum absolute value provides a lower bound on the size of the true causal effect: If the minimum absolute value of all values in the multiset is larger than one, then we know that the size of the true causal effect (up to sampling error) must be larger than one.

If `method="global"`, the method as described above is carried out, where all DAGs in the equivalence class of the estimated CPDAG or PDAG `graphEst` are computed using the function `pdag2allDags`. The parent set for each DAG is then used to estimate the corresponding possible total causal effect. This method is suitable for small graphs (say, up to 10 nodes) and can only be used for singleton \( X \).

If `method="local"`, we only consider all valid possible directions of undirected edges that have \( X \) as an endpoint.

In the case of a CPDAG, we consider all possible directions of undirected edges that have \( X \) as an endpoint, such that no new v-structure is created. Maathuis, Kalisch and Buehlmann (2009) showed that there is at least one DAG in the equivalence class for each such local configuration. Hence, the procedure is truly local in this setting.
In the case of a PDAG, we need to verify for all possible directions whether they lead to an amenable max. PDAG if we apply Meek’s orientation rules. In this setting the complexity of the "local" method is similar to the "optimal" one and it is not truly local. For details see Section 4.2 in Perkovic, Kalisch and Maathuis (2017).

We estimate the total causal effect of \( X \) on \( Y \) for each valid configuration as above, using linear regression adjusting for the corresponding possible parents. As we adjust for the same sets as in the "global" method, it follows that the multisets of total causal effects of the two methods have the same unique values. They may, however, have different multiplicities.

Since the parents of \( X \) are usually an inefficient valid adjustment set we provide a third method, that uses different adjustment sets.

If method="optimal", we do not determine all DAGs in the equivalence class of the CPDAG or PDAG. Instead, we only direct edges until obtaining an amenable PDAG, which is sufficient for computing the optimal valid adjustment set. Each amenable PDAG can be obtained by orienting the neighborhood of \( X \) and then applying Meek’s orientation rules, similar to the "local" method for PDAGs. This can be done faster than the "global" method but is slower than the "local" method, especially for CPDAGs. For details see Witte, Henckel, Maathuis and Didelez (2019).

For each amenable PDAG the corresponding optimal valid adjustment set is computed. The optimal set is a valid adjustment set irrespectively of whether \( X \) is a singleton. Hence, as opposed to the other two, this method can be applied to sets \( X \). Sometimes, however, a joint total causal effect cannot be estimated via adjustment. In these cases we recommend use of the pcalg function jointIda.

We then estimate the joint total causal effect of \( X \) on \( Y \) for each valid configuration with linear regression, adjusting for the possible optimal sets. If the estimated graph is correct, each of these regressions is guaranteed to be more efficient than the corresponding linear regression with any other valid adjustment set (see Henckel, Perkovic and Maathuis (2019) for more details). The estimates are, however, more sensitive to graph estimation errors than the ones obtained with the other two methods. If \( X \) is a singleton, the output of this method is a multiset of the same size as the output of the "local" method.

For example, a CPDAG may represent eight DAGs, and the "global" method may produce an estimate of the multiset of possible total effects \{1.3, -0.5, 0.7, 1.3, 1.3, -0.5, 0.7, 0.7\}. The unique values in this set are -0.5, 0.7 and 1.3, and the multiplicities are 2, 3 and 3. The "local" and "optimal" methods, on the other hand, may produce estimates of the set \{1.3, -0.5, -0.5, 0.7\}. The unique values are again -0.5, 0.7 and 1.3, but the multiplicities are now 2, 1 and 1. The fact that the unique values of the multisets for all three methods are identical implies that summary measures of the multiset that only depend on the unique values (such as the minimum absolute value) can be estimated with all three.

Value

A list of length |\( Y \)| of matrices, each containing the possible joint total causal effect of \( X \) on one node in \( Y \).

Author(s)

Markus Kalisch (<kalisch@stat.math.ethz.ch>), Emilija Perkovic and Leonard Henckel
References


See Also

*jointIda* for estimating the multiset of possible total joint effects; *idaFast* for faster estimation of the multiset of possible total causal effects for several target variables.

*pc* for estimating a CPDAG. *addBgKnowledge* for obtaining a PDAG from CPDAG and background knowledge.

Examples

```r
## Simulate the true DAG
suppressWarnings(RNGversion("3.5.0"))
set.seed(123)
p <- 10
myDAG <- randomDAG(p, prob = 0.2) ## true DAG
myCPDAG <- dag2cpdag(myDAG) ## true CPDAG
myPDAG <- addBgKnowledge(myCPDAG,2,3) ## true PDAG with background knowledge 2 -> 3
trueCov <- trueCov(myDAG) ## true covariance matrix

## simulate Gaussian data from the true DAG
n <- 10000
dat <- rmvDAG(n, myDAG)

## estimate CPDAG and PDAG -- see help(pc)
suffStat <- list(C = cor(dat), n = n)
pc.fit <- pc(suffStat, indepTest = gaussCItest, p=p, alpha = 0.01)
pc.fit.pdag <- addBgKnowledge(pc.fit@graph,2,3)

if (require(Rgraphviz)) {
  ## plot the true and estimated graphs
  par(mfrow = c(1,3))
  }
```
Suppose that we know the true CPDAG and covariance matrix

```r
(l.ida.cpdag <- ida(3,10, covTrue, myCPDAG, method = "local", type = "cpdag"))
(o.ida.cpdag <- ida(3,10, covTrue, myCPDAG, method = "optimal", type = "cpdag"))
```

Not run: (g.ida.cpdag <- ida(3,10, covTrue, myCPDAG, method = "global", type = "cpdag"))

All three methods produce the same unique values.

Suppose that we know the true PDAG and covariance matrix

```r
(l.ida.pdag <- ida(3,10, covTrue, myPDAG, method = "local", type = "pdag"))
(o.ida.pdag <- ida(3,10, covTrue, myPDAG, method = "optimal", type = "pdag"))
```

Not run: (g.ida.pdag <- ida(3,10, covTrue, myPDAG, method = "global", type = "pdag"))

All three methods produce the same unique values.

From the true DAG, we can compute the true causal effect of 3 on 10

```r
(ce.3.10 <- causalEffect(myDAG, 10, 3))
```

Indeed, this value is contained in the values found by all methods.

When working with data we have to use the estimated CPDAG and
the sample covariance matrix

```r
(l.ida.est.cpdag <- ida(3,10, cov(dat), pc.fit@graph, method = "local", type = "cpdag"))
(o.ida.est.cpdag <- ida(3,10, cov(dat), pc.fit@graph, method = "optimal", type = "cpdag"))
```

Not run: (g.ida.est.cpdag <- ida(3,10, cov(dat), pc.fit@graph, method = "global", type = "cpdag"))

End(Not run)

The unique values of the local and the global method are still identical.
While not identical, the values of the optimal method are very similar.
The true causal effect is contained in all three sets, up to a small
estimation error (0.118 vs. 0.112 with true value 0.114)

Similarly, when working with data and background knowledge we have to use the estimated PDAG and
the sample covariance matrix

```r
(l.ida.est.pdag <- ida(3,10, cov(dat), pc.fit.pdag, method = "local", type = "pdag"))
(o.ida.est.pdag <- ida(3,10, cov(dat), pc.fit.pdag, method = "optimal", type = "pdag"))
```

Not run: (g.ida.est.pdag <- ida(3,10, cov(dat), pc.fit.pdag, method = "global", type = "pdag"))

The unique values of the local and the global method are still identical.
While not necessarily identical, the values of the optimal method will be similar.

The true causal effect is contained in both sets, up to a small estimation error

All three can also be applied to sets y.

```r
(l.ida.cpdag.2 <- ida(3,c(6,10), cov(dat), pc.fit@graph, method = "local", type = "cpdag"))
(o.ida.cpdag.2 <- ida(3,c(6,10), cov(dat), pc.fit@graph, method = "optimal", type = "cpdag"))
```

Not run: (g.ida.cpdag.2 <- ida(3,c(6,10), cov(dat), pc.fit@graph, method = "global", type = "cpdag"))

End(Not run)

For the methods local and global we recommend use of idaFast in this case for better performance.

Note that only the optimal method can be applied to sets x.

```r
(o.ida.cpdag.2 <- ida(c(2,3),10, cov(dat), pc.fit@graph, method = "optimal", type = "cpdag"))
```
**Description**

This function estimates the multiset of possible total causal effects of one variable (x) on a several (i.e., a vector of) target variables (y) from observational data.

idaFast() is more efficient than looping over ida. Only method="local" (see ida) is available.

**Usage**

idaFast(x.pos, y.pos.set, mcov, graphEst)

**Arguments**

- x.pos (integer) position of variable x in the covariance matrix.
- y.pos.set integer vector of positions of the target variables y in the covariance matrix.
- mcov covariance matrix that was used to estimate graphEst
- graphEst estimated CPDAG from the function pc. If the output of pc is pc.fit, then the estimated CPDAG can be obtained by pc.fit@graph.

**Details**

This function performs ida(x.pos,y.pos,mcov,graphEst,method="local",y.notparent=FALSE,verbose=FALSE) for all values of y.pos in y.pos.set simultaneously, in an efficient way. See (the help about) ida for more details. Note that the option y.notparent = TRUE is not implemented, since it is not clear how to do that efficiently without orienting all edges away from y.pos.set at the same time, which seems not to be desirable. Suggestions are welcome.

**Value**

Matrix with length(y.pos.set) rows. Row i contains the multiset of estimated possible total causal effects of x on y.pos.set[i]. Note that all multisets in the matrix have the same length, since the parents of x are the same for all elements of y.pos.set.

**Author(s)**

Markus Kalisch (<kalisch@stat.math.ethz.ch>)

**References**

see the list in ida.
See Also

pc for estimating a CPDAG, and ida for estimating the multiset of possible total causal effects from observational data on only one target variable but with many more options (than here in idaFast).

Examples

## Simulate the true DAG
set.seed(123)
p <- 7
myDAG <- randomDAG(p, prob = 0.2) ## true DAG
myCPDAG <- dag2cpdag(myDAG) ## true CPDAG
covTrue <- trueCov(myDAG) ## true covariance matrix

## simulate data from the true DAG
n <- 10000
dat <- rmvDAG(n, myDAG)
cov.d <- cov(dat)

## estimate CPDAG (see help on the function "pc")
suffStat <- list(C = cor(dat), n = n)
pc.fit <- pc(suffStat, indepTest = gaussCItest, alpha = 0.01, p = p)

if(require(Rgraphviz)) {
  op <- par(mfrow=c(1,3))
  plot(myDAG, main="true DAG")
  plot(myCPDAG, main="true CPDAG")
  plot(pc.fit@graph, main="pc()-estimated CPDAG")
  par(op)
}

(eff.est1 <- ida(2, 5, cov.d, pc.fit@graph))## method = "local" is default
(eff.est2 <- ida(2, 6, cov.d, pc.fit@graph))
(eff.est3 <- ida(2, 7, cov.d, pc.fit@graph))

## These three computations can be combined in an efficient way
## by using idaFast :
(eff.estF <- idaFast(2, c(5, 6, 7), cov.d, pc.fit@graph))

iplotPC  

Plotting a pcAlgo object using the package igraph

Description

Notably, when the Rgraphviz package is not easily available, iplotPC() is an alternative for plotting a "pcAlgo" object, making use of package igraph.

It extracts the adjacency matrix and converts it into an object from package igraph which is then plotted.
Usage

iplotPC(pc.fit, labels = NULL)

Arguments

pc.fit        an \( \text{R} \) object of class \text{pcAlgo}, as returned from \text{skeleton()} or \text{pc()}.
labels        optional labels for nodes; by default, the labels from the pc.fit object are used.

Value

Nothing. As side effect, the plot of pcAlgo object pc.fit.

Note

Note that this function does not work on \text{fciAlgo} objects, as those need different edge marks.

Author(s)

Markus Kalisch <kalisch@stat.math.ethz.ch>

See Also

\text{showEdgeList} for printing the edge list of a \text{pcAlgo} object; \text{showAmat} for printing the adjacency matrix of a pcAlgo object.

Examples

```r
## Load predefined data
library(gmG)
n <- nrow(gmG)
V <- colnames(gmG)
## define sufficient statistics
suffStat <- list(C = cor(gmG), n = n)
## estimate CPDAG
pc.fit <- pc(suffStat, indepTest = gaussCItest,
             alpha = 0.01, labels = V, verbose = TRUE)
## Edge list
showEdgeList(pc.fit)
## Adjacency matrix
showAmat(pc.fit)
## Plot using package igraph; show estimated CPDAG:
iplotPC(pc.fit)
```
isValidGraph

*Check for a DAG, CPDAG or a maximally oriented PDAG*

**Description**
Check whether the adjacency matrix `amat` matches the specified type.

**Usage**
```
isValidGraph(amat, type = c("pdag", "cpdag", "dag"), verbose = FALSE)
```

**Arguments**
- `amat`: adjacency matrix of type `amat.cpdag` (see `amatType`)
- `type`: string specifying the type of graph of the adjacency matrix `amat`. It can be a DAG (`type="dag"`), a CPDAG (`type="cpdag"`) or a maximally oriented PDAG (`type="pdag"`) from Meek (1995).
- `verbose`: If TRUE, detailed output on why the graph might not be valid is provided.

**Details**
For a given adjacency matrix `amat` and graph type, this function checks whether the two match.

For `type = "dag"` we require that `amat` does NOT contain directed cycles.

For `type = "cpdag"` we require that `amat` does NOT contain directed or partially directed cycles. We also require that the undirected part of the CPDAG (represented by `amat`) is made up of chordal components and that our graph is maximally oriented according to rules from Meek (1995).

For `type = "pdag"` we require that `amat` does NOT contain directed cycles. We also require that the PDAG is maximally oriented according to rules from Meek (1995). Additionally, we require that the adjacency matrix `amat1` of the CPDAG corresponding to our PDAG (represented by `amat`), satisfies `isValidGraph(amat = amat1, type = "cpdag") == TRUE` and that there is no mismatch in the orientations implied by `amat` and `amat1`. We obtain `amat1` by extracting the skeleton and v-structures from `amat` and then closing the orientation rules from Meek (1995).

**Value**
TRUE, if the adjacency matrix `amat` is of the type specified and FALSE, otherwise.

**Author(s)**
Emilija Perkovic and Markus Kalisch

**References**
Examples

```r
## a -> b -> c
amat <- matrix(c(0,1,0, 0,0,1, 0,0,0), 3,3)
colnames(amat) <- rownames(amat) <- letters[1:3]
## graph::plot(as(t(amat), "graphNEL"))
isValidGraph(amat = amat, type = "dag") ## is a valid DAG
isValidGraph(amat = amat, type = "cpdag") ## not a valid CPDAG
isValidGraph(amat = amat, type = "pdag") ## is a valid PDAG

## a -- b -- c
amat <- matrix(c(0,1,0, 1,0,1, 0,1,0), 3,3)
colnames(amat) <- rownames(amat) <- letters[1:3]
## plot(as(t(amat), "graphNEL"))
isValidGraph(amat = amat, type = "dag") ## not a valid DAG
isValidGraph(amat = amat, type = "cpdag") ## is a valid CPDAG
isValidGraph(amat = amat, type = "pdag") ## is a valid PDAG

## a -- b -- c -- d -- a
amat <- matrix(c(0,1,0, 1,0,1, 0,1,0, 1,0,1, 0,1,0), 4,4)
colnames(amat) <- rownames(amat) <- letters[1:4]
## plot(as(t(amat), "graphNEL"))
isValidGraph(amat = amat, type = "dag") ## not a valid DAG
isValidGraph(amat = amat, type = "cpdag") ## not a valid CPDAG
isValidGraph(amat = amat, type = "pdag") ## not a valid PDAG
```

---

`jointIda`  
*Estimate Multiset of Possible Total Joint Effects*

**Description**

`jointIda()` estimates the multiset of possible total joint effects of a set of intervention variables \(X\) on another variable \(Y\) from observational data. This is a version of `ida` that allows multiple simultaneous interventions.

**Usage**

```r
jointIda(x.pos, y.pos, mcov, graphEst = NULL, all.pasets = NULL, 
technique = c("RRC", "MCD"), type = c("pdag", "cpdag", "dag"))
```

**Arguments**

- `x.pos` (integer vector) positions of the intervention variables \(X\) in the covariance matrix.
- `y.pos` (integer) position of variable \(Y\) in the covariance matrix. \(y.pos\) can also be an integer vector, see Note.
- `mcov` (estimated) covariance matrix.
graphEst (graphNEL object) Estimated CPDAG or PDAG. The CPDAG is typically from \texttt{pc}(): If the result of \texttt{pc} is \texttt{pc.fit}, the estimated CPDAG can be obtained by \texttt{pc.fit@graph}. The PDAG can be obtained from CPDAG by adding background knowledge using \texttt{addBgKnowledge()}. \texttt{graphEst} can only be considered if \texttt{all.pasets} is \texttt{NULL}.

\texttt{all.pasets} (an optional argument and the default is \texttt{NULL}) A list where each element is a list of size \texttt{length(x.pos)}. Each sub-list \texttt{all.pasets[[i]]} contains possible parent sets of \texttt{x.pos} in the same order, i.e., \texttt{all.pasets[[i]][[j]]} is a possible parent set of \texttt{X.pos[j]}. This option can be used if possible parent sets of the intervention variables are known.

\texttt{technique} character string specifying the technique that will be used to estimate the total joint causal effects (given the parent sets), see details below.

"RRC": Recursive regressions for causal effects.

"MCD": Modifying the Cholesky decomposition.

\texttt{type} Type of graph "\texttt{graphEst}"; can be of type "\texttt{cpdag}", "\texttt{pdag}" (e.g. a CPDAG with background knowledge from Meek, 1995) or "\texttt{dag}".

Details

It is assumed that we have observational data that are multivariate Gaussian and faithful to the true (but unknown) underlying causal DAG (without hidden variables). Under these assumptions, this function estimates the multiset of possible total joint effects of \textit{X} on \textit{Y}. Here the total joint effect of \textit{X} = (\textit{X}_1, \textit{X}_2) on \textit{Y} is defined via Pearl’s do-calculus as the vector 
\[
E[Y|do(X_1 = x_1, X_2 = x_2)] - E[Y|do(X_1 = x_1, X_2 = x_2 + 1)] - E[Y|do(X_1 = x_1 + 1, X_2 = x_2)],
\]
with a similar definition for more than two variables. These values are equal to the partial derivatives (evaluated at \((x_1, x_2)\)) of \(E[Y|do(X = x'_1, X_2 = x'_2)]\) with respect to \(x'_1\) and \(x'_2\). Moreover, under the Gaussian assumption, these partial derivatives do not depend on the values at which they are evaluated.

We estimate a multiset of possible total joint effects instead of the unique total joint effect, since it is typically impossible to identify the latter when the true underlying causal DAG is unknown (even with an infinite amount of data).

Conceptually, the method works as follows. First, we estimate the CPDAG or PDAG based on the data. The CPDAG represents the equivalence class of DAGs and can be estimated from observational data with the function \texttt{pc} (see the help file of this function).

The PDAG contains more orientations than the CPDAG and thus, represents a smaller equivalence class of DAGs, compared to the CPDAG. We can obtain a PDAG if we have background knowledge of, for example, certain edge orientations of undirected edges in the CPDAG. We obtain the PDAG by adding these orientations to the CPDAG using the function \texttt{addBgKnowledge} (see the help file of this function).

Then using the CPDAG or PDAG we extract a collection of "jointly valid" parent sets of the intervention variables from the estimated CPDAG. For each set of jointly valid parent sets we apply RRC (recursive regressions for causal effects) or MCD (modifying the Cholesky decomposition) to estimate the total joint effect of \textit{X} on \textit{Y} from the sample covariance matrix (see Section 3 of Nandy et al, 2015).
Value

A matrix representing the multiset containing the estimated possible total joint effects of \(X\) on \(Y\). The number of rows is equal to \(\text{length}(x.\text{pos})\), i.e., each column represents a vector of possible joint causal effects.

Note

For a single variable \(X\), \texttt{jointIda()} estimates the same quantities as \texttt{ida()}. If \texttt{graphEst} is of type = "cpdag", \texttt{jointIda()} obtains all.pasets by using the semi-local approach described in Section 5 in Nandy et. al, (2015). Nandy et. al, (2015) show that \texttt{jointIda()} yields correct multiplicities of the distinct elements of the resulting multiset (in the sense that it matches \texttt{ida()} with method="global" up to a constant factor).

If \texttt{graphEst} is of type = "pdag", \texttt{jointIda()} obtains all.pasets by using the semi-local approach described in Algorithm 2, Section 4.2 in Perkovic et. al (2017). For this case, \texttt{jointIda()} does not necessarily yield the correct multiplicities of the distinct elements of the resulting multiset (it behaves similarly to \texttt{ida()} with method="local").

\texttt{jointIda()} (like \texttt{idaFast}) also allows direct computation of the total joint effect of a set of intervention variables \(X\) on another set of target variables \(Y\). In this case, \(y.\text{pos}\) must be an integer vector containing positions of the target variables \(Y\) in the covariance matrix and the output is a list of matrices that correspond to the variables in \(Y\) in the same order. This method is slightly more efficient than looping over \texttt{jointIda()} with single target variables, if \texttt{all.pasets} is not specified.

Author(s)

Preetam Nandy, Emilija Perkovic

References


See Also

\texttt{ida}, the simple version; \texttt{pc} for estimating a CPDAG.

Examples

```r
## Create a weighted DAG
p <- 6
V <- as.character(1:p)
edL <- list(
  "1" = list(edges=c(3,4), weights=c(1.1,0.3)),
  "2" = list(edges=c(6), weights=c(0.4)),
  "3" = list(edges=c(2,4,6), weights=c(0.6,0.8,0.9)),
  "4" = list(edges=c(2), weights=c(0.5)),
  "5" = list(edges=c(1,4), weights=c(0.2,0.7)),
  "6" = NULL)
```
myDAG <- new("graphNEL", nodes=V, edgeL=edL, edgemode="directed") ## true DAG
myCPDAG <- dag2cpdag(myDAG) ## true CPDAG
myPDAG <- addBgKnowledge(myCPDAG,1,3) ## true PDAG with background knowledge 1 -> 3
covTrue <- trueCov(myDAG) ## true covariance matrix
n <- 1000
## simulate Gaussian data from the true DAG
dat <- if (require("mvtnorm")) {
  set.seed(123)
  rmvnorm(n, mean=rep(0,p), sigma=covTrue)
} else readRDS(system.file(package="pcalg", "external", "N_6_1000.rds"))
## estimate CPDAG and PDAG -- see help(pc), help(addBgKnoweldge)
suffStat <- list(C = cor(dat), n = n)
pc.fit <- pc(suffStat, indepTest = gaussCItest, p = p, alpha = 0.01, u2pd="relaxed")
pc.fit.pdag <- addBgKnowledge(pc.fit@graph,1,3)
if (require("Rgraphviz")) {
  ## plot the true and estimated graphs
  par(mfrow = c(1,3))
  plot(myDAG, main = "True DAG")
  plot(pc.fit, main = "Estimated CPDAG")
  plot(pc.fit.pdag, main = "Estimated PDAG")
}
## Suppose that we know the true CPDAG and covariance matrix
jointIda(x.pos=c(1,2), y.pos=6, covTrue, graphEst=myCPDAG, technique="RRC", type = "cpdag")
jointIda(x.pos=c(1,2), y.pos=6, covTrue, graphEst=myCPDAG, technique="MCD", type = "cpdag")
## Suppose that we know the true PDAG and covariance matrix
jointIda(x.pos=c(1,2), y.pos=6, covTrue, graphEst=myPDAG, technique="RRC", type = "pdag")
jointIda(x.pos=c(1,2), y.pos=6, covTrue, graphEst=myPDAG, technique="MCD", type = "pdag")
## Instead of knowing the true CPDAG or PDAG, it is enough to know only
## the jointly valid parent sets of the intervention variables
## to use RRC or MCD
## all.jointly.valid.pasets:
ajv.pasets <- list(list(5,c(3,4)),list(integer(0),c(3,4)),list(3,c(3,4)))
jointIda(x.pos=c(1,2), y.pos=6, covTrue, all.pasets=ajv.pasets, technique="RRC")
jointIda(x.pos=c(1,2), y.pos=6, covTrue, all.pasets=ajv.pasets, technique="MCD")
## From the true DAG, we can compute the true total joint effects
## using RRC or MCD
cat("Dim covTrue: ", dim(covTrue), "\n")
jointIda(x.pos=c(1,2), y.pos=6, covTrue, graphEst=myDAG, technique="RRC", type = "dag")
jointIda(x.pos=c(1,2), y.pos=6, covTrue, graphEst=myDAG, technique="MCD", type = "dag")
## When working with data, we have to use the estimated CPDAG or PDAG
## and the sample covariance matrix
jointIda(x.pos=c(1,2), y.pos=6, cov(dat), graphEst=pc.fit@graph, technique="RRC", type = "cpdag")
jointIda(x.pos=c(1,2), y.pos=6, cov(dat), graphEst=pc.fit@graph, technique="MCD", type = "cpdag")
**lega.path**

```
jointIda(x.pos=c(1,2), y.pos=6, cov(dat), graphEst=pc.fit.pdag, technique="RRC", type = "pdag")
jointIda(x.pos=c(1,2), y.pos=6, cov(dat), graphEst=pc.fit.pdag, technique="MCD", type = "pdag")

## RRC and MCD can produce different results when working with data
## jointIda also works when x.pos has length 1 and in the following example
## it gives the same result as ida() (see Note)
##
## When the CPDAG is known
jointIda(x.pos=1, y.pos=6, covTrue, graphEst=myCPDAG, technique="RRC", type = "cpdag")
ida(x.pos=1, y.pos=6, covTrue, graphEst=myCPDAG, method="global", type = "cpdag")

## When the PDAG is known
jointIda(x.pos=1, y.pos=6, covTrue, graphEst=myPDAG, technique="RRC", type = "pdag")
ida(x.pos=1, y.pos=6, covTrue, graphEst=myPDAG, method="global", type = "pdag")

## When the DAG is known
jointIda(x.pos=1, y.pos=6, covTrue, graphEst=myDAG, technique="RRC", type = "dag")
ida(x.pos=1, y.pos=6, covTrue, graphEst=myDAG, method="global")

## Note that, causalEffect(myDAG,y=6,x=1) does not give the correct value in this case,
## since this function requires that the variables are in a causal order.
```

---

**legal.path**

**Check if a 3-node-path is Legal**

**Description**

Check if the path $a \rightarrow b \rightarrow c$ is legal.

A 3-node path $a \rightarrow b \rightarrow c$ is “legal” iff either $b$ is a collider or $a \rightarrow b \rightarrow c$ is a triangle.

**Usage**

`legal.path(a, b, c, amat)`

**Arguments**

- **a, b, c** (integer) positions in adjacency matrix of nodes $a$, $b$, and $c$, respectively.
- **amat** Adjacency matrix (coding 0,1,2,3 for no edge, circle, arrowhead, tail; e.g., amat[a,b] = 2 and amat[b,a] = 3 implies $a \rightarrow b$)

**Value**

TRUE if path is legal, otherwise FALSE.

**Note**

Prerequisite: $a \rightarrow b \rightarrow c$ must be in a path (and this is not checked by `legal.path()`).
Author(s)
Markus Kalisch (<kalisch@stat.math.ethz.ch>)

Examples
amat <- matrix(c(0,1,0,0, 2,0,1,0,0, 2,2,0,2,1, 0,0,1,0,0, 0,0,2,0,0), 5,5)
legal.path(1,3,5, amat)
legal.path(1,2,3, amat)
legal.path(2,3,4, amat)

Description
Fits a Linear non-Gaussian Acyclic Model (LiNGAM) to the data and returns the corresponding DAG.
For details, see the reference below.

Usage
lingam(X, verbose = FALSE)

## For back-compatibility; this is *deprecated*
LINGAM(X, verbose = FALSE)

Arguments
X
n x p data matrix (n: sample size, p: number of variables).
verbose
logical or integer indicating that increased diagnostic output is to be provided.

Value
lingam() returns an \texttt{R} object of (S3) class "LINGAM", basically a \texttt{list} with components
\begin{itemize}
\item \texttt{Bpruned} a \( p \times p \) matrix \( B \) of linear coefficients, where \( B_{i,j} \) corresponds to a directed edge from \( j \) to \( i \).
\item \texttt{stde} a vector of length \( p \) with the standard deviations of the estimated residuals
\item \texttt{ci} a vector of length \( p \) with the intercepts of each equation
\end{itemize}

LINGAM() — *deprecated now* — returns a \texttt{list} with components
\begin{itemize}
\item \texttt{Adj} a \( p \times p \) 0/1 adjacency matrix \( A \). \( A[i,j] = 1 \) corresponds to a directed edge from \( i \) to \( j \).
\item \texttt{B} \( p \times p \) matrix of corresponding linear coefficients. Note it corresponds to the transpose of \texttt{Adj}, i.e., identical( \texttt{Adj}, t(\texttt{B}) != 0 ) is true.
\end{itemize}
Author(s)

Of LINGAM() and the underlying functionality.

Patrik Hoyer <patrik.hoyer@helsinki.fi>, Doris Entner <entnerd@hotmail.com>, Antti Hyttinen <antti.hyttinen@cs.helsinki.fi> and Jonas Peters <jonas.peters@tuebingen.mpg.de>.

References


See Also

fastICA from package fastICA is used.

Examples

```r
# Exp 1
set.seed(1234)
n <- 500
eps1 <- sign(rnorm(n)) * sqrt(abs(rnorm(n)))
eps2 <- runif(n) - 0.5
x2 <- 3 + eps2
x1 <- 0.9 * x2 + 7 + eps1

#truth: x1 <- x2
trueDAG <- cbind(c(0,1), c(0,0))
X <- cbind(x1, x2)
res <- lingam(X)

cat("true DAG:
")
show(trueDAG)
cat("estimated DAG:
")
show(res$amat)

cat("true constants:
")
show(c(7, 3))
cat("estimated constants:
")
show(res$ci)

cat("true (sample) noise standard deviations:
")
show(c(sd(eps1), sd(eps2)))
cat("estimated noise standard deviations:
")
show(res$stde)
```

# Exp 2

```r
set.seed(1234)
X <- matrix(rnorm(900), nrow = 100)
res <- lingam(X)

cat("estimated DAG:
")
show(res$amat)

cat("estimated constants:
")
show(res$ci)

cat("estimated noise standard deviations:
")
show(res$stde)
```
## Exp 2

```r
set.seed(123)
n <- 500
es1 <- sign(rnorm(n)) * sqrt(abs(rnorm(n)))
es2 <- runif(n) - 0.5
es3 <- sign(rnorm(n)) * abs(rnorm(n))^(1/3)
es4 <- rnorm(n)^2

x2 <- eps2
x1 <- 0.9*x2 + eps1
x3 <- 0.8*x2 + eps3
x4 <- -x1 -0.9*x3 + eps4

X <- cbind(x1,x2,x3,x4)

trueDAG <- cbind(x1 = c(0,1,0,0),
                 x2 = c(0,0,0,0),
                 x3 = c(0,1,0,0),
                 x4 = c(1,0,1,0))

# adjacency matrix:
# 0 0 0 1
# 1 0 1 0
# 0 0 0 1
# 0 0 0 0
```

```r
res1 <- lingam(X, verbose = TRUE)# details on LINGAM
res2 <- lingam(X, verbose = 2) # details on LINGAM and fastICA

## results are the same, of course:
stopifnot(identical(res1, res2))
cat("true DAG:
")
show(trueDAG)

cat("estimated DAG:
")
as(res1, "amat")
```

### mat2targets

**Conversion between an intervention matrix and a list of intervention targets**

**Description**

In a data set with \( n \) measurements of \( p \) variables, intervened variables can be specified in two ways:

- with a **logical** intervention matrix of dimension \( n \times p \), where the entry \([i,j]\) indicates whether variable \( j \) has been intervened in measurement \( i \); or
- with a list of (unique) intervention targets and a \( p \)-dimensional vector indicating the indices of the intervention targets of the \( p \) measurements.
The function `mat2targets` converts the first representation to the second one, the function `targets2mat` does the reverse conversion. The second representation can be used to create scoring objects (see `Score`) and to run causal inference methods based on interventional data such as `gies` or `simy`.

### Usage

```r
mat2targets(A)
targets2mat(p, targets, target.index)
```

### Arguments

- **A** Logical matrix with \( n \) rows and \( p \) columns, where \( n \) is the sample size of a data set with jointly interventional and observational data, and \( p \) is the number of variables. \( A[i,j] \) is TRUE iff variable \( j \) is intervened in data point \( i \).
- **p** Number of variables
- **targets** List of unique intervention targets
- **target.index** Vector of intervention target indices. The intervention target of data point \( i \) is encoded as `targets[target.index[i]]`.

### Value

- `mat2targets` returns a list with two components:
  - **targets** A list of unique intervention targets.
  - **target.index** A vector of intervention target indices. The intervention target of data point \( i \) is encoded as `targets[target.index[i]]`.

### Author(s)

Alain Hauser (<alain.hauser@bfh.ch>)

### See Also

`Score`, `gies`, `simy`

### Examples

```r
## Specify interventions using a matrix
p <- 5
n <- 10
A <- matrix(FALSE, nrow = n, ncol = p)
for (i in 1:n) A[i, (i-1) %% p + 1] <- TRUE

## Generate list of intervention targets and corresponding indices
target.list <- mat2targets(A)
for (i in 1:length(target.list$target.index))
  sprintf("Intervention target of %d-th data point: %d",
           i, target.list$targets[[target.list$target.index[i]]])
```
## Convert back to matrix representation

```r
all(A == targets2mat(p, target.list$targets, target.list$target.index))
```

---

### mcor

*Compute (Large) Correlation Matrix*

#### Description

Compute a correlation matrix, possibly by robust methods, applicable also for the case of a large number of variables.

#### Usage

```r
mcor(dm, method = c("standard", "Qn", "QnStable", "ogkScaleTau2", "ogkQn", "shrink"))
```

#### Arguments

- **dm**: numeric data matrix; rows are observations ("samples"), columns are variables.
- **method**: a string; "standard" (default), "Qn", "QnStable", "ogkQn" and "shrink" envoke standard, elementwise robust (based on $Q_n$ scale estimator, see $Q_n$), robust ($Q_n$ using OGK, see $covOGK$) or shinked correlation estimate respectively.

#### Details

The "standard" method envoke a standard correlation estimator. "Qn" envoke a robust, elementwise correlation estimator based on the $Q_n$ scale estimator. "QnStable" also uses the $Q_n$ scale estimator, but uses an improved way of transforming that into the correlation estimator. "ogkQn" envoke a correlation estimator based on $Q_n$ using OGK. "shrink" is only useful when used with pcSelect. An optimal shrinkage parameter is used. Only correlation between response and covariates is shinked.

#### Value

A correlation matrix estimated by the specified method.

#### Author(s)

Markus Kalisch <kalisch@stat.math.ethz.ch> and Martin Maechler

#### References

See those in the help pages for $Q_n$ and $covOGK$ from package *robustbase*.

#### See Also

$Q_n$ and $covOGK$ from package *robustbase*, $pcorOrder$ for computing partial correlations.
Examples

```r
## produce uncorrelated normal random variables
set.seed(42)
x <- rnorm(100)
y <- 2*x + rnorm(100)
## compute correlation of var1 and var2
mcor(cbind(x, y), method="standard")

## repeat but this time with heavy-tailed noise
yNoise <- 2*x + rcauchy(100)
mcor(cbind(x, yNoise), method="standard")  ## shows almost no correlation
mcor(cbind(x, yNoise), method="Qn")        ## shows a lot correlation
mcor(cbind(x, yNoise), method="QnStable") ## shows still much correlation
mcor(cbind(x, yNoise), method="ogkQn")   ## ditto
```

opt.target

Get an optimal intervention target

Description

Given a (observational or interventional) essential graph (or "CPDAG"), find the optimal intervention target that maximizes the number of edges that can be oriented after the intervention.

Usage

```r
opt.target(essgraph, max.size, use.node.names = TRUE)
```

Arguments

- `essgraph`: An `EssGraph` or `graphNEL` object representing a (observational or interventional) essential graph (or "CPDAG").
- `max.size`: Maximum size of the intervention target. Only 1 and the number of vertices of `essgraph` are allowed; the latter means no size limit is applied (the default if the parameter is missing).
- `use.node.names`: Indicates if the intervention target should be returned as a list of node names (if `TRUE`) or indices (if `FALSE`).

Details

This function implements active learning strategies for structure learning from interventional data, one that calculates an optimal single-vertex intervention target, and one that calculates an optimal intervention target of arbitrary size. "Optimal" means the proposed intervention target guarantees the highest number of edges that can be oriented after performing the intervention, assuming the essential graph provided as input is the true essential graph under the currently available interventional data (i.e., neglecting possible estimation errors).

optAdjSet

Compute the optimal adjustment set

Description

optAdjSet computes the optimal valid adjustment set relative to the variables (X,Y) in the given graph.
Usage

\texttt{optAdjSet(graphEst,x.pos,y.pos)}

Arguments

- \texttt{graphEst}: graphNel object or adjacency matrix of type amat.cpdag.
- \texttt{x.pos, x}: Positions of variables \(X\) in the covariance matrix.
- \texttt{y.pos, y}: Positions of variables \(Y\) in the covariance matrix.

Details

Suppose we have data from a linear SEM compatible with a known causal graph \(G\) and our aim is to estimate the total joint effect of \(X\) on \(Y\). Here the total joint effect of \(X = (X_1, X_2)\) on \(Y\) is defined via Pearl’s do-calculus as the vector \(\langle E[Y|do(X_1 = x_1 + 1, X_2 = x_2)] - E[Y|do(X_1 = x_1, X_2 = x_2)], E[Y|do(X_1 = x_1, X_2 = x_2 + 1)] - E[Y|do(X_1 = x_1, X_2 = x_2)]\rangle\), with a similar definition for more than two variables. These values are equal to the partial derivatives (evaluated at \(x_1, x_2\)) of \(E[Y|do(X = x'_1, X_2 = x'_2)]\) with respect to \(x_1'\) and \(x_2'\). Moreover, under the linearity assumption, these partial derivatives do not depend on the values at which they are evaluated.

It is possible to estimate the total joint effect of \(X\) on \(Y\) with a simple linear regression of the form \(\text{lm}(Y \sim X + Z)\), if and only if the covariate set \(Z\) is a valid adjustment set (see Perkovic et al. (2018)). Often, however, there are multiple such valid adjustment sets, providing total effect estimates with varying accuracies. Suppose that there exists a valid adjustment set relative to \((X, Y)\) in causal graph \(G\), and each node in \(Y\) is a descendant of \(X\), then there exists a valid adjustment which provides the total effect estimate with the optimal asymptotic variance, which we will refer to as \(O(X, Y, G)\) (Henckel et al., 2019). This function returns this optimal valid adjustment set \(O(X, Y, G)\).

The restriction that each node in \(Y\) be a descendant of the node set \(X\) is not notable, as the total effect of the node set \(X\) on a non-descendant is always 0. If provided with a node set \(Y\) that does not fulfill this condition this function computes a pruned node set \(Y_2\) by removing all nodes from \(Y\) that are not descendants of \(X\) and returns \(O(X, Y_2, G)\) instead. The user will be alerted to this and given the pruned set \(Y_2\).

Value

A vector with the positions of the nodes of the optimal set \(O(X, Y, G)\).

Author(s)

Leonard Henckel

References


Examples

```r
suppressWarnings(RNGversion("3.5.0"))
set.seed(123)
p <- 10
## true DAG
myDAG <- randomDAG(p, prob = 0.3)
## true CPDAG
myCPDAG <- dag2cpdag(myDAG)
## true PDAG with added background knowledge 5 -> 6
myPDAG <- addBgKnowledge(myCPDAG, 5, 6)
par(mfrow = c(1,3))
plot(myDAG)
plot(myPDAG)
plot(myCPDAG) ## plot of the graphs
## if the CPDAG C is amenable relative to (X, Y),
## the optimal set will be the same for all DAGs
## and any max. PDAGs obtained by adding background knowledge to C
(optAdjSet(myDAG, 3, 10))
(optAdjSet(myPDAG, 3, 10))
(optAdjSet(myCPDAG, 3, 10))

## the optimal adjustment set can also be compute for sets X and Y
(optAdjSet(myDAG, c(3,4), c(9,10)))
(optAdjSet(myPDAG, c(3,4), c(9,10)))
(optAdjSet(myCPDAG, c(3,4), c(9,10)))

## The only restriction is that it requires all nodes in Y to be
## descendants of X.
## However, if a node in Y is non-descendant of X the lowest variance
## partial total effect estimate is simply 0.
## Hence, we can proceed with a pruned Y. This function does this automatically!
(optAdjSet(myDAG, 1, c(3,9)))

## Note that for sets X there may be no valid adjustment set even
## if the PDAG is is amenable relative to (X, Y).
## Not run: optAdjSet(myPDAG, c(4,9),7)
```

pag2mag  

Transform a PAG into a MAG in the Corresponding Markov Equivalence Class

Description

Transform a Partial Ancestral Graph (PAG) into a valid Maximal Ancestral Graph (MAG) that belongs to the Markov equivalence class represented by the given PAG, with no additional edges into node x.
pag2mag

Usage

pag2magAM(amat.pag, x, max.chordal = 10, verbose = FALSE)

Arguments

amat.pag  Adjacency matrix of type amat.pag
x         (integer) position in adjacency matrix of node in the PAG into which no additional edges are oriented.
max.chordal Positive integer: graph paths larger than max.chordal are considered to be too large to be checked for chordality.
verbose   Logical; if true, some output is produced during computation.

Details

This function converts a PAG (adjacency matrix) to a valid MAG (adjacency matrix) that belongs to the Markov equivalence class represented by the given PAG. Note that we assume that there are no selection variables, meaning that the edges in the PAG can be of the following types: ->, <->, o->, and o-o. In a first step, it uses the Arrowhead Augmentation of Zhang (2006), i.e., any o-> edge is oriented into ->. Afterwards, it orients each chordal component into a valid DAG without orienting any additional edges into x.

This function is used in the Generalized Backdoor Criterion backdoor with type="pag", see Maathuis and Colombo (2015) for details.

Value

The output is an adjacency matrix of type amat.pag representing a valid MAG that belongs to the Markov equivalence class represented by the given PAG.

Author(s)

Diego Colombo, Markus Kalisch and Martin Maechler.

References


See Also

fci, dag2pag, backdoor
Examples

```r
## create the graph
set.seed(78)
p <- 12
g <- randomDAG(p, prob = 0.4)
## Compute the true covariance and then correlation matrix of g:
true.corr <- cov2cor(trueCov(g))

## define nodes 2 and 6 to be latent variables
L <- c(2,6)

## Find PAG
## As dependence "oracle", we use the true correlation matrix in
gaussCItest() with a large "virtual sample size" and a large alpha:
true.pag <- dag2pag(suffStat = list(C= true.corr, n= 10^9),
                   indepTest = gaussCItest, graph=g, L=L, alpha= 0.9999)

## find a valid MAG such that no additional edges are directed into
 amat.mag <- pag2magAM(true.pag@amat, 4)  # -> the adj.matrix of the MAG
```

ParDAG-class

This virtual base class represents a parametric causal model.

Description

This virtual base class represents a parametric causal model.

Details

The class "ParDAG" serves as a basis for simulating observational and/or interventional data from causal models as well as for parameter estimation (maximum-likelihood estimation) for a given causal model in the presence of a data set with jointly observational and interventional data.

The virtual base class "ParDAG" provides a “skeleton” for all functions relied to the aforementioned task. In practical cases, a user may always choose an appropriate class derived from ParDAG which represents a specific parametric model class. The base class itself does not represent such a model class.

Constructor

```r
new("ParDAG", nodes, in.edges, params)
```

- **nodes**: Vector of node names; cf. also field .nodes.
- **in.edges**: A list of length p consisting of index vectors indicating the edges pointing into the nodes of the DAG.
- **params**: A list of length p consisting of parameter vectors modeling the conditional distribution of a node given its parents; cf. also field .params.
Fields

.nodes: Vector of node names; defaults to `as.character(1:p)`, where `p` denotes the number of nodes (variables) of the model.

.in.edges: A list of length `p` consisting of index vectors indicating the edges pointing into the nodes of the DAG.

.params: A list of length `p` consisting of parameter vectors modeling the conditional distribution of a node given its parents. The entries of the parameter vectors only get a concrete meaning in derived classes belonging to specific parametric model classes.

Class-Based Methods

node.count(): Yields the number of nodes (variables) of the model.

simulate(n, target, int.level): Generates `n` (observational or interventional) samples from the parametric causal model. The intervention target to be used is specified by the parameter `target`; if the target is empty (`target = integer(0)`), observational samples are generated. `int.level` indicates the values of the intervened variables; if it is a vector of the same length as `target`, all samples are drawn from the same intervention levels; if it is a matrix with `n` rows and as many columns as `target` has entries, its rows are interpreted as individual intervention levels for each sample.

edge.count(): Yields the number of edges (arrows) in the DAG.

mle.fit(score): Fits the parameters using an appropriate `Score` object.

Methods

plot signature(x = "ParDAG", y = "ANY"): plots the underlying DAG of the causal model. Parameters are not visualized.

Author(s)

Alain Hauser (<alain.hauser@bfh.ch>)

See Also

GaussParDAG

Description

Estimate the equivalence class of a directed acyclic graph (DAG) from observational data, using the PC-algorithm.
Usage

pc(suffStat, indepTest, alpha, labels, p,
fixedGaps = NULL, fixedEdges = NULL, NAdelete = TRUE, m.max = Inf,
u2pd = c("relaxed", "rand", "retry"),
skel.method = c("stable", "original", "stable.fast"),
conservative = FALSE, maj.rule = FALSE, solve.confl = FALSE,
numCores = 1, verbose = FALSE)

Arguments

suffStat  A list of sufficient statistics, containing all necessary elements for the conditional independence decisions in the function indepTest.

indepTest  A function for testing conditional independence. It is internally called as indepTest(x, y, S, suffStat), and tests conditional independence of x and y given S. Here, x and y are variables, and S is a (possibly empty) vector of variables (all variables are denoted by their (integer) column positions in the adjacency matrix). suffStat is a list, see the argument above. The return value of indepTest is the p-value of the test for conditional independence.

alpha  significance level (number in (0, 1) for the individual conditional independence tests.

labels  (optional) character vector of variable (or “node”) names. Typically preferred to specifying p.

p  (optional) number of variables (or nodes). May be specified if labels are not, in which case labels is set to 1:p.

numCores  Specifies the number of cores to be used for parallel estimation of skeleton.

verbose  If TRUE, detailed output is provided.

fixedGaps  A logical matrix of dimension p*p. If entry [i, j] or [j, i] (or both) are TRUE, the edge i-j is removed before starting the algorithm. Therefore, this edge is guaranteed to be absent in the resulting graph.

fixedEdges  A logical matrix of dimension p*p. If entry [i, j] or [j, i] (or both) are TRUE, the edge i-j is never considered for removal. Therefore, this edge is guaranteed to be present in the resulting graph.

NAdelete  If indepTest returns NA and this option is TRUE, the corresponding edge is deleted. If this option is FALSE, the edge is not deleted.

m.max  Maximal size of the conditioning sets that are considered in the conditional independence tests.

u2pd  String specifying the method for dealing with conflicting information when trying to orient edges (see details below).

skel.method  Character string specifying method; the default, "stable" provides an order-independent skeleton, see skeleton.

conservative  Logical indicating if the conservative PC is used. In this case, only option u2pd = "relaxed" is supported. Note that therefore the resulting object might not be extendable to a DAG. See details for more information.
Logical indicating that the triples shall be checked for ambiguity using a majority rule idea, which is less strict than the conservative PC algorithm. For more information, see details.

If TRUE, the orientation of the v-structures and the orientation rules work with lists for candidate sets and allow bi-directed edges to resolve conflicting edge orientations. In this case, only option u2pd = relaxed is supported. Note, that therefore the resulting object might not be a CPDAG because bi-directed edges might be present. See details for more information.

Details

Under the assumption that the distribution of the observed variables is faithful to a DAG, this function estimates the Markov equivalence class of the DAG. We do not estimate the DAG itself, because this is typically impossible (even with an infinite amount of data), since different DAGs can describe the same conditional independence relationships. Since all DAGs in an equivalence class describe the same conditional independence relationships, they are equally valid ways to describe the conditional dependence structure that was given as input.

All DAGs in a Markov equivalence class have the same skeleton (i.e., the same adjacency information) and the same v-structures (see definition below). However, the direction of some edges may be undetermined, in the sense that they point one way in one DAG in the equivalence class, while they point the other way in another DAG in the equivalence class.

A Markov equivalence class can be uniquely represented by a completed partially directed acyclic graph (CPDAG). A CPDAG contains undirected and directed edges. The edges have the following interpretation: (i) there is a (directed or undirected) edge between i and j if and only if variables i and j are conditionally dependent given S for all possible subsets S of the remaining nodes; (ii) a directed edge $i \rightarrow j$ means that this directed edge is present in all DAGs in the Markov equivalence class; (iii) an undirected edge $i \leftrightarrow j$ means that there is at least one DAG in the Markov equivalence class with edge $i \rightarrow j$ and there is at least one DAG in the Markov equivalence class with edge $i \leftarrow j$.

The CPDAG is estimated using the PC algorithm (named after its inventors Peter Spirtes and Clark Glymour). The skeleton is estimated by the function skeleton which uses a modified version of the original PC algorithm (see Colombo and Maathuis (2014) for details). The original PC algorithm is known to be order-dependent, in the sense that the output depends on the order in which the variables are given. Therefore, Colombo and Maathuis (2014) proposed a simple modification, called PC-stable, that yields order-independent adjacencies in the skeleton (see the help file of this function for details). Subsequently, as many edges as possible are oriented. This is done in two steps. It is important to note that if no further actions are taken (see below) these two steps still remain order-dependent.

The edges are oriented as follows. First, the algorithm considers all triples $(a, b, c)$, where $a$ and $b$ are adjacent, $b$ and $c$ are adjacent, but $a$ and $c$ are not adjacent. For all such triples, we direct both edges towards $b$ ($a \rightarrow b \leftarrow c$) if and only if $b$ was not part of the conditioning set that made the edge between $a$ and $c$ drop out. These conditioning sets were saved in sepset. The structure $a \rightarrow b \leftarrow c$ is called a v-structure.

After determining all v-structures, there may still be undirected edges. It may be possible to direct some of these edges, since one can deduce that one of the two possible directions of the edge is invalid because it introduces a new v-structure or a directed cycle. Such edges are found by repeat-
edly applying rules R1-R3 of the PC algorithm as given in Algorithm 2 of Kalisch and Bühlmann (2007). The algorithm stops if none of the rules is applicable to the graph.

The conservative PC algorithm (conservative = TRUE) is a slight variation of the PC algorithm (see Ramsey et al. 2006). After the skeleton is computed, all potential v-structures \( a \rightarrow b \rightarrow c \) are checked in the following way. We test whether \( a \) and \( c \) are independent conditioning on all subsets of the neighbors of \( a \) and all subsets of the neighbors of \( c \). When a subset makes \( a \) and \( c \) conditionally independent, we call it a separating set. If \( b \) is in no such separating set or in all such separating sets, no further action is taken and the usual PC is continued. If, however, \( b \) is in only some separating sets, the triple \( a \rightarrow b \rightarrow c \) is marked as 'ambiguous'. Moreover, if no separating set is found among the neighbors, the triple is also marked as 'ambiguous'. An ambiguous triple is not oriented as a v-structure. Furthermore, no further orientation rule that needs to know whether \( a \rightarrow b \rightarrow c \) is a v-structure or not is applied. Instead of using the conservative version, which is quite strict towards the v-structures, Colombo and Maathuis (2014) introduced a less strict version for the v-structures called majority rule. This adaptation can be called using maj.rule = TRUE. In this case, the triple \( a \rightarrow b \rightarrow c \) is marked as 'ambiguous' if and only if \( b \) is in exactly 50 percent of such separating sets or no separating set was found. If \( b \) is in less than 50 percent of the separating sets it is set as a v-structure, and if in more than 50 percent it is set as a non v-structure (for more details see Colombo and Maathuis, 2014). The usage of both the conservative and the majority rule versions resolve the order-dependence issues of the determination of the v-structures.

Sampling errors (or hidden variables) can lead to conflicting information about edge directions. For example, one may find that \( a \rightarrow b \rightarrow c \) and \( b \rightarrow c \rightarrow d \) should both be directed as v-structures. This gives conflicting information about the edge \( b \rightarrow c \), since it should be directed as \( b \leftarrow c \) in v-structure \( a \rightarrow b \leftarrow c \), while it should be directed as \( b \rightarrow c \) in v-structure \( b \rightarrow c \rightarrow d \). With the option solve.confl = FALSE, in such cases, we simply overwrite the directions of the conflicting edge. In the example above this means that we obtain \( a \rightarrow b \rightarrow c \leftarrow d \) if \( a \rightarrow b \rightarrow c \) was visited first, and \( a \rightarrow b \leftarrow c \leftarrow d \) if \( b \rightarrow c \rightarrow d \) was visited first, meaning that the final orientation on the edge depends on the ordering in which the v-structures were considered. With the option solve.confl = TRUE (which is only supported with option u2pd = "relaxed"), we first generate a list of all (unambiguous) v-structures (in the example above \( a \rightarrow b \rightarrow c \) and \( b \rightarrow c \rightarrow d \)), and then we simply orient them allowing both directions on the edge \( b \rightarrow c \), namely we allow the bi-directed edge \( b \leftrightarrow c \) resolving the order-dependence issues on the edge orientations. We denote bi-directed edges in the adjacency matrix \( M \) of the graph as \( M[b,c] = 2 \) and \( M[c,b] = 2 \). In a similar way, using lists for the candidate edges for each orientation rule and allowing bi-directed edges, the order-dependence issues in the orientation rules can be resolved. Note that bi-directed edges merely represent a conflicting orientation and they should not to be interpreted causally. The usage of these lists for the candidate edges and allowing bi-directed edges resolves the order-dependence issues on the orientation of the v-structures and on the orientation rules, see Colombo and Maathuis (2014) for more details.

Note that calling (conservative = TRUE), or maj.rule = TRUE, together with solve.confl = TRUE produces a fully order-independent output, see Colombo and Maathuis (2014).

Sampling errors, non faithfulness, or hidden variables can also lead to non-extendable CPDAGs, meaning that there does not exist a DAG that has the same skeleton and v-structures as the graph found by the algorithm. An example of this is an undirected cycle consisting of the edges \( a \rightarrow b \rightarrow c \rightarrow d \) and \( d \rightarrow a \). In this case it is impossible to direct the edges without creating a cycle or a new v-structure. The option u2pd specifies what should be done in such a situation. If the option is set to "relaxed", the algorithm simply outputs the invalid CPDAG. If the option is set to "rand", all direction information is discarded and a random DAG is generated on the skeleton, which is
then converted into its CPDAG. If the option is set to "retry", up to 100 combinations of possible
directions of the ambiguous edges are tried, and the first combination that results in an extendable
CPDAG is chosen. If no valid combination is found, an arbitrary DAG is generated on the skeleton
as in the option "rand", and then converted into its CPDAG. Note that the output can also be an
invalid CPDAG, in the sense that it cannot arise from the oracle PC algorithm, but be extendible to
a DAG, for example $a \rightarrow b \leftarrow c \leftarrow d$. In this case, u2pd is not used.

Using the function `isValidGraph` one can check if the final output is indeed a valid CPDAG.

Notes: (1) Throughout, the algorithm works with the column positions of the variables in the adja-
cency matrix, and not with the names of the variables. (2) When plotting the object, undirected and
bidirected edges are equivalent.

**Value**

An object of class "pcAlgo" (see `pcAlgo`) containing an estimate of the equivalence class of the
underlying DAG.

**Author(s)**

Markus Kalisch (kalisch@stat.math.ethz.ch), Martin Maechler, and Diego Colombo.

**References**

D. Colombo and M.H. Maathuis (2014). Order-independent constraint-based causal structure learn-

Using Graphical Models with the R Package pcalg. *Journal of Statistical Software* **47(11)** 1–26,
https://www.jstatsoft.org/v47/i11/.

M. Kalisch and P. Buehlmann (2007). Estimating high-dimensional directed acyclic graphs with
the PC-algorithm. *JMLR* **8** 613-636.

J. Ramsey, J. Zhang and P. Spirtes (2006). Adjacency-faithfulness and conservative causal infer-
ence. In *Proceedings of the 22nd Annual Conference on Uncertainty in Artificial Intelligence*. AUAI
Press, Arlington, VA.

The MIT Press.

**See Also**

`skeleton` for estimating a skeleton of a DAG; `udag2pdag` for converting the skeleton to a CPDAG;
`gaussCItest`, `disCItest`, `binCItest` and `dsepTest` as examples for `indepTest`. `isValidGraph`
for testing whether the output is a valid CPDAG.

**Examples**

```r
# Using Gaussian Data
# Load predefined data
data(gmG)
```
n <- nrow(gmG8$x)
V <- colnames(gmG8$x)  # labels aka node names

## estimate CPDAG
pc.fit <- pc(suffStat = list(C = cor(gmG8$x), n = n),
             indepTest = gaussCItest,  # indep.test: partial correlations
             alpha = 0.01, labels = V, verbose = TRUE)

if (require(Rgraphviz)) {
  ## show estimated CPDAG
  par(mfrow=c(1,2))
  plot(pc.fit, main = "Estimated CPDAG")
  plot(gmG8$g, main = "True DAG")
}

#############################################################################
## Using d-separation oracle
#############################################################################
## define sufficient statistics (d-separation oracle)
suffStat <- list(g = gmG8$g, jp = RBGL::johnson.all.pairs.sp(gmG8$g))
## estimate CPDAG
fit <- pc(suffStat, indepTest = dsepTest, labels = V,
          alpha = 0.01)  # value is irrelevant as dsepTest returns either 0 or 1

if (require(Rgraphviz)) {
  ## show estimated CPDAG
  plot(fit, main = "Estimated CPDAG")
  plot(gmG8$g, main = "True DAG")
}

#############################################################################
## Using discrete data
#############################################################################
## Load data
data(gmD)
V <- colnames(gmD$x)
## define sufficient statistics
suffStat <- list(dm = gmD$x, nlev = c(3,2,3,4,2), adaptDF = FALSE)
## estimate CPDAG
pc.D <- pc(suffStat,
           indepTest = disCItest, alpha = 0.01, labels = V, verbose = TRUE)

if (require(Rgraphviz)) {
  ## show estimated CPDAG
  par(mfrow = c(1,2))
  plot(pc.D, main = "Estimated CPDAG")
  plot(gmD$g, main = "True DAG")
}

#############################################################################
## Using binary data
#############################################################################
## Load binary data
data(gmB)
V <- colnames(gmB$x)
## estimate CPDAG
pc.B <- pc(suffStat = list(dm = gmB$x, adaptDF = FALSE),
indepTest = binCItest, alpha = 0.01, labels = V, verbose = TRUE)

if (require(Rgraphviz)) {
  ## show estimated CPDAG
  plot(pc.B, main = "Estimated CPDAG")
  plot(gmB$g, main = "True DAG")
}

#########################################################################
## Detecting ambiguities due to sampling error
#########################################################################
## Load predefined data
data(gmG)
n <- nrow(gmG$x)
V <- colnames(gmG$x) # labels aka node names

## estimate CPDAG
pc.fit <- pc(suffStat = list(C = cor(gmG$x), n = n),
indepTest = gaussCItest, # indep.test: partial correlations
  alpha=0.01, labels = V, verbose = TRUE)

## due to sampling error, some edges were overwritten:
isValidGraph(as(pc.fit, "amat"), type = "cpdag")

## re-fit with solve.conf1 = TRUE
pc.fit2 <- pc(suffStat = list(C = cor(gmG$x), n = n),
indepTest = gaussCItest, # indep.test: partial correlations
  alpha=0.01, labels = V, verbose = TRUE,
solve.conf1 = TRUE)

## conflicting edge is V5 - V6
as(pc.fit2, "amat")

---

**pc.cons.intern**

Utility for conservative and majority rule in PC and FCI

**Description**

The `pc.cons.intern()` function is used in `pc` and `fci`, notably when `conservative = TRUE` (conservative orientation of v-structures) or `maj.rule = TRUE` (majority rule orientation of v-structures).

**Usage**

```r
pc.cons.intern(sk, suffStat, indepTest, alpha, version.unf = c(NA, NA),
              maj.rule = FALSE, verbose = FALSE)
```
Arguments

- **sk**: A skeleton object as returned from `skeleton()`.

- **suffStat**: Sufficient statistic: List containing all necessary elements for the conditional independence decisions in the function `indepTest`.

- **indepTest**: Pre-defined function for testing conditional independence. The function is internally called as `indepTest(x,y,S,suffStat)`, and tests conditional independence of x and y given S. Here, x and y are variables, and S is a (possibly empty) vector of variables (all variables are denoted by their column numbers in the adjacency matrix). `suffStat` is a list containing all relevant elements for the conditional independence decisions. The return value of `indepTest` is the p-value of the test for conditional independence.

- **alpha**: Significance level for the individual conditional independence tests.

- **version.unf**: Vector of length two. If `version.unf[2]==1`, the initial separating set found by the PC/FCI algorithm is added to the set of separating sets; if `version.unf[2]==2`, it is not added. In the latter case, if the set of separating sets is empty, the triple is marked as unambiguous if `version.unf[1]==1`, and as ambiguous if `version.unf[1]==2`.

- **maj.rule**: Logical indicating if the triples are checked for ambiguity using the majority rule idea, which is less strict than the standard conservative method.

- **verbose**: Logical asking for detailed output.

Details

For any unshielded triple A-B-C, consider all subsets of the neighbors of A and of the neighbors of C, and record all such sets D for which A and C are conditionally independent given D. We call such sets “separating sets”.

If `version.unf[2]==1`, the initial separating set found by the PC/FCI algorithm is added to this set of separating sets. If `version.unf[2]==2`, the initial separating set is not added (as in Tetrad).

In the latter case, if the set of separating sets is empty, then the triple is marked as ‘ambiguous’ if `version.unf[1]==2`, for example in `pc`, or as ‘unambiguous’ if `version.unf[1]==1`, for example in `fc`. Otherwise, there is at least one separating set. If `maj.rule=FALSE`, the conservative PC algorithm is used (Ramsey et al., 2006): If B is in some but not all separating sets, the triple is marked as ambiguous. Otherwise it is treated as in the standard PC algorithm. If `maj.rule=TRUE`, the majority rule is applied (Colombo and Maathuis, 2014): The triple is marked as ‘ambiguous’ if B is in exactly 50 percent of the separating sets. If it is in less than 50 percent it is marked as a v-structure, and if it is in more than 50 percent it is marked as a non v-structure.

Note: This function modifies the separating sets for unambiguous triples in the skeleton object (adding or removing B) to ensure that the usual orientations rules later on lead to the correct v-structures/non v-structures.

Value

- **unfTripl**: numeric vector of triples coded as numbers (via `triple2numb()`) that were marked as ambiguous.
Vector containing the version (1 or 2) of the corresponding triple saved in unfTripl (1=normal ambiguous triple, i.e., B is in some sepsets but not all or none; 2=triple coming from version.unf[1]==2, i.e., a and c are indep given the initial sepset but there does not exist a subset of the neighbours of a or of c that d-separates them.)

The updated skeleton-object (separating sets might have been updated).

Author(s)

Markus Kalisch (<kalisch@stat.math.ethz.ch>) and Diego Colombo.

References


See Also

`skeleton`, `pc`, `fci`

Transform the adjacency matrix from *pcalg* into a *dagitty* object

**Description**

Transform the adjacency matrix of type amat.cpdag or amat.pag (for details on coding see `amatType`).

**Usage**

`pcalg2dagitty(amat, labels, type = "cpdag")`

**Arguments**

*amat*  adjacency matrix of type amat.cpdag or amat.pag

*labels*  character vector of variable (or “node”) names.

*type*  string specifying the type of graph of the adjacency matrix amat. It can be a DAG (type="dag"), a CPDAG (type="cpdag") or a maximally oriented PDAG (type="pdag") from Meek (1995); then the type of adjacency matrix is assumed to be amat.cpdag. It can also be a MAG (type = "mag") or a PAG (type="pag"); then the type of the adjacency matrix is assumed to be amat.pag.
Details

For a given adjacency matrix amat the form amat.cpdag or amat.pag and a specified graph type, this function returns a dagitty object corresponding to the graph structure specified by amat, labels and type. The resulting object is compatible with the dagitty package.

Value

A dagitty graph (see the dagitty package).

Author(s)

Emilija Perkovic and Markus Kalisch

Examples

data(gmG)
n <- nrow(gmG)
V <- colnames(gmG) # labels aka node names

amat <- wgtMatrix(gmG)
 amat[amat != 0] <- 1
if(requireNamespace("dagitty")) {
  dagitty_dag1 <- pcalg2dagitty(amat,V,type="dag")
}

----------------------------------------

pcAlgo  PC-Algorithm [OLD]: Estimate Skeleton or Equivalence Class of a DAG

Description

This function is DEPRECATED! Use skeleton, pc or fci instead.

Use the PC-algorithm to estimate the underlying graph ("skeleton") or the equivalence class (CPDAG) of a DAG.

Usage

pcAlgo(dm = NA, C = NA, n=NA, alpha, corMethod = "standard",
       verbose=FALSE, directed=FALSE, G=NULL, datatype = "continuous",
       NAdelete=TRUE, m.max=1Inf, u2pd = "rand", psepset=FALSE)
Arguments

- **dm**: Data matrix; rows correspond to samples, cols correspond to nodes.
- **C**: Correlation matrix; this is an alternative for specifying the data matrix.
- **n**: Sample size; this is only needed if the data matrix is not provided.
- **alpha**: Significance level for the individual partial correlation tests.
- **corMethod**: A character string specifying the method for (partial) correlation estimation. "standard", "QnStable", "Qn" or "ogkQn" for standard and robust (based on the Qn scale estimator without and with OGK) correlation estimation. For robust estimation, we recommend "QnStable".
- **verbose**: 0-no output, 1-small output, 2-details; using 1 and 2 makes the function very much slower
- **directed**: If FALSE, the underlying skeleton is computed; if TRUE, the underlying CPDAG is computed
- **G**: The adjacency matrix of the graph from which the algorithm should start (logical)
- **datatype**: Distinguish between discrete and continuous data
- **NAdelete**: Delete edge if pval=NA (for discrete data)
- **m.max**: Maximal size of conditioning set
- **u2pd**: Function used for converting skeleton to cpdag. "rand" (use udag2pdag); "relaxed" (use udag2pdagRelaxed); "retry" (use udag2pdagSpecial)
- **psepset**: If true, also possible separation sets are tested.

Value

An object of class "pcAlgo" (see **pcAlgo**) containing an undirected graph (object of class "graph", see **graph-class** from the package **graph**) (without weights) as estimate of the skeleton or the CPDAG of the underlying DAG.

Author(s)

Markus Kalisch (<kalisch@stat.math.ethz.ch>) and Martin Maechler.

References


Description

This class of objects is returned by the functions `skeleton` and `pc` to represent the (skeleton) of an estimated CPDAG. Objects of this class have methods for the functions plot, show and summary.

Usage

```r
## S4 method for signature 'pcAlgo,ANY'
plot(x, y, main = NULL,
     zvalue.lwd = FALSE, lwd.max = 7, labels = NULL, ...)
## S3 method for class 'pcAlgo'
print(x, amat = FALSE, zero.print = ".", ...)
## S4 method for signature 'pcAlgo'
summary(object, amat = TRUE, zero.print = ".", ...)
## S4 method for signature 'pcAlgo'
show(object)
```

Arguments

- `x`, `object`: a "pcAlgo" object.
- `y`: (generic `plot()` argument; unused).
- `main`: main title for the plot (with an automatic default).
- `zvalue.lwd`: logical indicating if the line width (`lwd`) of the edges should be made proportional to the entries of matrix zMin (originally) or derived from matrix pMax.
- `lwd.max`: maximal `lwd` to be used, if `zvalue.lwd` is true.
- `labels`: if non-NULL, these are used to define node attributes `nodeAttrs` and `attrs`, passed to `agopen()` from package `Rgraphviz`.
- `amat`: logical indicating if the adjacency matrix should be shown (printed) as well.
- `zero.print`: string for printing 0 (‘zero’) entries in the adjacency matrix.
- `...`: optional further arguments (passed from and to methods).

Creation of objects

Objects are typically created as result from `skeleton()` or `pc()`, but could be be created by calls of the form `new("pcAlgo",...).`
Slots

The slots call, n, max.ord, n.edgetests, sepset, and pMax are inherited from class "gAlgo", see there.

In addition, "pcAlgo" has slots

`graph`: Object of class "graph": the undirected or partially directed graph that was estimated.

`zMin`: Deprecated.

Extends

Class "gAlgo".

Methods

- plot signature(x = "pcAlgo"): Plot the resulting graph. If argument "zvalue.lwd" is true, the linewidth an edge reflects zMin, so that thicker lines indicate more reliable dependencies. The argument "lwd.max" controls the maximum linewidth.
- show signature(object = "pcAlgo"): Show basic properties of the fitted object
- summary signature(object = "pcAlgo"): Show details of the fitted object

Author(s)

Markus Kalisch and Martin Maechler

See Also

pc, skeleton, fciAlgo

Examples

```r
showClass("pcAlgo")

## generate a pcAlgo object
p <- 8
set.seed(45)
myDAG <- randomDAG(p, prob = 0.3)
n <- 10000
d.mat <- rmvDAG(n, myDAG, errDist = "normal")
suffStat <- list(C = cor(d.mat), n = n)
pc.fit <- pc(suffStat, indepTest = gaussCItest, alpha = 0.01, p = p)

## use methods of class pcAlgo
show(pc.fit)
if(require(Rgraphviz))
  plot(pc.fit, main = "Fitted graph")
summary(pc.fit)

## access slots of this object
(g <- pc.fit@graph)
str(ss <- pc.fit@sepset, max=1)
```
pcorOrder  

Compute Partial Correlations

Description
This function computes partial correlations given a correlation matrix using a recursive algorithm.

Usage
pcorOrder(i, j, k, C, cut.at = 0.9999999)

Arguments
- **i, j**: (integer) position of variable \(i\) and \(j\), respectively, in correlation matrix.
- **k**: (integer) positions of zero or more conditioning variables in the correlation matrix.
- **C**: Correlation matrix (matrix)
- **cut.at**: Number slightly smaller than one; if \(c\) is cut.at, values outside of \([-c, c]\) are set to \(-c\) or \(c\) respectively.

Details
The partial correlations are computed using a recursive formula if the size of the conditioning set is one. For larger conditioning sets, the pseudoinverse of parts of the correlation matrix is computed (by `pseudoinverse()` from package `corpcor`). The pseudoinverse instead of the inverse is used in order to avoid numerical problems.

Value
The partial correlation of \(i\) and \(j\) given the set \(k\).

Author(s)
Markus Kalisch <kalisch@stat.math.ethz.ch> and Martin Maechler

See Also
- `condIndFisherZ` for testing zero partial correlation.

Examples
```r
## produce uncorrelated normal random variables
mat <- matrix(rnorm(3*20), 20, 3)
## compute partial correlation of var1 and var2 given var3
pcorOrder(1, 2, 3, cor(mat))

## define graphical model, simulate data and compute
## partial correlation with bigger conditional set
```
pcSelect <- randomDAG(20, prob = 0.2)
dat <- rmvDAG(1000, genDAG)
C <- cor(dat)
pcorOrder(2,5, k = c(3,7,8,14,19), C)

---

**pcSelect**  
*PC-Select: Estimate subgraph around a response variable*

**Description**

The goal is feature selection: If you have a response variable \( y \) and a data matrix \( dm \), we want to know which variables are “strongly influential” on \( y \). The type of influence is the same as in the PC-Algorithm, i.e., \( y \) and \( x \) (a column of \( dm \)) are associated if they are correlated even when conditioning on any subset of the remaining columns in \( dm \). Therefore, only very strong relations will be found and the result is typically a subset of other feature selection techniques. Note that there are also robust correlation methods available which render this method robust.

**Usage**

```r
pcSelect(y, dm, alpha, corMethod = "standard", verbose = FALSE, directed = FALSE)
```

**Arguments**

- \( y \)  
  response vector.
- \( dm \)  
  data matrix (rows: samples/observations, columns: variables); \( nrow(dm) = length(y) \).
- \( alpha \)  
  significance level of individual partial correlation tests.
- \( corMethod \)  
  a string determining the method for correlation estimation via \( \text{mcor()} \); specifically any of the \( \text{mcor(\ldots)} \) can be used, e.g., "Qn" for one kind of robust correlation estimate.
- \( verbose \)  
  logical or in \{0, 1, 2\}:
    - **FALSE, 0**: No output,
    - **TRUE, 1**: Little output,
    - **2**: Detailed output.
    
    Note that such diagnostic output may make the function considerably slower.
- \( directed \)  
  logical; should the output graph be directed?

**Details**

This function basically applies \texttt{pc} on the data matrix obtained by joining \( y \) and \( dm \). Since the output is not concerned with the edges found within the columns of \( dm \), the algorithm is adapted accordingly. Therefore, the runtime and the ability to deal with large datasets is typically increased substantially.
Value

G  A logical vector indicating which column of dm is associated with y.

zMin  The minimal z-values when testing partial correlations between y and each column of dm. The larger the number, the more consistent is the edge with the data.

Author(s)

Markus Kalisch (<kalisch@stat.math.ethz.ch>) and Martin Maechler.

References


See Also

pc which is the more general version of this function; pcSelect.presel which applies pcSelect() twice.

Examples

```r
p <- 10
## generate and draw random DAG:
suppressWarnings(RNGversion("3.5.0"))
set.seed(101)
myDAG <- randomDAG(p, prob = 0.2)
if (require(Rgraphviz)) {
  plot(myDAG, main = "randomDAG(10, prob = 0.2)"
}
## generate 1000 samples of DAG using standard normal error distribution
n <- 1000
d.mat <- rmvDAG(n, myDAG, errDist = "normal")

## let's pretend that the 10th column is the response and the first 9 columns are explanatory variable. Which of the first 9 variables "cause" the tenth variable?
y <- d.mat[,10]
dm <- d.mat[,-10]
(pcS <- pcSelect(d.mat[,10], d.mat[,-10], alpha=0.05))
## You see, that variable 4,5,6 are considered as important
## By inspecting zMin,
with(pcS, zMin[G])
## you can also see that the influence of variable 6
## is most evident from the data (its zMin is 18.64, so quite large - as
## a rule of thumb for judging what is large, you could use quantiles
## of the Standard Normal Distribution)
```
**pcSelect.presel**

*Estimate Subgraph around a Response Variable using Preselection*

**Description**

This function uses `pcSelect` to preselect some covariates and then runs `pcSelect` again on the reduced data set.

**Usage**

```r
cpSelect.presel(y, dm, alpha, alphapre, corMethod = "standard", verbose = 0, directed=FALSE)
```

**Arguments**

- `y`: Response vector.
- `dm`: Data matrix (rows: samples, cols: nodes; i.e., `length(y) == nrow(dm)`).
- `alpha`: Significance level of individual partial correlation tests.
- `alphapre`: Significance level for `pcSelect` in preselection.
- `corMethod`: "standard" or "Qn" for standard or robust correlation estimation.
- `verbose`: 0-no output, 1-small output, 2-details (using 1 and 2 makes the function very much slower).
- `directed`: Logical; should the output graph be directed?

**Details**

First, `pcSelect` is run using `alphapre`. Then, only the important variables are kept and `pcSelect` is run on them again.

**Value**

- `pcs`: Logical vector indicating which column of `dm` is associated with `y`.
- `zMin`: The minimal z-values when testing partial correlations between `y` and each column of `dm`. The larger the number, the more consistent is the edge with the data.
- `Xnew`: Preselected Variables.

**Author(s)**

Philipp Ruetimann

**See Also**

`pcSelect`
Examples

```r
p <- 10
## generate and draw random DAG :
set.seed(101)
myDAG <- randomDAG(p, prob = 0.2)
if(require(Rgraphviz))
    plot(myDAG, main = "randomDAG(10, prob = 0.2)"

## generate 1000 samples of DAG using standard normal error distribution
n <- 1000
d.mat <- rmvDAG(n, myDAG, errDist = "normal")

## let's pretend that the 10th column is the response and the first 9 columns are explanatory variable. Which of the first 9 variables "cause" the tenth variable?
y <- d.mat[,10]
dm <- d.mat[,,-10]
res <- pcSelect.presel(d.mat[,10], d.mat[,,-10], alpha=0.05, alphapre=0.6)
```

---

### pdag2allDags

**Enumerate All DAGs in a Markov Equivalence Class**

**Description**

pdag2allDags computes all DAGs in the Markov Equivalence Class Represented by a Given Partially Directed Acyclic Graph (PDAG).

**Usage**

```r
pdag2allDags(gm, verbose = FALSE)
```

**Arguments**

- `gm` adjacency matrix of type `amat.cpdag`
- `verbose` logical; if true, some output is produced during computation

**Details**

All DAGs extending the given PDAG are computing while avoiding new v-structures and cycles. If no DAG is found, the function returns NULL.

**Value**

List with two elements:

- `dags`: Matrix; every row corresponds to a DAG; every column corresponds to an entry in the adjacency matrix of this DAG. Thus, the adjacency matrix (of type `amat.cpdag`) contained in the i-th row of matrix dags can be obtained by calling matrix(dags[i,],p,p,byrow = TRUE) (assuming the input PDAG has p nodes).
nodeNms Node labels of the input PDAG.

Author(s)
Markus Kalisch (<kalisch@stat.math.ethz.ch>)

Examples

```r
## Example 1
gm <- rbind(c(0,1),
             c(1,0))
colnames(gm) <- rownames(gm) <- LETTERS[1:2]
res1 <- pdag2allDags(gm)
## adjacency matrix of first DAG in output
amat1 <- matrix(res1$dags[1,],2,2, byrow = TRUE)
colnames(amat1) <- rownames(amat1) <- res1$nodeNms
amat1 ## A --> B

## Example 2
gm <- rbind(c(0,1,1),
             c(1,0,1),
             c(1,1,0))
colnames(gm) <- rownames(gm) <- LETTERS[1:ncol(gm)]
res2 <- pdag2allDags(gm)
## adjacency matrix of first DAG in output
amat2 <- matrix(res2$dags[1,],3,3, byrow = TRUE)
colnames(amat2) <- rownames(amat2) <- res2$nodeNms
amat2

## Example 3
gm <- rbind(c(0,1,1,0,0),
             c(1,0,0,0,0),
             c(1,0,0,0,0),
             c(0,1,1,0,1),
             c(0,0,0,1,0))
colnames(gm) <- rownames(gm) <- LETTERS[1:ncol(gm)]
res3 <- pdag2allDags(gm)
## adjacency matrix of first DAG in output
amat3 <- matrix(res3$dags[1,],5,5, byrow = TRUE)
colnames(amat3) <- rownames(amat3) <- res3$nodeNms
amat3

## for convenience a simple plotting function
## for the function output
plotAllDags <- function(res) {
  require(graph)
  p <- sqrt(ncol(res$dags))
  nDags <- ceiling(sqrt(nrow(res$dags)))
  par(mfrow = c(nDags, nDags))
  for (i in 1:nrow(res$dags)) {
    tmp <- matrix(res$dags[i,],p,p)
    colnames(tmp) <- rownames(tmp) <- res$nodeNms
    plot(as(tmp, "graphNEL"))
  }
}
```
pdag2dag

Extend a Partially Directed Acyclic Graph (PDAG) to a DAG

Description
This function extends a PDAG (Partially Directed Acyclic Graph) to a DAG, if this is possible.

Usage
pdag2dag(g, keepVstruct=TRUE)

Arguments
g
Input PDAG (graph object)
keepVstruct
Logical indicating if the v-structures in g are kept. Otherwise they are ignored and an arbitrary extension is generated.

Details
Direct undirected edges without creating directed cycles or additional v-structures. The PDAG is consistently extended to a DAG using the algorithm by Dor and Tarsi (1992). If no extension is possible, a DAG corresponding to the skeleton of the PDAG is generated and a warning message is produced.

Value
List with entries

graph
Contains a consistent DAG extension (graph object).
success
Is TRUE iff the extension was possible.

Author(s)
Markus Kalisch <kalisch@stat.math.ethz.ch>

References
Examples

```r
p <- 10 # number of random variables
n <- 10000 # number of samples
s <- 0.4 # sparsness of the graph

## generate random data
set.seed(42)
g <- randomDAG(p, prob = s) # generate a random DAG
d <- rmvDAG(n, g) # generate random samples

gSkel <- pcAlgo(d, alpha = 0.05) # estimate of the skeleton
(gPDAG <- udag2pdag(gSkel))
(gDAG <- pdag2dag(gPDAG@graph))
```

Description

Estimate the final skeleton in the FCI algorithm (Spirtes et al, 2000), as described in Steps 2 and 3 of Algorithm 3.1 in Colombo et al. (2012). The input of this function consists of an initial skeleton that was estimated by the PC algorithm (Step 1 of Algorithm 3.1 in Colombo et al. (2012)). Given the initial skeleton, all unshielded triples are considered and oriented as colliders when appropriate. Then, for all nodes x in the resulting partially directed graph G, Possible-D-SEP(x,G) is computed, using the function qreach. Finally, for any edge y-z that is present in G, conditional independence between Y and Z is tested given all subsets of Possible-D-SEP(y,G) and all subsets of Possible-D-SEP(z,G). These tests are done at level alpha, using indepTest. If the pair of nodes is judged to be independent given some set S, then S is recorded in sepset(y,z) and sepset(z,y) and the edge y-z is deleted. Otherwise, the edge remains and there is no change to sepset.

Usage

```r
pdsep(skel, suffStat, indepTest, p, sepset, alpha, pMax, m.max = Inf, pdsep.max = Inf, NAdelete = TRUE, unfVect = NULL, biCC = FALSE, verbose = FALSE)
```

Arguments

- `skel`: Graph object returned by `skeleton`.
- `suffStat`: Sufficient statistic: A list containing all necessary elements for making conditional independence decisions using function `indepTest`.
- `indepTest`: Predefined function for testing conditional independence. The function is internally called as `indepTest(x, y, S, suffStat)` for testing conditional independence of x and y given S. Here, x and y are node numbers of the adjacency matrix, S is a (possibly empty) vector of node numbers of the adjacency matrix.
and suffStat is a list containing all relevant elements for making conditional independence decisions. The return value of indepTest is the p-value of the test for conditional independence.

- **p**
  Number of variables.

- **sepset**
  List of length p; each element of the list contains another list of length p. The element sepset[[x]][[y]] contains the separation set that made the edge between x and y drop out. This object is thought to be obtained from a pcAlgo-object or fciAlgo-object.

- **alpha**
  Significance level for the individual conditional independence tests.

- **pMax**
  Matrix with the maximal p-values of conditional independence tests in a previous call of skeleton, pc or fci which produced G. This object is thought to be obtained from a pcAlgo-object or fciAlgo-object.

- **m.max**
  Maximum size of the conditioning sets that are considered in the conditional independence tests.

- **pdsep.max**
  Maximum size of Possible-D-SEP for which subsets are considered as conditioning sets in the conditional independence tests. If the nodes x and y are adjacent in the graph and the size of Possible-D-SEP(x,G) \ x.y, is bigger than pdsep.max, the edge is simply left in the graph. Note that if pdsep.max is less than Inf, the final PAG is typically a supergraph of the one computed with pdsep.max = Inf, because fewer tests may have been performed in the former.

- **NAdelete**
  If indepTest returns NA and this option is TRUE, the corresponding edge is deleted. If this option is FALSE, the edge is not deleted.

- **unfVect**
  Vector containing numbers that encode the unfaithful triple (as returned by pc.cons.intern). This is needed in the conservative FCI.

- **biCC**
  Logical; if TRUE, only nodes on paths between nodes a and c are considered to be in sepset(a,c). This uses biconnected components, see biConnComp from RBGL.

- **verbose**
  Logical indicating that detailed output is to be provided.

### Details

To make the code more efficient, we only perform tests that were not performed in the estimation of the initial skeleton.

Note that the Possible-D-SEP sets are computed once in the beginning. They are not updated after edge deletions, in order to make sure that the output of the algorithm does not depend on the ordering of the variables (see also Colombo and Maathuis (2014)).

### Value

A list with the following elements:

- **G**
  Updated adjacency matrix representing the final skeleton

- **sepset**
  Updated sepsets

- **pMax**
  Updated matrix containing maximal p-values

- **allPdsep**
  Possible-D-Sep for each node
max. ord  Maximal order of conditioning sets during independence tests
n. edgetests  Number of conditional edgetests performed, grouped by the size of the conditioning set.

Author(s)
Markus Kalisch (<kalisch@stat.math.ethz.ch>) and Diego Colombo.

References

See Also
qreach to find Possible-D-SEP(x,G); fci.

Examples

```r
p <- 10
## generate and draw random DAG:
set.seed(44)
myDAG <- randomDAG(p, prob = 0.2)

## generate 10000 samples of DAG using gaussian distribution
library(RBGL)
n <- 10000
d.mat <- rmvDAG(n, myDAG, errDist = "normal")

## estimate skeleton
indepTest <- gaussCItest
suffStat <- list(C = cor(d.mat), n = n)
alpha <- 0.01
skel <- skeleton(suffStat, indepTest, alpha=alpha, p=p)

## prepare input for pdsep
sepset <- skel@sepset
pMax <- skel@pMax

## call pdsep to find Possible-D-Sep and enhance the skeleton
pdsepRes <- pdsep(skel@graph, suffStat, indepTest, p, sepset, alpha, pMax, verbose = TRUE)

## call pdsep with biconnected components to find Possible-D-Sep and enhance the skeleton
pdsepResBicc <- pdsep(skel@graph, suffStat, indepTest, p, sepset, alpha, pMax, biCC= TRUE, verbose = TRUE)
```
**plotAG**  
*Plot partial ancestral graphs (PAG)*

**Description**

This function is DEPRECATED! Use the `plot` method of the `fciAlgo` class instead.

**Usage**

```r
plotAG(amat)
```

**Arguments**

- `amat`  
  Adjacency matrix (coding 0,1,2,3 for no edge, circle, arrowhead, tail; e.g., `amat[a,b] = 2` and `amat[b,a] = 3` implies a -> b)

**Author(s)**

Markus Kalisch (<kalisch@stat.math.ethz.ch>)

**See Also**

- `fci`

---

**plotSG**  
*Plot the subgraph around a Specific Node in a Graph Object*

**Description**

Plots a subgraph for a specified starting node and a given graph. The subgraph consists of those nodes that can be reached from the starting node by passing no more than a specified number of edges.

**Usage**

```r
plotSG(graphObj, y, dist, amat = NA, directed = TRUE, 
plot = requireNamespace("Rgraphviz"), main = , 
cex.main = 1.25, font.main = par("font.main"), col.main=par("col.main"), 
...)
```
Arguments

graphObj  An \( \mathcal{R} \) object of class \texttt{graph}.

\( y \)  (integer) position of the starting node in the adjacency matrix.

\( \text{dist} \)  Distance of nodes included in subgraph from starting node \( y \).

\( \text{amat} \)  Precomputed adjacency matrix of type \texttt{amat.cpdag} (optional)

\( \text{directed} \)  \texttt{logical} indicating if the subgraph should be directed.

\( \text{plot} \)  \texttt{logical} indicating if the subgraph should be plotted (or just returned). Defaults to true when \texttt{Rgraphviz} is installed.

\( \text{main} \)  title to be used, with a sensible default; see \texttt{title}.

\( \text{cex.main}, \text{font.main}, \text{col.main} \)  optional settings for the \texttt{main} title; see \texttt{title}.

\( \ldots \)  optional arguments passed to the \texttt{plot()} from package \texttt{Rgraphviz}.

Details

Commencing at the starting point \( y \) the function looks for the neighbouring nodes. Beginning with direct parents and children it will continue hierarchically through the distances to \( y \). Note that the neighbourhood does \textit{not} depend on edge directions. If \texttt{directed} is true (as per default), the orientation of the edges is taken from the initial graph.

For the plotting, the package \texttt{Rgraphviz} must be installed.

Value

the desired subgraph is returned; invisibly, i.e., via \texttt{invisible}, if \texttt{plot} is true.

Author(s)

Daniel Stekhoven, then Martin Maechler.

Examples

```R
## generate a random DAG:
p <- 10
set.seed(45)
myDAG <- randomDAG(p, prob = 0.3)

if(requireNamespace("Rgraphviz")) {
  ## plot whole the DAG
  plot(myDAG, main = "randomDAG(10, prob = 0.3)"

  op <- par(mfrow = c(3,2))
  ## plot the neighbours of node number 8 up to distance 1
  plotSG(myDAG, 8, 1, directed = TRUE)
  plotSG(myDAG, 8, 1, directed = FALSE)

  ## plot the neighbours of node number 8 up to distance 2
  plotSG(myDAG, 8, 2, directed = TRUE)
  plotSG(myDAG, 8, 2, directed = FALSE)
} 
```
## plot the neighbours of node number 8 up to distance 3
plotSG(myDAG, 8, 3, directed = TRUE)
plotSG(myDAG, 8, 3, directed = FALSE)

## Note that the layout of the subgraph might be different than in the
## original graph, but the graph structure is identical
par(op)
} else {  
## without 'Rgraphviz'
sg2d <- plotSG(myDAG, 8, 2, directed = TRUE, plot=FALSE)
sg2u <- plotSG(myDAG, 8, 2, directed = FALSE, plot=FALSE)
}

---

### possAn

**Find possible ancestors of given node(s).**

**Description**

In a DAG, CPDAG, MAG or PAG determine which nodes are (possible) ancestors of x on definite status or just any paths potentially avoiding given nodes on the paths.

**Usage**

```r
possAn(m, x, y = NULL, possible = TRUE, ds = TRUE,
       type = c("cpdag", "pdag", "dag", "mag", "pag"))
```

**Arguments**

- `m`: Adjacency matrix in coding according to type.
- `x`: Node positions of starting nodes.
- `y`: Node positions of nodes through which a path must not go.
- `possible`: If TRUE, possible ancestors are returned.
- `ds`: If TRUE, only paths of definite status are considered.
- `type`: Type of adjacency matrix in m. The coding is according to amatType.

**Details**

Not all possible combinations of the arguments are currently implemented and will issue an error if called.

**Value**

Vector of all node positions found as (possible) ancestors of the nodes in x.

**Author(s)**

Markus Kalisch
See Also
amatType

Examples

```r
## a -- b -> c
amat <- matrix(c(0,1,0, 1,0,1, 0,0,0), 3,3)
colnames(amat) <- rownames(amat) <- letters[1:3]
plot(as(t(amat), "graphNEL"))

possAn(m = amat, x = 3, possible = TRUE, ds = FALSE, type = "pdag") ## all nodes
possAn(m = amat, x = 3, y = 2, possible = TRUE, ds = FALSE, type = "pdag") ## only node 1
```

possDe  

Find possible descendants of given node(s).

Description

In a DAG, CPDAG, MAG or PAG determine which nodes are (possible) descendants of x on definite status or just any paths potentially avoiding given nodes on the paths.

Usage

```r
possDe(m, x, y = NULL, possible = TRUE, ds = TRUE,
type = c("cpdag", "pdag", "dag", "mag", "pag"))
```

Arguments

- `m`  
  Adjacency matrix in coding according to type.
- `x`  
  Node positions of starting nodes.
- `y`  
  Node positions of nodes through which a path must not go.
- `possible`  
  If TRUE, possible descendents are returned.
- `ds`  
  If TRUE, only paths of definite status are considered.
- `type`  
  Type of adjacency matrix in m. The coding is according to amatType.

Details

Not all possible combinations of the arguments are currently implemented and will issue an error if called.

Value

Vector of all node positions found as (possible) descendents of the nodes in x.

Author(s)

Markus Kalisch
### possibleDe

[DEPRECATED] Find possible descendants on definite status paths.

#### Description

This function is DEPRECATED! Use `possDe` instead.

In a DAG, CPDAG, MAG or PAG determine which nodes are possible descendants of x on definite status paths.

#### Usage

```r
possibleDe(amat, x)
```

#### Arguments

- **amat**: adjacency matrix of type `amat.pag`
- **x**: (integer) position of node x (node of interest) in the adjacency matrix.

#### Details

A non-endpoint vertex X on a path p in a partial mixed graph is said to be of a *definite status* if it is either a collider or a definite non-collider on p. The path p is said to be of a *definite status* if all non-endpoint vertices on the path are of a definite status (see e.g. Maathuis and Colombo (2015), Def. 3.4).

A possible descendant of x can be reached moving to adjacent nodes of x but never going against an arrowhead.

#### Value

Vector with possible descendents.

#### Author(s)

Diego Colombo
qreach

References

See Also
backdoor, amatType

Examples
amat <- matrix( c(0,3,0,0,0, 0,2,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,1,0, 0,0,0,0,1,0, 0,0,0,0,0,0, 0,0,0,0,1,0, 0,0,0,0,0,0, 0,0,0,0,0,0), nrow=6, ncol=6)
colnames(amat) <- rownames(amat) <- letters[1:6]
if(require(Rgraphviz)) {
  plotAG(amat)
}
possibleDe(amat, 1) ## a, b are poss. desc. of a
possibleDe(amat, 4) ## d, e, f are poss. desc. of d

qreach(x, amat, verbose = FALSE)

Arguments

x (integer) position of vertex x in the adjacency matrix of which Possible-D-SEP set is to be computed.

amat Adjacency matrix of type amat.pag.

verbose Logical, asking for details on output

Value
Vector of column positions indicating the nodes in Possible-D-SEP of x.
r.gauss.pardag

Generate a Gaussian Causal Model Randomly

Description
Generate a random Gaussian causal model. Parameters specifying the connectivity as well as coefficients and error terms of the corresponding linear structural equation model can be specified. The observational expectation value of the generated model is always 0, meaning that no interception terms are drawn.

Usage
r.gauss.pardag(p, prob, top.sort = FALSE, normalize = FALSE, lbe = 0.1, ube = 1, neg.coef = TRUE, labels = as.character(1:p), lbv = 0.5, ubv = 1)

Arguments
- p: the number of nodes.
- prob: probability of connecting a node to another node.
- top.sort: logical indicating whether the output graph should be topologically sorted, meaning that arrows always point from lower to higher node indices.
- normalize: logical indicating whether weights and error variances should be normalized such that the diagonal of the corresponding observational covariance matrix is 1.
- lbe, ube: lower and upper bounds of the absolute values of edge weights.
- neg.coef: logical indicating whether negative edge weights are also admissible.
- labels: (optional) character vector of variable (or “node”) names.
- lbv, ubv: lower and upper bound on error variances of the noise terms in the structural equations.

See Also
fci and pdsep which both use this function.
Details

The underlying directed acyclic graph (DAG) is generated by drawing an undirected graph from an Erdős-Rényi model orienting the edges according to a random topological ordering drawn uniformly from the set of permutations of \( p \) variables. This means that any two nodes are connected with (the same) probability \( \text{prob} \), and that the connectivity of different pairs of nodes is independent.

A Gaussian causal model can be represented as a set of linear structural equations. The regression coefficients of the model can be represented as "edge weights" of the DAG. Edge weights are drawn uniformly and independently from the interval between \( \text{lbw} \) and \( \text{ubw} \); if \( \text{neg.coef} = \text{TRUE} \), their sign is flipped with probability 0.5. Error variances are drawn uniformly and independently from the interval between \( \text{lbv} \) and \( \text{ubv} \).

If \( \text{normalize} = \text{TRUE} \), the edge weights and error variances are normalized \textit{in the end} to ensure that the diagonal elements of the observational covariance matrix are all 1; the procedure used is described in Hauser and Bühlmann (2012). Note that in this case the error variances and edge weights are no longer guaranteed to lie in the specified intervals \textit{after normalization}.

Value

An object of class "GaussParDAG".

Author(s)

Alain Hauser (<alain.hauser@bfh.ch>)

References


See Also

\texttt{GaussParDAG}, \texttt{randomDAG}

Examples

```r
set.seed(307)

## Plot some random DAGs
if (require(Rgraphviz)) {
  ## Topologically sorted random DAG
  myDAG <- r.gauss.pardag(p = 10, prob = 0.2, top.sort = TRUE)
  plot(myDAG)

  ## Unsorted DAG
  myDAG <- r.gauss.pardag(p = 10, prob = 0.2, top.sort = FALSE)
  plot(myDAG)
}
```
Without normalization, edge weights and error variances lie within the specified borders:

```r
set.seed(307)
myDAG <- r.gauss.pardag(p = 10, prob = 0.4, lbe = 0.1, ube = 1, lbv = 0.5, ubv = 1.5, neg.coef = FALSE)
B <- myDAG$weight.mat()
V <- myDAG$err.var()
any((B > 0 & B < 0.1) | B > 1)
any(V < 0.5 | V > 1.5)
```

After normalization, edge weights and error variances are not necessarily within the specified range, but the diagonal of the observational covariance matrix consists of ones only:

```r
set.seed(308)
myDAG <- r.gauss.pardag(p = 10, prob = 0.4, normalize = TRUE, lbe = 0.1, ube = 1, lbv = 0.5, ubv = 1.5, neg.coef = FALSE)
B <- myDAG$weight.mat()
V <- myDAG$err.var()
any((B > 0 & B < 0.1) | B > 1)
any(V < 0.5 | V > 1.5)
diag(myDAG$cov.mat())
```

---

**randDAG**

**Random DAG Generation**

**Description**

Generating random directed acyclic graphs (DAGs) with fixed expected number of neighbours. Several different methods are provided, each intentionally biased towards certain properties. The methods are based on the analogue *.game functions in the igraph package.

**Usage**

```r
randDAG(n, d, method = "er", par1=NULL, par2=NULL, DAG = TRUE, weighted = TRUE, wFUN = list(runif, min=0.1, max=1))
```

**Arguments**

- `n` integer, at least 2, indicating the number of nodes in the DAG.
- `d` a positive number, corresponding to the expected number of neighbours per node, more precisely the expected sum of the in- and out-degree.
- `method` a string, specifying the method used for generating the random graph. See details below.
- `par1`, `par2` optional additional arguments, dependent on the method. See details.
- `DAG` logical, if TRUE, labelled graph is directed to a labelled acyclic graph.
- `weighted` logical indicating if edge weights are computed according to `wFUN`. 
a function for computing the edge weights in the DAG. It takes as first argument a number of edges \( m \) for which it returns a vector of length \( m \) containing the weights. Alternatively, \( wFUN \) can be a list consisting of the function in the first entry and of further arguments of the function in the additional entries. The default (only if \( \text{weighted} \) is true) is a uniform weight in \([0, 1]\). See the examples for more.

Details

A (weighted) random graph with \( n \) nodes and expected number of neighbours \( d \) is constructed. For \( \text{DAG} = \text{TRUE} \), the graph is oriented to a DAG. There are eight different random graph models provided, each selectable by the parameters \( \text{method} \), \( \text{par1} \) and \( \text{par2} \), with \( \text{method} \), a string, taking one of the following values:

- **regular**: Graph where every node has exactly \( d \) incident edges. \( \text{par1} \) and \( \text{par2} \) are not used.
- **watts**: Watts-Strogatz graph that interpolates between the regular (\( \text{par1}\to0 \)) and Erdos-Renyi graph (\( \text{par1}\to1 \)). The parameter \( \text{par1} \) is per default \( 0.5 \) and has to be in \((0, 1)\). \( \text{par2} \) is not used.
- **er**: Erdos-Renyi graph where every edge is present independently. \( \text{par1} \) and \( \text{par2} \) are not used.
- **power**: A graph with power-law degree distribution with expectation \( d \).\( \text{par1} \) and \( \text{par2} \) are not used.
- **bipartite**: Bipartite graph with at least \( \text{par1} \times n \) nodes in group 1 and at most \((1-\text{par1})\times n \) nodes in group 2. The argument \( \text{par1} \) has to be in \([0, 1]\) and is per default \( 0.5 \). \( \text{par2} \) is not used.
- **barabasi**: A graph with power-law degree distribution and preferential attachement according to parameter \( \text{par1} \). It must hold that \( \text{par1} \geq 1 \) and the default is \( \text{par1}=1 \). \( \text{par2} \) is not used.
- **geometric**: A geometric random graph in dimension \( \text{par1} \), where \( \text{par1} \) can take values from \{2, 3, 4, 5\} and is per default \( 2 \). If \( \text{par2} = \text{"geo"} \) and \( \text{weighted} = \text{TRUE} \), then the weights are computed according to the Euclidean distance. There are currently no other option for \( \text{par2} \) implemented.
- **interEr**: A graph with \( \text{par1} \) islands of Erdos-Renyi graphs, every pair of those connected by a certain number of edges proportional to \( \text{par2} \) (fraction of inter-connectivity). It is required that \( n/s \) be integer and \( \text{par2} \) in \((0, 1)\). Defaults are \( \text{par1}=2 \) and \( \text{par2}=0.25 \), respectively.

Value

A graph object of class \text{graphNEL}.

Note

The output is not topologically sorted (as opposed to the output of \text{randomDAG}).

Author(s)

Markus Kalisch (<kalisch@stat.math.ethz.ch>) and Manuel Schuerch.

References

These methods are mainly based on the analogue functions in the \text{igraph} package.
See Also

the package igraph, notably help pages such as random.graph.game or barabasi.game; unifDAG from package unifDAG for generating uniform random DAGs. randomDAG a limited and soon deprecated version of randDAG; rmvDAG for generating multivariate data according to a DAG.

Examples

```r
set.seed(37)
dag1 <- randDAG(10, 4, "regular")
dag2 <- randDAG(10, 4, "watts")
dag3 <- randDAG(10, 4, "er")
dag4 <- randDAG(10, 4, "power")
dag5 <- randDAG(10, 4, "bipartite")
dag6 <- randDAG(10, 4, "barabasi")
dag7 <- randDAG(10, 4, "geometric")
dag8 <- randDAG(10, 4, "interEr", par2 = 0.5)

## require("Rgraphviz")
par(mfrow=c(4,2))
plot(dag1,main="Regular graph")
plot(dag2,main="Watts-Strogatz graph")
plot(dag3,main="Erdos-Renyi graph")
plot(dag4,main="Power-law graph")
plot(dag5,main="Bipartite graph")
plot(dag6,main="Barabasi graph")
plot(dag7,main="Geometric random graph")
plot(dag8,main="Interconnected island graph")

set.seed(45)
dag0 <- randDAG(6,3)
dag1 <- randDAG(6,3, weighted=FALSE)
dag2 <- randDAG(6,3, DAG=FALSE)
par(mfrow=c(1,2))
plot(dag1)
plot(dag2)  ## undirected graph
dag@edgeData ## note the uniform weights between 0.1 and 1
dag1@edgeData ## note the constant weights

wFUN <- function(m,lB,uB) { runif(m,lB,uB) }
dag <- randDAG(6,3,wFUN=list(wFUN,1,4))
dag@edgeData  ## note the uniform weights between 1 and 4
```

randomDAG

Generate a Directed Acyclic Graph (DAG) randomly

Description

Generate a random Directed Acyclic Graph (DAG). The resulting graph is topologically ordered from low to high node numbers.
Usage

randomDAG(n, prob, lB = 0.1, uB = 1, V = as.character(1:n))

Arguments

n  Number of nodes, \( n \geq 2 \).
prob  Probability of connecting a node to another node with higher topological ordering.
lB, uB  Lower and upper limit of edge weights, chosen uniformly at random, i.e., by runif(., min=lB, max=uB).
V  character vector length \( n \) of node names.

Details

The \( n \) nodes are ordered. Start with first node. Let the number of nodes with higher order be \( k \). Then, the number of neighbouring nodes is drawn as Bin(\( k \), prob). The neighbours are then drawn without replacement from the nodes with higher order. For each node, a weight is uniformly sampled from \( lB \) to \( uB \). This procedure is repeated for the next node in the original ordering and so on.

Value

An object of class "graphNEL", see graph-class from package graph, with \( n \) named ("1" to "n") nodes and directed edges. The graph is topologically ordered. Each edge has a weight between \( lB \) and \( uB \).

Author(s)

Markus Kalisch (<kalisch@stat.math.ethz.ch>) and Martin Maechler

See Also

randDAG for a more elaborate version of this function; rmvDAG for generating data according to a DAG; compareGraphs for comparing the skeleton of a DAG with some other undirected graph (in terms of TPR, FPR and TDR).

Examples

set.seed(101)
myDAG <- randomDAG(n = 20, prob = 0.2, lB = 0.1, uB = 1)
## require(Rgraphviz)
plot(myDAG)
**rfci**  
*Estimate an RFCI-PAG using the RFCI Algorithm*

**Description**

Estimate an RFCI-PAG from observational data, using the RFCI-algorithm.

**Usage**

```r
rfci(suffStat, indepTest, alpha, labels, p, 
skel.method = c("stable", "original", "stable.fast"),
fixedGaps = NULL, fixedEdges = NULL, NAdelete = TRUE,
m.max = Inf, rules = rep(TRUE, 10),
conservative = FALSE, maj.rule = FALSE,
numCores = 1, verbose = FALSE)
```

**Arguments**

- `suffStat`: Sufficient statistics: List containing all necessary elements for the conditional independence decisions in the function `indepTest`.
- `indepTest`: Predefined function for testing conditional independence. The function is internally called as `indepTest(x,y,S,suffStat)`, and tests conditional independence of `x` and `y` given `S`. Here, `x` and `y` are variables, and `S` is a (possibly empty) vector of variables (all variables are denoted by their column numbers in the adjacency matrix). `suffStat` is a list containing all relevant elements for the conditional independence decisions. The return value of `indepTest` is the `p-value` of the test for conditional independence.
- `alpha`: significance level (number in `(0, 1)` for the individual conditional independence tests.
- `labels`: (optional) character vector of variable (or “node”) names. Typically preferred to specifying `p`.
- `p`: (optional) number of variables (or nodes). May be specified if `labels` are not, in which case `labels` is set to `1:p`.
- `skel.method`: Character string specifying method; the default, "stable" provides an order-independent skeleton, see `skeleton`.
- `fixedGaps`: A logical matrix of dimension `p*p`. If entry `[i,j]` or `[j,i]` (or both) are TRUE, the edge `i-j` is removed before starting the algorithm. Therefore, this edge is guaranteed to be absent in the resulting graph.
- `fixedEdges`: A logical matrix of dimension `p*p`. If entry `[i,j]` or `[j,i]` (or both) are TRUE, the edge `i-j` is never considered for removal. Therefore, this edge is guaranteed to be present in the resulting graph.
- `NAdelete`: If `indepTest` returns `NA` and this option is `TRUE`, the corresponding edge is deleted. If this option is `FALSE`, the edge is not deleted.
- `m.max`: Maximum size of the conditioning sets that are considered in the conditional independence tests.
rfci

rules Logical vector of length 10 indicating which rules should be used when directing edges. The order of the rules is taken from Zhang (2009).

conservative Logical indicating if the unshielded triples should be checked for ambiguity after the skeleton has been found, similar to the conservative PC algorithm.

maj.rule Logical indicating if the unshielded triples should be checked for ambiguity after the skeleton has been found using a majority rule idea, which is less strict than the conservative.

numCores Specifies the number of cores to be used for parallel estimation of skeleton.

verbose If true, more detailed output is provided.

Details
This function is rather similar to fci. However, it does not compute any Possible-D-SEP sets and thus does not make tests conditioning on subsets of Possible-D-SEP. This makes RFCI much faster than FCI. The orientation rules for v-structures and rule 4 were modified in order to produce an RFCI-PAG which, in the oracle version, is guaranteed to have the correct ancestral relationships.

The first part of the RFCI algorithm is analogous to the PC and FCI algorithm. It starts with a complete undirected graph and estimates an initial skeleton using the function skeleton, which produces an initial order-independent skeleton, see skeleton for more details. All edges of this skeleton are of the form o-o. Due to the presence of hidden variables, it is no longer sufficient to consider only subsets of the neighborhoods of nodes x and y to decide whether the edge x o-o y should be removed. The FCI algorithm performs independence tests conditioning on subsets of Possible-D-SEP to remove those edges. Since this procedure is computationally infeasible, the RFCI algorithm uses a different approach to remove some of those superfluous edges before orienting the v-structures and the discriminating paths in orientation rule 4.

Before orienting the v-structures, we perform the following additional conditional independence tests. For each unshielded triple a-b-c in the initial skeleton, we check if both a and b and b and c are conditionally dependent given the separating of a and c (sepset(a,c)). These conditional dependencies may not have been checked while estimating the initial skeleton, since sepset(a,c) does not need to be a subset of the neighbors of a or of the neighbors of c. If both conditional dependencies hold and b is not in the sepset(a,c), the triple is oriented as a v-structure a->b<-c. On the other hand, if an additional conditional independence relationship may be detected, say a is independent from b given the sepset(a,c), the edge between a and c is removed from the graph and the set responsible for that is saved in sepset(a,b). The removal of an edge can destroy or create new unshielded triples in the graph. To solve this problem we work with lists (for details see Colombo et al., 2012).

Before orienting discriminating paths, we perform the following additional conditional independence tests. For each triple a <-* b o-*c with a -> c, the algorithm searches for a discriminating path p = <d, . . . , a,b,c> for b of minimal length, and checks that the vertices in every consecutive pair (f1,f2) on p are conditionally dependent given all subsets of sepset(d,c) \ f1,f2 . If we do not find any conditional independence relationship, the path is oriented as in rule (R4). If one or more conditional independence relationships are found, the corresponding edges are removed, their minimal separating sets are stored.

Conservative RFCI can be computed if the argument of conservative is TRUE. After the final skeleton is computed and the additional local tests on all unshielded triples, as described above, have been done, all potential v-structures a-b-c are checked in the following way. We test whether a and c are independent conditioning on any subset of the neighbors of a or any subset of the
neighbors of c. When a subset makes a and c conditionally independent, we call it a separating set. If b is in no such separating set or in all such separating sets, no further action is taken and the normal version of the RFCI algorithm is continued. If, however, b is in only some separating sets, the triple a-b-c is marked ‘ambiguous’. If a is independent of c given some S in the skeleton (i.e., the edge a-c dropped out), but a and c remain dependent given all subsets of neighbors of either a or c, we will call all triples a-b-c ‘unambiguous’. This is because in the RFCI algorithm, the true separating set might be outside the neighborhood of either a or c. An ambiguous triple is not oriented as a v-structure. Furthermore, no further orientation rule that needs to know whether a-b-c is a v-structure or not is applied. Instead of using the conservative version, which is quite strict towards the v-structures, Colombo and Maathuis (2014) introduced a less strict version for the v-structures called majority rule. This adaptation can be called using maj.rule = TRUE. In this case, the triple a-b-c is marked as ‘ambiguous’ if and only if b is in exactly 50 percent of such separating sets or no separating set was found. If b is in less than 50 percent of the separating sets it is set as a v-structure, and if in more than 50 percent it is set as a non v-structure (for more details see Colombo and Maathuis, 2014).

The implementation uses the stabilized skeleton skeleton, which produces an initial order-independent skeleton. The final skeleton and edge orientations can still be order-dependent, see Colombo and Maathuis (2014).

Value

An object of class fciAlgo (see fciAlgo) containing the estimated graph (in the form of an adjacency matrix with various possible edge marks), the conditioning sets that lead to edge removals (sepset) and several other parameters.

Author(s)

Diego Colombo and Markus Kalisch (<kalisch@stat.math.ethz.ch>).

References


See Also

fci and fciPlus for estimating a PAG using the FCI algorithm; skeleton for estimating an initial skeleton using the RFCI algorithm; pc for estimating a CPDAG using the PC algorithm; gaussCItest, disCItest, binCItest and dsepTest as examples for indepTest.

Examples

# Example without latent variables
set.seed(42)
p <- 7
## generate and draw random DAG :
myDAG <- randomDAG(p, prob = 0.4)

## find skeleton and PAG using the RFCI algorithm
suffStat <- list(C = cov2cor(trueCov(myDAG)), n = 10^9)
indepTest <- gaussCItest
res <- rfci(suffStat, indepTest, alpha = 0.9999, p=p, verbose=TRUE)

##################################################% --------------
## Example with hidden variables
## Zhang (2008), Fig. 6, p.1882
##################################################

## create the DAG :
V <- LETTERS[1:5]
edL <- setNames(vector("list", length = 5), V)
edL[[1]] <- list(edges=c(2,4),weights=c(1,1))
edL[[2]] <- list(edges=3,weights=c(1))
edL[[3]] <- list(edges=5,weights=c(1))
edL[[4]] <- list(edges=5,weights=c(1))
# and leave edL[[ 5 ]] empty
g <- new("graphNEL", nodes=V, edgeL=edL, edgemode="directed")
if (require(Rgraphviz))
  plot(g)

## define the latent variable
L <- 1

## compute the true covariance matrix of g
cov.mat <- trueCov(g)
## delete rows and columns belonging to latent variable L
true.cov <- cov.mat[-L,-L]
## transform covariance matrix into a correlation matrix
true.corr <- cov2cor(true.cov)

## find PAG with RFCI algorithm
## as dependence "oracle", we use the true correlation matrix in
gaussCItest() with a large "virtual sample size" and a large alpha :
rfci.pag <- rfci(suffStat = list(C = true.corr, n = 10^9),
                indepTest = gaussCItest, alpha = 0.9999, labels = V[-L],
                verbose=TRUE)

## define PAG given in Zhang (2008), Fig. 6, p.1882
corr.pag <- rbind(c(0,1,1,0),
                  c(1,0,0,2),
                  c(1,0,0,2),
                  c(0,3,3,0))

## check that estimated and correct PAG are in agreement:
stopifnot(corr.pag == rfci.pag@amat)
Generate Multivariate Data according to a DAG

Description

Generate multivariate data with dependency structure specified by a (given) DAG (Directed Acyclic Graph) with nodes corresponding to random variables. The DAG has to be topologically ordered.

Usage

```r
rmvDAG(n, dag,
  errDist = c("normal", "cauchy", "t4", "mix", "mixt3", "mixN100"),
  mix = 0.1, errMat = NULL, back.compatible = FALSE,
  use.node.names = !back.compatible)
```

Arguments

- `n`: number of samples that should be drawn. (integer)
- `dag`: a graph object describing the DAG; must contain weights for all the edges. The nodes must be topologically sorted. (For topological sorting use `tsort` from the RBGL package.)
- `errDist`: string specifying the distribution of each node. Currently, the options "normal", "t4", "cauchy", "mix", "mixt3" and "mixN100" are supported. The first three generate standard normal-, t(df=4)- and cauchy-random numbers. The options containing the word "mix" create standard normal random variables with a mix of outliers. The outliers for the options "mix", "mixt3", "mixN100" are drawn from a standard cauchy, t(df=3) and N(0,100) distribution, respectively. The fraction of outliers is determined by the `mix` argument.
- `mix`: for the "mix*" error distribution, `mix` specifies the fraction of "outlier" samples (i.e., Cauchy, t3 or N(0,100)).
- `errMat`: numeric `n` x `p` matrix specifying the error vectors `e_i` (see Details), instead of specifying `errDist` (and maybe `mix`).
- `back.compatible`: logical indicating if the data generated should be the same as with `pcalg` version 1.0-6 and earlier (where `wgtMatrix()` differed).
- `use.node.names`: logical indicating if the column names of the result matrix should equal `nodes(dag)`, very sensibly, but new, hence the default.

Details

Each node is visited in the topological order. For each node `i` we generate a `p`-dimensional value `X_i` in the following way: Let `X_1, ..., X_k` denote the values of all neighbours of `i` with lower order. Let `w_1, ..., w_k` be the weights of the corresponding edges. Furthermore, generate a random vector `E_i` according to the specified error distribution. Then, the value of `X_i` is computed as

```
X_i = w_1 * X_1 + ... + w_k * X_k + E_i.
```

If node `i` has no neighbors with lower order, `X_i = E_i` is set.
Value

A \( n \times p \) matrix with the generated data. The \( p \) columns correspond to the nodes (i.e., random variables) and each of the \( n \) rows correspond to a sample.

Author(s)

Markus Kalisch (<kalisch@stat.math.ethz.ch>) and Martin Maechler.

See Also

randomDAG for generating a random DAG; skeleton and pc for estimating the skeleton and the CPDAG of a DAG that corresponds to the data.

Examples

```r
## generate random DAG
p <- 20
rDAG <- randomDAG(p, prob = 0.2, lB=0.1, uB=1)

if (require(Rgraphviz)) {
  ## plot the DAG
  plot(rDAG, main = "randomDAG(20, prob = 0.2, ..")
}

## generate 1000 samples of DAG using standard normal error distribution
n <- 1000
d.normMat <- rmvDAG(n, rDAG, errDist="normal")

## generate 1000 samples of DAG using standard t(df=4) error distribution
d.t4Mat <- rmvDAG(n, rDAG, errDist="t4")

## generate 1000 samples of DAG using standard normal with a cauchy mixture of 30 percent
## generate 1000 samples of DAG using standard normal with a cauchy mixture of 30 percent
d.mixMat <- rmvDAG(n, rDAG, errDist="mix",mix=0.3)

require(MASS) ## for mvrnorm()
Sigma <- toeplitz(ARMAacf(0.2, lag.max = p - 1))
dim(Sigma)# p x p
## *Correlated* normal error matrix "e_i" (against model assumption)
eMat <- mvrnorm(n, mu = rep(0, p), Sigma = Sigma)
d.CnormMat <- rmvDAG(n, rDAG, errMat = eMat)
```

---

**rmvnorm.ivent**

Simulate from a Gaussian Causal Model

**Description**

Produces one or more samples from the observational or an interventional distribution associated to a Gaussian causal model.
Score-class

Usage

\texttt{rmvnorm.ivent(n, object, target = integer(0), target.value = numeric(0))}

Arguments

- \textit{n}: Number of samples required.
- \textit{object}: An instance of \texttt{GaussParDAG}
- \textit{target}: Intervention target: vector of intervened nodes. If the vector is empty, samples from the observational distribution are generated. Otherwise, samples from an interventional distribution are simulated.
- \textit{target.value}: Values of the intervened variables. If \textit{target.value} is a vector of the same length as \textit{target}, the indicated intervention levels are used for all \textit{n} samples. If \textit{target.value} is a matrix of dimension \textit{n} by \text{length(target)}, the \textit{i}-th sample is simulated using the \textit{i}-th row of the matrix as intervention levels.

Value

- If \textit{n} = 1 a vector of length \(p\) is returned, where \(p\) denotes the number of nodes of \textit{object}. Otherwise an \(n\) by \(p\) matrix is returned with one sample per row.

Author(s)

Alain Hauser (<alain.hauser@bfh.ch>)

Examples

\begin{verbatim}
set.seed(307)
myDAG <- r.gauss.pardag(5, 0.5)
var(rmvnorm.ivent(n = 1000, myDAG))
myDAG$cov.mat()
var(rmvnorm.ivent(n = 1000, myDAG, target = 1, target.value = 1))
myDAG$cov.mat(target = 1, ivent.var = 0)
\end{verbatim}

Score-class

\textbf{Virtual Class "Score"}

Description

This virtual base class represents a score for causal inference; it is used in the causal inference functions \texttt{ges, gies} and \texttt{simy}.
Score-class

Details

Score-based structure learning algorithms for causal inference such as Greedy Equivalence Search (GES, implemented in the function ges), Greedy Interventional Equivalence Search (GIES, implemented in the function gies) and the dynamic programming approach of Silander and Myllymäki (2006) (implemented in the function simy) try to find the DAG model which maximizes a scoring criterion for a given data set. A widely-used scoring criterion is the Bayesian Information Criterion (BIC).

The virtual class Score is the base class for providing a scoring criterion to the mentioned causal inference algorithms. It does not implement a concrete scoring criterion, but it defines the functions that must be provided by its descendants (cf. methods).

Knowledge of this class is only required if you aim to implement an own scoring criterion. At the moment, it is recommended to use the predefined scoring criteria for multivariate Gaussian data derived from Score, GaussL0penIntScore and GaussL0penObsScore.

Fields

The fields of Score are mainly of interest for users who aim at deriving an own class from this virtual base class, i.e., implementing an own score function.

.data: Node labels. They are passed to causal inference methods by default to label the nodes of the resulting graph.

decom: Indicates whether the represented score is decomposable (cf. details). At the moment, only decomposable scores are supported by the implementation of the causal inference algorithms; support for non-decomposable scores is planned.

.targets: List representing the preprocessed input data; this is typically a statistic which is sufficient for the calculation of the score.

.pardag.class: Name of the class of the parametric DAG model corresponding to the score. This must name a class derived from ParDAG.

c.fcn: Only used internally; must remain empty for (user specified) classes derived from Score.

Constructor

new("Score",
  data = matrix(1, 1, 1),
  targets = list(integer(0)),
  target.index = rep(as.integer(1), nrow(data)),
  nodes = colnames(data),
  ...
)

data Data matrix with \( n \) rows and \( p \) columns. Each row corresponds to one realization, either interventional or observational.

targets List of mutually exclusive intervention targets that have been used for data generation.

target.index Vector of length \( n \); the \( i \)-th entry specifies the index of the intervention target in targets under which the \( i \)-th row of data was measured.

.nodes Node labels

... Additional parameters used by derived (and non-virtual) classes.
Methods

Note that since Score is a virtual class, its methods cannot be called directly, but only on derived classes.

local.score(vertex, parents, ...) For decomposable scores, this function calculates the local score of a vertex and its parents. Must throw an error in derived classes that do not represent a decomposable score.

global.score.int(edges, ...) Calculates the global score of a DAG, represented as a list of in-edges: for each vertex in the DAG, this list contains a vector of parents.

global.score(dag, ...) Calculates the global score of a DAG, represented as object of a class derived from ParDAG.

local.fit(vertex, parents, ...) Calculates a local model fit of a vertex and its parents, e.g. by MLE. The result is a vector of parameters whose meaning depends on the model class; it matches the convention used in the corresponding causal model (cf. .pardag.class).

global.fit(dag, ...) Calculates the global MLE of a DAG, represented by an object of the class specified by .pardag.class. The result is a list of vectors, one per vertex, each in the same format as the result vector of local.mle.

Author(s)

Alain Hauser (<alain.hauser@bfh.ch>)

References


See Also

ges, gies, simy, GaussL0penIntScore, GaussL0penObsScore

shd

Compute Structural Hamming Distance (SHD)

Description

Compute the Structural Hamming Distance (SHD) between two graphs. In simple terms, this is the number of edge insertions, deletions or flips in order to transform one graph to another graph.

Usage

shd(g1, g2)
showAmat

Arguments

  g1  Graph object
  g2  Graph object

Value

  The value of the SHD (numeric).

Author(s)

  Markus Kalisch <kalisch@stat.math.ethz.ch> and Martin Maechler

References


Examples

  ## generate two graphs
  g1 <- randomDAG(10, prob = 0.2)
  g2 <- randomDAG(10, prob = 0.2)
  ## compute SHD
  (shd.val <- shd(g1,g2))

showAmat  

*Show Adjacency Matrix of pcAlgo object*

Description

  This function is deprecated - Use `as(*, "amat")` instead!
  Show the adjacency matrix of a "pcAlgo" object; this is intended to be an alternative if the `Rgraphviz` package does not work.

Usage

  showAmat(object)

Arguments

  object  an R object of class `pcAlgo`, as returned from `skeleton()` or `pc()`.

Value

  The adjacency matrix.
# showEdgeList

**Note**

For "fciAlgo" objects, the show method produces a similar result.

**Author(s)**

Markus Kalisch (<kalisch@stat.math.ethz.ch>)

**See Also**

showEdgeList for showing the edge list of a pcAlgo object. iplotPC for plotting a "pcAlgo" object using the package igraph also for an example of showAmat().

---

**showEdgeList** | **Show Edge List of pcAlgo object**
---

**Description**

Show the list of edges (of the graph) of a pcAlgo object; this is intended to be an alternative if Rgraphviz does not work.

**Usage**

```r
showEdgeList(object, labels = NULL)
```

**Arguments**

- `object` an R object of class pcAlgo, as returned from skeleton() or pc().
- `labels` optional labels for nodes; by default, the labels from the object are used.

**Value**

none; the purpose is in (the side effect of) printing the edge list.

**Note**

This is not quite ok for "fciAlgo" objects, yet.

**Author(s)**

Markus Kalisch (<kalisch@stat.math.ethz.ch>)

**See Also**

showAmat for the adjacency matrix of a pcAlgo object. iplotPC for plotting a pcAlgo object using the package igraph, also for an example of showEdgeList().
Estimate Interventional Markov Equivalence Class of a DAG

Description

Estimate the interventional essential graph representing the Markov equivalence class of a DAG using the dynamic programming (DP) approach of Silander and Myllymäki (2006). This algorithm maximizes a decomposable scoring criterion in exponential runtime.

Usage

```r
simy(score, labels = score$getNodes(), targets = score$getTargets(),
       verbose = FALSE, ...)
```

Arguments

- `score`: An instance of a class derived from `Score`.
- `labels`: Node labels; by default, they are determined from the scoring object.
- `targets`: A list of intervention targets (cf. details). A list of vectors, each vector listing the vertices of one intervention target.
- `verbose`: if `TRUE`, detailed output is provided.
- `...`: Additional arguments for debugging purposes and fine tuning.

Details

This function estimates the interventional Markov equivalence class of a DAG based on a data sample with interventional data originating from various interventions and possibly observational data. The intervention targets used for data generation must be specified by the argument `targets` as a list of (integer) vectors listing the intervened vertices; observational data is specified by an empty set, i.e. a vector of the form `integer(0)`. As an example, if data contains observational samples as well as samples originating from an intervention at vertices 1 and 4, the intervention targets must be specified as `list(integer(0),as.integer(1),as.integer(c(1,4)))`.

An interventional Markov equivalence class of DAGs can be uniquely represented by a partially directed graph called interventional essential graph. Its edges have the following interpretation:

1. a directed edge $a \rightarrow b$ stands for an arrow that has the same orientation in all representatives of the interventional Markov equivalence class;
2. an undirected edge $a - b$ stands for an arrow that is oriented in one way in some representatives of the equivalence class and in the other way in other representatives of the equivalence class.

Note that when plotting the object, undirected and bidirected edges are equivalent.

The DP approach of Silander and Myllymäki (2006) is a score-based algorithm that guarantees to find the optimum of any decomposable scoring criterion. Its CPU and memory consumption grow exponentially with the number of variables in the system, irrespective of the sparseness of the true or estimated DAG. The implementation in the pcalg package is feasible up to approximately 20 variables, depending on the user’s computer.
Value

`simy` returns a list with the following two components:

- **essgraph**: An object of class `EssGraph` containing an estimate of the equivalence class of the underlying DAG.
- **repr**: An object of a class derived from `ParDAG` containing a (random) representative of the estimated equivalence class.

Author(s)

Alain Hauser (<alain.hauser@bfh.ch>)

References


See Also

gies, Score, EssGraph

Examples

```r
########################################################################
## Using Gaussian Data
########################################################################
## Load predefined data
data(gmInt)

## Define the score (BIC)
score <- new("GaussL0penIntScore", gmInt$x, gmInt$targets, gmInt$target.index)

## Estimate the essential graph
simy.fit <- simy(score)
eDAG <- simy.fit$essgraph
as(eDAG, "graph")

## Look at the graph incidence matrix (a "sparseMatrix"):
if(require(Matrix))
  show( as(as(eDAG, "graphNEL"), "Matrix") )

## Plot the estimated essential graph and the true DAG
if (require(Rgraphviz)) {
  par(mfrow=c(1,2))
  plot(eDAG, main = "Estimated ess. graph")
  plot(gmInt$g, main = "True DAG")
}
```
Estimate (Initial) Skeleton of a DAG using the PC / PC-Stable Algorithm

Description

Estimate the skeleton of a DAG without latent and selection variables using the PC Algorithm or estimate an initial skeleton of a DAG with arbitrarily many latent and selection variables using the FCI and the RFCI algorithms.

If used in the PC algorithm, it estimates the order-independent “PC-stable” (“stable”) or original PC (“original”) “skeleton” of a directed acyclic graph (DAG) from observational data.

When used in the FCI and RFCI algorithms, this function estimates only an initial order-independent (or PC original) “skeleton”. Because of the presence of latent and selection variables, to find the final skeleton those algorithms need to perform additional tests later on and consequently some edges can be further deleted.

Usage

skeleton(suffStat, indepTest, alpha, labels, p, method = c("stable", "original", "stable.fast"), m.max = Inf, fixedGaps = NULL, fixedEdges = NULL, NAdelete = TRUE, numCores = 1, verbose = FALSE)

Arguments

suffStat Sufficient statistics: List containing all necessary elements for the conditional independence decisions in the function indepTest.

indepTest Predefined function for testing conditional independence. The function is internally called as indepTest(x,y,S,suffStat) and tests conditional independence of x and y given S. Here, x and y are variables, and S is a (possibly empty) vector of variables (all variables are denoted by their column numbers in the adjacency matrix). suffStat is a list containing all relevant elements for the conditional independence decisions. The return value of indepTest is the p-value of the test for conditional independence.

alpha significance level (number in (0, 1) for the individual conditional independence tests.

labels (optional) character vector of variable (or “node”) names. Typically preferred to specifying p.

p (optional) number of variables (or nodes). May be specified if labels are not, in which case labels is set to 1:p.

method Character string specifying method; the default, "stable" provides an order-independent skeleton, see ‘Details’ below.

m.max Maximal size of the conditioning sets that are considered in the conditional independence tests.
fixedGaps logical symmetric matrix of dimension p*p. If entry [i,j] is true, the edge \( i - j \) is removed before starting the algorithm. Therefore, this edge is guaranteed to be absent in the resulting graph.

fixedEdges a logical symmetric matrix of dimension p*p. If entry [i,j] is true, the edge \( i - j \) is never considered for removal. Therefore, this edge is guaranteed to be present in the resulting graph.

NAdelete logical needed for the case \( \text{indepTest}(\cdot) \) returns NA. If it is true, the corresponding edge is deleted, otherwise not.

numCores number of processor cores to use for parallel computation. Only available for method = "stable.fast".

verbose if TRUE, detailed output is provided.

Details

Under the assumption that the distribution of the observed variables is faithful to a DAG and that there are no latent and selection variables, this function estimates the skeleton of the DAG. The skeleton of a DAG is the undirected graph resulting from removing all arrowheads from the DAG. Edges in the skeleton of a DAG have the following interpretation:

There is an edge between \( i \) and \( j \), \( i - j \), if and only if variables \( i \) and \( j \) are conditionally dependent given \( S \) for all possible subsets \( S \) of the remaining nodes.

On the other hand, the distribution of the observed variables is faithful to a DAG with arbitrarily many latent and selection variables, skeleton() estimates the initial skeleton of the DAG. Edges in this initial skeleton of a DAG have the following interpretation:

There is an edge \( i - j \) if and only if variables \( i \) and \( j \) are conditionally dependent given \( S \) for all possible subsets \( S \) of the neighbours of \( i \) and the neighbours of \( j \).

The data are not required to follow a specific distribution, but one should make sure that the conditional independence test used in \( \text{indepTest} \) is appropriate for the data. Pre-programmed versions of \( \text{indepTest} \) are available for Gaussian data (\( \text{gaussCItest} \)), discrete data (\( \text{disCItest} \)), and binary data (see \( \text{binCItest} \)). Users may also specify their own \( \text{indepTest} \) function.

The PC algorithm (Spirtes, Glymour and Scheines, 2000) (method = "original") is known to be order-dependent, in the sense that the output may depend on the order in which the variables are given. Therefore, Colombo and Maathuis (2014) proposed a simple modification, called “PC-stable”, which yields order-independent adjacencies in the skeleton, provided by \( \text{pc}() \) with the new default method = "stable". This stable variant of the algorithm is also available with the method = "stable.fast": it runs the algorithm of Colombo and Maathuis (2014) faster than method = "stable" in general, but should be regarded as an experimental option at the moment.

The algorithm starts with a complete undirected graph. In each step, it visits all pairs \( (i, j) \) of adjacent nodes in the current graph, and determines based on conditional independence tests whether the edge \( i - j \) should be removed. In particular, for each step \( m \) \((m = 0, 1, \ldots)\) of the size of the conditioning sets, the algorithm at first determines the neighbours \( a(i) \) of each node \( i \) in the graph. Then, the algorithm visits all pairs \( (i, j) \) of adjacent nodes in the current graph, and the edge \( i - j \) is kept if and only if the null hypothesis \( i \) and \( j \) are conditionally independent given \( S \) rejected at significance level \( \alpha \) for all subsets \( S \) of size \( m \) of \( a(i) \) and of \( a(j) \) (as judged by the function \( \text{indepTest} \)). For the "stable" method, the neighborhoods \( a(i) \) are kept fixed within each value of \( m \), and this makes the algorithm order-independent. Method "original", the original PC algorithm would update the neighbour list after each edge change.
The algorithm stops when $m$ is larger than the largest neighbourhood size of all nodes, or when $m$ has reached the limit $m_{\text{max}}$ which may be set by the user.

Since the FCI (Spirtes, Glymour and Scheines, 2000) and RFCI (Colombo et al., 2012) algorithms are built up from the PC algorithm, they are also order-dependent in the skeleton. To resolve their order-dependence issues in the skeleton is more involved, see Colombo and Maathuis (2014). However, now, with method = "stable", this function estimates an initial order-independent skeleton in these algorithms (for additional details on how to make the final skeleton of FCI fully order-independent see fci and Colombo and Maathuis (2014)).

The information in fixedGaps and fixedEdges is used as follows. The gaps given in fixedGaps are introduced in the very beginning of the algorithm by removing the corresponding edges from the complete undirected graph. Pairs $(i,j)$ in fixedEdges are skipped in all steps of the algorithm, so that these edges remain in the graph.

Note: Throughout, the algorithm works with the column positions of the variables in the adjacency matrix, and not with the names of the variables.

Value

An object of class "pcAlgo" (see pcAlgo) containing an estimate of the skeleton of the underlying DAG, the conditioning sets (sepset) that led to edge removals and several other parameters.

Author(s)

Markus Kalisch (<kalisch@stat.math.ethz.ch>), Martin Maechler, Alain Hauser, and Diego Colombo.

References


See Also

pc for generating a partially directed graph using the PC algorithm; fci for generating a partial ancestral graph using the FCI algorithm; rfci for generating a partial ancestral graph using the RFCI algorithm; udag2pdag for converting the skeleton to a CPDAG.

Further, gaussCItest, disCItest, binCItest and dsepTest as examples for indepTest.

Examples

```r
##################################################
## Using Gaussian Data
##################################################
```
## Load predefined data

data(gmG)

n <- nrow(gmG$x)
V <- colnames(gmG$x) # labels aka node names

## estimate Skeleton

skel.fit <- skeleton(suffStat = list(C = cor(gmG$x), n = n),
indepTest = gaussCItest, alpha = 0.01, labels = V, verbose = TRUE)

if (require(Rgraphviz)) {
  # show estimated Skeleton
  par(mfrow=c(1,2))
  plot(skel.fit, main = "Estimated Skeleton")
  plot(gmG$g, main = "True DAG")
}

## Using d-separation oracle

## define sufficient statistics (d-separation oracle)

Ora.stat <- list(g = gmG$g, jp = RBGL::johnson.all.pairs.sp(gmG$g))

## estimate Skeleton

fit.Ora <- skeleton(suffStat=Ora.stat, indepTest = dsepTest, labels = V,
alpha=0.01) # - irrelevant as dsepTest returns either 0 or 1

if (require(Rgraphviz)) {
  # show estimated Skeleton
  plot(fit.Ora, main = "Estimated Skeleton (d-sep oracle)")
  plot(gmG$g, main = "True DAG")
}

## Using discrete data

## Load data

data(gmD)

V <- colnames(gmD$x) # labels aka node names

## define sufficient statistics

suffStat <- list(dm = gmD$x, nlev = c(3,2,3,4,2), adaptDF = FALSE)

## estimate Skeleton

skel.fit <- skeleton(suffStat,
indepTest = disCItest, alpha = 0.01, labels = V, verbose = TRUE)

if (require(Rgraphviz)) {
  # show estimated Skeleton
  par(mfrow = c(1,2))
  plot(skel.fit, main = "Estimated Skeleton")
  plot(gmD$g, main = "True DAG")
}

##############################################################
## Using binary data

### Load binary data
```r
data(gmB)
X <- gmB$x
```

### estimate Skeleton
```r
skel.fm2 <- skeleton(suffStat = list(dm = X, adaptDF = FALSE),
                     indepTest = binCItest, alpha = 0.01,
                     labels = colnames(X), verbose = TRUE)
```

```r
if (require(Rgraphviz)) {
  ## show estimated Skeleton
  par(mfrow = c(1,2))
  plot(skel.fm2, main = "Binary Data 'gmB': Estimated Skeleton")
  plot(gmB$g, main = "True DAG")
}
```

---

**trueCov**  
*Covariance matrix of a DAG.*

### Description

Compute the (true) covariance matrix of a generated DAG.

### Usage

```r
trueCov(dag, back.compatible = FALSE)
```

### Arguments

- **dag**  
  Graph object containing the DAG.

- **back.compatible**  
  Logical indicating if the data generated should be the same as with `pcalg` version 1.0-6 and earlier (where `wgtMatrix()` differed).

### Value

Covariance matrix.

### Note

This function can *not* be used to estimate the covariance matrix from an estimated DAG or corresponding data.

### Author(s)

Markus Kalisch
udag2apag

**Last step of RFCI algorithm: Transform partially oriented graph into RFCI-PAG**

**Description**

This function performs the last step of the RFCI algorithm: It transforms a partially oriented graph in which the v-structures have been oriented into an RFCI Partial Ancestral Graph (PAG) (see Colombo et al (2012)).

While orienting the edges, this function performs some additional conditional independence tests in orientation rule 4 to ensure correctness of the ancestral relationships. As a result of these additional tests, some additional edges can be deleted. The result is the final adjacency matrix indicating also the edge marks and the updated sepsets.

**Usage**

```r
udag2apag(apag, suffStat, indepTest, alpha, sepset,
            rules = rep(TRUE, 10), unfVect = NULL, verbose = FALSE)
```

**Arguments**

- `apag`: Adjacency matrix of type `amat.pag`
- `suffStat`: Sufficient statistics: A `list` containing all necessary elements for the conditional independence decisions in the function `indepTest`.
- `indepTest`: Pre-defined function for testing conditional independence. The function is internally called as `indepTest(x, y, S, suffStat)`, and tests conditional independence of `x` and `y` given `S`. Here, `x` and `y` are variables, and `S` is a (possibly empty) set of variables (all variables are coded by their column numbers in the adjacency matrix).
- `alpha`: Significance level for the conditional independence tests.
- `sepset`: Sepset vector.
- `rules`: Rules for deleting edges.
- `unfVect`: Unoriented factors.
- `verbose`: If `TRUE`, additional information is printed.

**Examples**

```r
set.seed(123)
g <- randomDAG(n = 5, prob = 0.3)  ## generate random DAG
if(require(Rgraphviz)) {
  plot(g)
}

## Compute true covariance matrix
trueCov(g)

## For comparison:
## Estimate true covariance matrix after generating data from g
d <- rmvDAG(10000, g)
cov(d)
```
suffStat is a list containing all relevant elements for the conditional independence decisions. The return value of indepTest is the p-value of the test for conditional independence.

alpha  Significance level for the individual conditional independence tests.
sepset  List of length p; each element of the list contains another list of length p. The element sepset[[x]][[y]] contains the separation set that made the edge between x and y drop out. Each separation set is a vector with (integer) positions of variables in the adjacency matrix. This object is thought to be obtained from a pcAlgo-object.

rules  Logical vector of length 10 with TRUE or FALSE for each rule, where TRUE in position i means that rule i (Ri) will be applied. By default, all rules are active.
unfVect  Vector containing numbers that encode the ambiguous triples (as returned by pc.cons.intern). This is needed in the conservative and in the majority rule versions of RFCI.
verbose  Logical indicating if detailed output is to be given.

Details

The partially oriented graph in which the v-structures have been oriented is transformed into an RFCI-PAG using adapted rules of Zhang (2008). This function is similar to udag2pag used to orient the skeleton into a PAG in the FCI algorithm. However, it is slightly more complicated because we perform additional conditional independence tests when applying rule 4, to ensure correctness of the ancestral relationships. As a result, some additional edges can be deleted, see Colombo et al. (2012). Because of these additional tests, we need to give suffStat, indepTest, and alpha as inputs. Since edges can be deleted, the input adjacency matrix apag and the input separating sets sepset can change in this algorithm.

If unfVect = NULL (no ambiguous triples), the orientation rules are applied to each eligible structure until no more edges can be oriented. On the other hand, hand, if one uses conservative or majority rule FCI and ambiguous triples have been found in pc.cons.intern, unfVect contains the numbers of all ambiguous triples in the graph. In this case, the orientation rules take this information into account. For example, if a *-> b o-* c and <a,b,c> is an unambiguous unshielded triple and not a v-structure, then we obtain b -* c (otherwise we would create an additional v-structure). On the other hand, if a *-> b o-* c but <a,b,c> is an ambiguous unshielded triple, then the circle mark at b is not oriented.

Note that the algorithm works with columns’ position of the adjacency matrix and not with the names of the variables.

Note that this function does not resolve possible order-dependence in the application of the orientation rules, see Colombo and Maathuis (2014).

Value

apag  Final adjacency matrix of type amat.pag
sepset  Updated list of separating sets

Author(s)

Diego Colombo and Markus Kalisch (<kalisch@stat.math.ethz.ch>)
References


See Also

rfci, udag2pag, dag2pag, udag2pdag, udag2pdagSpecial, udag2pdagRelaxed

Examples

```r
### Example with hidden variables
### Zhang (2008), Fig. 6, p.1882

## create the DAG:
amat <- t(matrix(c(0,1,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,1,0,5,5),5,5))
V <- LETTERS[1:5]
colnames amat) <- rownames(amat <- V
edL <- setNames(vector("list",length=5), V)
edL[[1]] <- list(edges= c(2,4), weights=c(1,1))
edL[[2]] <- list(edges= 3, weights=c(1))
edL[[3]] <- list(edges= 5, weights=c(1))
edL[[4]] <- list(edges= 5, weights=c(1))
# and leave edL[[ 5 ]] empty
g <- new("graphNEL", nodes=V, edgeL=edL, edgemode="directed")
if (require(Rgraphviz))
  plot(g)

## define the latent variable
L <- 1

## compute the true covariance matrix of g
cov.mat <- trueCov(g)

## delete rows and columns belonging to latent variable L
true.cov <- cov.mat[-L,-L]

## transform covariance matrix into a correlation matrix
true.corr <- cov2cor(true.cov)

n <- 100000
alpha <- 0.01
p <- ncol(true.corr)
```
if (require("MASS")) {
  ## generate 100000 samples of DAG using standard normal error distribution
  set.seed(289)
  d.mat <- mvrnorm(n, mu = rep(0, p), Sigma = true.cov)

  ## estimate the skeleton of given data
  suffStat <- list(C = cor(d.mat), n = n)
  indepTest <- gaussCItest
  resD <- skeleton(suffStat, indepTest, alpha = alpha, labels=colnames(true.corr))

  ## estimate all ordered unshielded triples
  amat.resD <- as(resD@graph, "matrix")
  print(u.t <- find.unsh.triple(amat.resD)) # four of them

  ## check and orient v-structures
  vstrucs <- rfci.vStruc(suffStat, indepTest, alpha=alpha, sepset = resD@sepset, g.amat = amat.resD, unshTripl= u.t$unshTripl, unshVect = u.t$unshVect, verbose = TRUE)

  ## Estimate the final skeleton and extend it into a PAG
  ## (using all 10 rules, as per default):
  resP <- udag2apag(vstrucs$amat, suffStat, indepTest=indepTest, alpha=alpha, sepset=vstrucs$sepset, verbose = TRUE)
  print(Amat <- resP$graph)
} # only if "MASS" is there

---

**udag2pag**

*Last steps of FCI algorithm: Transform Final Skeleton into FCI-PAG*

**Description**

This function perform the last steps of the FCI algorithm, as it transforms an un-oriented final skeleton into a Partial Ancestral Graph (PAG). The final skeleton must have been estimated with `pdsep()`. The result is an adjacency matrix indicating also the edge marks.

**Usage**

```r
udag2pag(pag, sepset, rules = rep(TRUE, 10), unfVect = NULL, verbose = FALSE, orientCollider = TRUE)
```

**Arguments**

- **pag**: Adjacency matrix of type `amat.pag`
- **sepset**: List of length p; each element of the list contains another list of length p. The element `sepset[[x]][[y]]` contains the separation set that made the edge between x and y drop out. Each separation set is a vector with (integer) positions of variables in the adjacency matrix. This object is thought to be obtained from a `pcAlgo`-object.
rules Array of length 10 containing TRUE or FALSE for each rule. TRUE in position i means that rule i (R_i) will be applied. By default, all rules are used.
unfVect Vector containing numbers that encode ambiguous unshielded triples (as returned by pc.cons.intern). This is needed in the conservative and majority rule versions of FCI.
verbose If TRUE, detailed output is provided.
orientCollider if TRUE, collider are oriented.

Details

The skeleton is transformed into an FCI-PAG using rules by Zhang (2008).

If unfVect = NULL (i.e., one uses standard FCI or one uses conservative/majority rule FCI but there are no ambiguous triples), then the orientation rules are applied to each eligible structure until no more edges can be oriented. On the other hand, if one uses conservative or majority rule FCI and ambiguous triples have been found in pc.cons.intern, unfVect contains the numbers of all ambiguous triples in the graph. In this case, the orientation rules take this information into account. For example, if a \( *\rightarrow b \circ \rightarrow c \) and \( <a,b,c> \) is an unambiguous unshielded triple and not a v-structure, then we obtain \( b \rightarrow c \) (otherwise we would create an additional v-structure). On the other hand, if a \( *\rightarrow b \circ \rightarrow c \) but \( <a,b,c> \) is an ambiguous unshielded triple, then the circle mark at b is not oriented.

Note that the algorithm works with columns’ position of the adjacency matrix and not with the names of the variables.

Note that this function does not resolve possible order-dependence in the application of the orientation rules, see Colombo and Maathuis (2014).

Value

Adjacency matrix of type amat.pag.

Author(s)

Diego Colombo and Markus Kalisch (<kalisch@stat.math.ethz.ch>)

References


See Also

fci.udag2pag, dag2pag; further, udag2pdag (incl. udag2pdagSpecial and udag2pdagRelaxed).
### Example with hidden variables

Zhang (2008), Fig. 6, p.1882

```r
amat <- t(matrix(c(0,1,0,1, 0,0,1,0, 0,0,0,0, 0,0,0,0, 0,0,0,1,0),5,5))
V <- as.character(1:5)
colnames(amat) <- rownames(amat) <- V
edL <- vector("list",length=5)
names(edL) <- V
edL[[1]] <- list(edges= c(2,4),weights=c(1,1))
edL[[2]] <- list(edges= 3, weights=c(1))
edL[[3]] <- list(edges= 5, weights=c(1))
edL[[4]] <- list(edges= 5, weights=c(1))
g <- new("graphNEL", nodes=V, edgeL=edL, edgemode="directed")

if(require("Rgraphviz")) plot(g) else print(g)

## define the latent variable
L <- 1

cov.mat <- trueCov(g)

## compute the true covariance matrix of g
cov.mat <- true Cov(g)

corr <- cov2cor(true.cov)

if (require("MASS")) {
  n <- 100000
  alpha <- 0.01
  set.seed(314)
  d.mat <- mvrnorm(n, mu = rep(0,dim(true.corr)[1]), Sigma = true.cov)
  
  ## estimate the skeleton of given data
  suffStat <- list(C = cor(d.mat), n = n)
  indepTest <- gaussC1test
  resD <- skeleton(suffStat, indepTest, p=dim(true.corr)[2], alpha = alpha)

  ## estimate v-structures conservatively
  tmp <- pc.cons.intern(resD, suffStat, indepTest, alpha, version.unf = c(1, 1))
  tripleList <- tmp$unfTripl
  resD <- tmp$sk

  ## estimate the final skeleton of given data using Possible-D-Sep
  pdsepRes <- pdsep(resD@graph, suffStat, indepTest, p=dim(true.corr)[2],
```
```r
resD@sepset, alpha = alpha, m.max = Inf,
pMax = resD@pMax)

## extend the skeleton into a PAG using all 10 rules
resP <- udag2pag(pag = pdsepRes$G, pdsepRes$sepset, rules = rep(TRUE, 10),
          verbose = TRUE)
colnames(resP) <- rownames(resP) <- as.character(2:5)
print(resP)
```

}\ # only if "MASS" is there

---

udag2pdag

Last PC Algorithm Step: Extend Object with Skeleton to Completed PDAG

Description

These functions perform the last step in the PC algorithm: Transform an object of the class "pcAlgo" containing a skeleton and corresponding conditional independence information into a completed partially directed acyclic graph (CPDAG). The functions first determine the v-structures, and then apply the three orientation rules as described in Sprirtes et al (2000) and Meek (1995) to orient as many of the remaining edges as possible.

In the oracle version and when all assumptions hold, all three functions yield the same CPDAG. In the sample version, however, the resulting CPDAG may be invalid in the sense that one cannot extend it a DAG without additional unshielded colliders by orienting the undirecting edges. This can for example happen due to errors in the conditional independence tests or violations of the faithfulness assumption. The three functions deal with such conflicts in different ways, as described in Details.

Usage

```r
udag2pdag (gInput, verbose)
udag2pdagRelaxed(gInput, verbose, unfVect=NULL, solve.confl=FALSE,
    orientCollider = TRUE, rules = rep(TRUE, 3))
udag2pdagSpecial(gInput, verbose, n.max=100)
```

Arguments

- **gInput** "pcAlgo"-object containing skeleton and conditional independence information.
- **verbose** 0: No output; 1: Details
- **unfVect** vector containing numbers that encode ambiguous triples (as returned by `pc.cons.intern`). This is needed in the conservative and majority rule PC algorithms.
- **solve.confl** if TRUE, the orientation of the v-structures and the orientation rules work with lists for candidate sets and allow bi-directed edges to resolve conflicting edge orientations. Note that therefore the resulting object is order-independent but might not be a PDAG because bi-directed edges can be present.
udag2pdag

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n.max</td>
<td>maximum number of tries for re-orienting doubly visited edges in udag2pdagSpecial.</td>
</tr>
<tr>
<td>orientCollider</td>
<td>if TRUE, collider are oriented.</td>
</tr>
<tr>
<td>rules</td>
<td>Array of length 3 containing TRUE or FALSE for each rule. TRUE in position i means that rule i (Ri) will be applied. By default, all rules are used.</td>
</tr>
</tbody>
</table>

Details

for **udag2pdag**: If there are edges that are part of more than one v-structure (i.e., the edge b - c in the v-structures a -> b <- c and b -> c <- d), earlier edge orientations are simply overwritten by later ones. Thus, if a -> b <- c is considered first, the edge b - c is first oriented as b <- c and later overwritten by b -> c. The v-structures are considered in lexicographical ordering. If the resulting graph is extendable to a DAG without additional v-structures, then the rules of Meek (1995) and Spirtes et al (2000) are applied to obtain the corresponding CPDAG. Otherwise, the edges are oriented randomly to obtain a DAG that fits on the skeleton, discarding all information about the v-structures. The resulting DAG is then transformed into its CPDAG. Note that the output of udag2pdag is random whenever the initial graph was not extendable.

Although the output of udag2pdag is always extendable, it is not necessarily a valid CPDAG in the sense that it describes a Markov equivalence class of DAGs. For example, two v-structures a -> b <- c and b -> c <- d (considered in this order) would yield the output a -> b -> c <- d. This is extendable to a DAG (it already is a DAG), but it does not describe a Markov equivalence class of DAGs, since the DAG a <- b -> c <- d describes the same conditional independencies.

for **udag2pdagSpecial**: If the graph after orienting the v-structures as in udag2pdag is extendable to a DAG without additional v-structures, then the rules of Meek (1995) and Spirtes et al (2000) are applied to obtain the corresponding CPDAG. Otherwise, the algorithm tries at most n.max different random orderings of the v-structures (hence overwriting orientations in different orders), until it finds one that yields an extendable CPDAG. If this fails, the edges are oriented randomly to obtain a DAG that fits on the skeleton, discarding all information about the v-structures. The resulting DAG is then transformed into its CPDAG. Note that the output of udag2pdagSpecial is random whenever the initial graph was not extendable.

Although the output of udag2pdag is always extendable, it is not necessarily a valid CPDAG in the sense that it describes a Markov equivalence class of DAGs. For example, two v-structures a -> b <- c and b -> c <- d (considered in this order) would yield the output a -> b -> c <- d. This is extendable to a DAG (it already IS a DAG), but it does not describe a Markov equivalence class of DAGs, since the DAG a <- b -> c <- d describes the same conditional independencies.

for **udag2pdagRelaxed**: This is the default version in the PC/RFCl/FCI algorithm. It does not test whether the output is extendable to a DAG without additional v-structures.

If unfVect = NULL (no ambiguous triples), the three orientation rules are applied to each eligible structure until no more edges can be oriented. Otherwise, unfVect contains the numbers of all ambiguous triples in the graph as determined by pc.cons.intern. Then the orientation rules take this information into account. For example, if a -> b - c and <a,b,c> is an unambiguous triple and a non-v-structure, then rule 1 implies b -> c. On the other hand, if a -> b - c but <a,b,c> is an ambiguous triple, then the edge b - c is not oriented.

If solve.confl = FALSE, earlier edge orientations are overwritten by later ones as in udag2pdag and udag2pdagSpecial.

If solv.confl = TRUE, both the v-structures and the orientation rules work with lists for the candidate edges and allow bi-directed edges if there are conflicting orientations. For example,
two v-structures \( a \to b \leftarrow c \) and \( b \to c \leftarrow d \) then yield \( a \to b \leftarrow c \leftarrow d \). This option can be used to get an order-independent version of the PC algorithm (see Colombo and Maathuis (2014)). We denote bi-directed edges, for example between two variables \( i \) and \( j \), in the adjacency matrix \( M \) of the graph as \( M[i,j]=2 \) and \( M[j,i]=2 \). Such edges should be interpreted as indications of conflicts in the algorithm, for example due to errors in the conditional independence tests or violations of the faithfulness assumption.

**Value**

for `udag2pdag()` and `udag2pdagRelaxed()`: oriented "pcAlgo"-object.

for `udag2pdagSpecial`: a `list` with components

- `pcObj` An oriented "pcAlgo"-object.
- `evisit` Matrix counting the number of orientation attempts per edge
- `xtbl.orig` Logical indicating whether the original graph with v-structures is extendable.
- `xtbl` Logical indicating whether the final graph with v-structures is extendable
- `amat0` Adjacency matrix of original graph with v-structures (type `amat.cpdag`).
- `amat1` Adjacency matrix of final graph with v-structures after changing the ordering in which the v-structures are considered (type `amat.cpdag`).
- `status` Integer code with values
  - 0: Original try is extendable;
  - 1: Reorienting double edge visits helps;
  - 2: Original try is not extendable; reorienting double visits does not help; result is acyclic, has original v-structures, but perhaps additional v-structures.
- `counter` Number of orderings of the v-structures until success or \( n_{max} \).

**Author(s)**

Markus Kalisch (<kalisch@stat.math.ethz.ch>)

**References**


**See Also**

`pc`, `pdag2dag`, `dag2cpdag`, `udag2pag`, `udag2apag`, `dag2pag`.
Examples

```r
## simulate data
set.seed(123)
p <- 10
myDAG <- randomDAG(p, prob = 0.2)
trueCPDAG <- dag2cpdag(myDAG)
n <- 1000
d.mat <- rmvDAG(n, myDAG, errDist = "normal")

## estimate skeleton
resU <- skeleton(suffStat = list(C = cor(d.mat), n = n),
                 indepTest = gaussCItest, ## (partial correlations)
                 alpha = 0.05, p=p)

## orient edges using three different methods
resD1 <- udag2pdagRelaxed(resU, verbose=0)
resD2 <- udag2pdagSpecial(resU, verbose=0, n.max=100)
resD3 <- udag2pdag(resU, verbose=0)
```

visibleEdge

Check visible edge.

Description

Check if the directed edge from x to z in a MAG or in a PAG is visible or not.

Usage

`visibleEdge(amat, x, z)`

Arguments

- `amat`: Adjacency matrix of type `amat.pag`
- `x, z`: (integer) position of variable `x` and `z`, respectively, in the adjacency matrix.

Details

All directed edges in DAGs and CPDAGs are said to be visible. Given a MAG M / PAG P, a directed edge A -> B in M / P is visible if there is a vertex C not adjacent to B, such that there is an edge between C and A that is into A, or there is a collider path between C and A that is into A and every non-endpoint vertex on the path is a parent of B. Otherwise A -> B is said to be invisible. (see Maathuis and Colombo (2015), Def. 3.1)

Value

TRUE if edge is visible, otherwise FALSE.
### Description

Given a `graph` object `g`, as generated e.g., by `randomDAG`, return the matrix of its edge weights, the “weight matrix”.

### Usage

```r
wgtMatrix(g, transpose = TRUE)
```

### Arguments

- `g`  
  graph object (package `graph`) of, say, `p` nodes, e.g. containing a DAG.
- `transpose`  
  logical indicating if the weight matrix should be transposed (`.t()`, see details).
Details

When generating a DAG (e.g. using randomDAG), a graph object is usually generated and edge weights are usually specified. This function extracts the edge weights and arranges them in a matrix $M$.

If transpose is TRUE (default), $M_{i,j}$ is the weight of the edge from $j$ to $i$. If transpose is false, $M_{i,j}$ is the weight of the edge from $i$ to $j$.

Nowadays, this is a trivial wrapper around as(g,"matrix") using the (coerce) method provided by the graph package.

Value

The $p \times p$ weight matrix $M$.

Note

This function can not be used to estimate the edge weights in an estimated DAG / CPDAG.

Author(s)

Markus Kalisch

See Also

randomDAG for generating a random DAG; rmvDAG for simulating data from a generated DAG.

Examples

```r
set.seed(123)
g <- randomDAG(n = 5, prob = 0.3)  ## generate random DAG
if(require(Rgraphviz)) {
  plot(g)
}

## edge weights as matrix
wgtMatrix(g)

## for comparison: edge weights in graph object
g@edgeData@data
```
Index

* **CPDAG**
  dag2essgraph, 30
  ges, 64
  opt.target, 99

* **DAG**
  dag2essgraph, 30
  randDAG, 136

* **Gaussian causal model**
  GaussParDAG-class, 60
  r.gauss.pardag, 134
  rmvnorm.ivent, 145

* **arith**
  getNextSet, 69

* **classes**
  EssGraph-class, 39
  fciAlgo-class, 47
  gAlgo-class, 55
  GaussL0penIntScore-class, 56
  GaussL0penObsScore-class, 58
  GaussParDAG-class, 60
  ParDAG-class, 104
  pcAlgo-class, 116
  Score-class, 146

* **datagen**
  r.gauss.pardag, 134
  randomDAG, 138
  rmvDAG, 144
  rmvnorm.ivent, 145

* **datasets**
  gmB, 73
  gmD, 74
  gmG, 75
  gmI, 76
  gmInt, 77
  gml, 79

* **dynamic programming**
  simy, 151

* **essential graph**
  dag2essgraph, 30
  gds, 62
  ges, 64
  gies, 70
  opt.target, 99

* **graphs**
  adjustment, 5
  ages, 7
  amatType, 11
  backdoor, 14
  beta.special, 18
  beta.special.pcObj, 19
  compareGraphs, 24
  corGraph, 28
  dag2cpdag, 29
  dag2essgraph, 30
  dag2pag, 32
  fci, 41
  fciPlus, 48
  gac, 51
  gds, 62
  ges, 64
  ida, 80
  idaFast, 85
  iplotPC, 86
  isValidGraph, 88
  jointIda, 89
  LINGAM, 94
  optAdjSet, 100
  pag2mag, 102
  pc, 105
  pc.cons.intern, 111
  pcAlgo, 114
  pcSelect, 119
  pcSelect.presel, 121
  pdag2dag, 124
  plotAG, 128
  plotSG, 128
  possAn, 130
  possDe, 131
INDEX

r.gauss.pardag, 134
randDAG, 136
randomDAG, 138
crfci, 140
shd, 148
crshowEdgeList, 150
crsum, 151
crskelton, 153
crudag2apag, 158
crudag2pag, 161
crudag2pdag, 164

* graph
  getGraph, 68
gies, 70
* greedy DAG search
  gds, 62
* greedy equivalence search
  ges, 64
* greedy interventional equivalence search
  gies, 70
* hplot
  showAmat, 149
* htest
  condIndFisherZ, 26
* interventional Markov equivalence
  simy, 151
* interventional data
  mat2targets, 96
* interventions
  opt.target, 99
* intervention
  mat2targets, 96
* list
  mat2targets, 96
* manip
  checkTriple, 22
  mat2targets, 96
* methods
  getGraph, 68
* misc
  find.unsh.triple, 50
  legal.path, 93
  possibleDe, 132
  trueCov, 157
  visibleEdge, 167
  wgtMatrix, 168
* models
  addBgKnowledge, 4
adjustment, 5
ages, 7
backdoor, 14
beta.special, 18
beta.special.pcObj, 19
corGraph, 28
crudag2cpdag, 29
crudag2essgraph, 30
crudag2pag, 32
fci, 41
fciPlus, 48
gac, 51
GaussParDAG-class, 60
gds, 62
ges, 64
gies, 70
ida, 80
idaFast, 85
jointIda, 89
LINGAM, 94
pag2mag, 102
pc, 105
pc.cons.intern, 111
pcAlgo, 114
pcSelect, 119
pcSelect.presel, 121
plotAG, 128
crudag2pag, 136
rfci, 140
rmvnorm.ivent, 145
crudskelton, 153
crududag2apag, 158
crududag2pag, 161
crududag2pdag, 164
* multivariate
  backdoor, 14
  beta.special, 18
  condIndFisherZ, 26
cruddag2cpdag, 29
cruddag2pag, 32
fci, 41
fciPlus, 48
gac, 51
ida, 80
idaFast, 85
jointIda, 89
LINGAM, 94
mcor, 98
pag2mag, 102
pc, 105
pc.cons.intern, 111
pcAlgo, 114
pcorOrder, 118
pcSelect, 119
pcSelect.presel, 121
plotAG, 128
rfci, 140
rmvDAG, 144
skeleton, 153
udag2apag, 158
udag2pag, 161
udag2pdag, 164
* random directed acyclic graph
  randDAG, 136
* random graph
  r.gauss.pardag, 134
* robust
  mcor, 98
* structural equation model
  GaussParDAG-class, 60
* utilities
  getNextSet, 69

addBgKnowledge, 4, 83, 90
adjustment, 5
ages, 7
agopen, 116
amat.cpdag, 14, 50, 51, 122, 129, 166
amat.cpdag (amatType), 11
amat.pag, 14, 36, 51, 103, 132, 133, 158, 159,
  161, 162, 167
amat.pag (amatType), 11
amatType, 4, 11, 15, 47, 52, 88, 113, 130–133
as, 11

backdoor, 14, 37, 53, 103, 133, 168
barabasi.game, 138
beta.special, 18, 20
beta.special.pcoObj, 19, 19
biConnComp, 42, 126
binCItest, 20, 27, 35, 39, 45, 109, 142, 154,
  155

call, 55
causalEffect (ida), 80
character, 28, 41, 49, 139

ccheckTriple, 22
class, 11, 33, 37, 38, 44, 49, 75, 109, 115,
  142, 155
coeerce, 68, 169
coeerce, fciAlgo, amat-method (amatType), 11
coeerce, fciAlgo, matrix-method
  (amatType), 11
coeerce, LINGAM, amat-method (amatType), 11
coeerce, pcAlgo, amat-method (amatType), 11
coeerce, pcAlgo, matrix-method (amatType), 11
combn, 69
compareGraphs, 24, 139
condIndFisherZ, 26, 118
corGraph, 28
covOGK, 98
dag2cpdag, 19, 20, 29, 31, 166
dag2essgraph, 29, 30, 30
dag2pag, 11, 16, 32, 47, 103, 160, 162, 166
data.frame, 74
discItest, 21, 27, 34, 39, 45, 109, 142, 154,
  155
dreach, 36
dsep, 37, 38
dsepTest, 21, 27, 35, 37, 38, 45, 109, 142, 155
eenvRefClass, 40, 56, 58, 60
EssGraph, 9, 31, 63, 64, 67, 72, 73, 99, 100,
  152
EssGraph-class, 39
factor, 74
fastICA, 95
fci, 11, 16, 20, 21, 26, 33, 34, 37, 38, 41,
  47–49, 53, 68, 69, 103, 111–114,
  126–128, 134, 141, 142, 155, 162
fciAlgo, 11, 12, 33, 44, 49, 55, 87, 117, 128,
  142, 150
fciAlgo-class, 46
fciPlus, 11, 45, 47, 48, 53, 142
find.unsh.triple, 50
function, 22, 32, 41, 49, 106, 137, 153
gac, 6, 11, 12, 16, 51
gAlgo, 47, 48, 117
gAlgo-class, 55
gaussCItest, 21, 35, 39, 45, 109, 142, 154,
  155
INDEX

gaussCItest (condIndFisherZ), 26
GaussL0penIntScore, 59, 147, 148
GaussL0penIntScore-class, 56
GaussL0penObsScore, 57, 147, 148
GaussL0penObsScore-class, 58
GaussParDAG, 105, 135, 146
GaussParDAG-class, 60
gds, 62
ges, 8, 9, 58, 59, 62–64, 64, 73, 146–148
getGraph, 12, 28, 68
getGraph, ANY-method (getGraph), 68
getGraph, fciAlgo-method (getGraph), 68
getGraph, matrix-method (getGraph), 68
getGraph, pcAlgo-method (getGraph), 68
getGraph-methods (getGraph), 68
getNextSet, 69
gies, 56, 57, 62–64, 70, 97, 146–148, 152
global.mle, GaussL0penIntScore-method (GaussL0penIntScore-class), 56
global.mle, GaussL0penObsScore-method (GaussL0penObsScore-class), 58
global.score, GaussL0penIntScore-method (GaussL0penIntScore-class), 56
global.score, GaussL0penObsScore-method (GaussL0penObsScore-class), 58
gmB, 73
gmD, 74
gmG, 75, 77
gmG8 (gmG), 75
gmI, 76
gmI7 (gmI), 76
 gmInt, 77
 gml, 79
graph, 12, 28, 68, 69, 117, 129, 168
graphAM, 68
graphNEL, 31, 99, 137
gSquareBin, 35
gSquareBin (binCItest), 20
gSquareDis, 20, 21
gSquareDis (disCItest), 34
ida, 18, 19, 80, 85, 86, 89, 91
idaFast, 19, 83, 85, 91
identical, 76–78
igraph, 138
integer, 47, 55
invisible, 129
iplotPC, 86, 150
isValidGraph, 4, 88, 109
johnson.all.pairs.sp, 37, 38
jointIda, 82, 83, 89
legal.path, 93
LINGAM, 94
lingam, 11
lingam (LINGAM), 94
list, 20, 22, 26, 32, 35, 38, 41, 47, 49, 52, 55, 62, 74, 75, 77, 94, 106, 137, 158, 166
local.mle, GaussL0penIntScore-method (GaussL0penIntScore-class), 56
local.mle, GaussL0penObsScore-method (GaussL0penObsScore-class), 58
local.score, GaussL0penIntScore-method (GaussL0penIntScore-class), 56
local.score, GaussL0penObsScore-method (GaussL0penObsScore-class), 58
logical, 27, 42, 47, 96, 116, 119, 120, 129, 134
mat2targets, 96
matrix, 55
mcor, 98, 119
opt.target, 99
optAdjSet, 100
pag2mag, 102
pag2magAM, 14, 16, 37
pag2magAM (pag2mag), 102
ParDAG, 9, 31, 41, 57, 59–61, 64, 67, 72, 152
ParDAG-class, 104
pc, 11, 16, 20–22, 26, 30, 33, 34, 37, 38, 42,
  45, 53, 66–68, 80, 81, 83, 85–87, 90,
  91, 105, 111–114, 116, 117, 119, 
  120, 126, 142, 145, 149, 150, 155, 
  166
pc.cons.intern, 111, 126, 159, 162, 164, 165
pcalg2dagitty, 113
pcAlgo, 11, 19, 20, 48, 55, 86, 87, 109, 114,
  115, 149, 150, 153, 164
pcAlgo-class, 116
pccorOrder, 27, 98, 118
pcSelect, 119, 121
pcSelect.presel, 120, 121
pdag2allDags, 81, 122
pdag2dag, 124, 166
pdsep, 43, 45, 125, 134, 161
plot, 47
plot,EssGraph,ANY-method
   (EssGraph-class), 39
plot,fciAlgo,ANY-method
   (fciAlgo-class), 47
plot,ParDAG,ANY-method (ParDAG-class), 104
plot,pcAlgo,ANY-method
   (pcAlgo-class), 116
plotAG, 128
plotSG, 128
possAn, 130
possDe, 131, 132
possibleDe, 132
print.fciAlgo (fciAlgo-class), 47
print.pcAlgo (pcAlgo-class), 116
pseudoinverse, 118
Qn, 98
qnorm, 26
qreach, 43, 127, 133
r.gauss.pardag, 134
randDAG, 136, 138, 139
random.graph.game, 138
randomDAG, 25, 30, 32, 77, 135, 137, 138, 145, 158, 168, 169
rfci, 11, 47, 140, 155, 160
rmvDAG, 138, 139, 144, 169
rmvnorm.ivent, 145
runif, 139
Score, 31, 40, 56–59, 62, 64, 65, 67, 70, 73, 97, 105, 151, 152
Score-class, 146
shd, 148
show, 47
show,fciAlgo-method (fciAlgo-class), 47
show,pcAlgo-method (pcAlgo-class), 116
show.fci.amat (amatType), 11
show.pc.amat (amatType), 11
showAmat, 87, 149, 150
showEdgeList, 87, 150, 150
simy, 56, 57, 97, 146–148, 151
skeleton, 11, 20, 21, 26, 34, 37, 38, 42, 43, 45, 55, 68, 70, 87, 106, 107, 109, 112–114, 116, 117, 125, 126, 140–142, 145, 149, 150, 153
summary, 47
summary,fciAlgo-method (fciAlgo-class), 47
summary,pcAlgo-method (pcAlgo-class), 116
t, 168
targets2mat (mat2targets), 96
title, 129
triple2numb (pc.cons.intern), 111
trueCov, 157
tsort, 144
udag2apag, 158, 162, 166
udag2pag, 159, 160, 161, 166
udag2pdag, 19, 109, 155, 160, 162, 164, 165
udag2pdagRelaxed, 19, 160, 165
udag2pdagRelaxed (udag2pdag), 164
udag2pdagSpecial, 19, 160, 165
udag2pdagSpecial (udag2pdag), 164
unifDAG, 138
visibleEdge, 167
wgtMatrix, 76, 168
zStat (condIndFisherZ), 26