Package ‘cna’

May 13, 2020

Type Package

Title Causal Modeling with Coincidence Analysis

Version 2.2.3

Date 2020-05-13

Description Provides comprehensive functionalities for causal modeling with Coincidence Analysis (CNA), which is a configurational comparative method of causal data analysis that was first introduced in Baumgartner (2009) <doi:10.1177/0049124109339369>, and generalized in Baumgartner & Ambuehl (2018) <doi:10.1017/psrm.2018.45>. CNA is related to Qualitative Comparative Analysis (QCA), but contrary to the latter, it is custom-built for uncovering causal structures with multiple outcomes and it builds causal models from the bottom up by gradually combining single factors to complex dependency structures until the requested thresholds of model fit are met. The new functionalities provided by this package version include functions for evaluating and benchmarking the correctness of CNA's output, a function determining whether a solution is an INUS model, a function bringing non-INUS expressions into INUS form, and a function for identifying cyclic models. The package vignette has been updated accordingly.

License GPL (>= 2)

URL https://CRAN.R-project.org/package=cna

Depends R (>= 3.2.0)

Imports Rcpp, utils, stats, matrixStats

LinkingTo Rcpp

Suggests dplyr

NeedsCompilation yes

LazyData yes

Maintainer Mathias Ambuehl <mathias.ambuehl@consultag.ch>

Author Mathias Ambuehl [aut, cre, cph],
Michael Baumgartner [aut, cph],
Ruedi Epple [ctb],
Veli-Pekka Parkkinen [ctb],
Alrik Thiem [ctb]

Repository CRAN

Date/Publication 2020-05-13 18:20:08 UTC

R topics documented:

cna-package 2
allCombs 4
cna 6
coherence 18
condition 19
condTbl 24
cyclic 28
d.autonomy 30
d.educate 32
d.irrigate 33
d.jobsecurity 33
d.minaret 34
d.pacts 35
d.pban 35
d.performance 36
d.volatile 37
d.women 37
full.tt 38
is.inus 40
is.submodel 42
makeFuzzy 45
minimalize 46
minimalizeCsf 48
randomConds 50
redundant 54
selectCases 56
some 58
truthTab 60
tt2df 64

Index **66**

---

cna-package *cna: A Package for Causal Modeling with Coincidence Analysis*

Description

Coincidence Analysis (CNA) is a configurational comparative method of causal data analysis that was first introduced for crisp-set (i.e. binary) data in Baumgartner (2009a, 2009b, 2013) and generalized for multi-value and fuzzy-set data in Baumgartner and Ambuehl (2018). The **cna** package reflects and implements the method's latest stage of development.

CNA is related to Qualitative Comparative Analysis (QCA) (Ragin 1987, 2008). Like QCA, CNA processes configurational data, i.e. data consisting of observed cases featuring different factor configurations, it searches for redundancy-free sufficient and necessary conditions of causally modeled outcomes, it places a Boolean ordering on causally relevant factor values (instead of e.g. quantifying net effects and effect sizes in the vein of regression analysis), and it draws on the same regularity theoretic notion of causation as QCA, i.e. the notion first introduced by Mackie (1974). Contrary to QCA, however, CNA is custom-built to treat multiple factors as outcomes, and it does not generate causal models from the top down by first building maximal Boolean dependency structures and then gradually eliminating redundant elements (using e.g. Quine-McCluskey optimization; cf. McCluskey 1965); rather, CNA builds causal models from the bottom up by gradually combining single factor values to complex dependency structures until the requested thresholds of model fit are met, such that the resulting models are automatically redundancy-free. As a consequence of these differences, CNA can identify common-cause and causal-chain structures and it can avoid the task of redundancy elimination (which creates various problems for QCA). Moreover, the algorithm does not require an input identifying the endogenous factors; it can infer that from the data. Finally, data fragmentation (limited diversity) does not force CNA to resort to counterfactual reasoning.

The new functionalities provided by version 2.2 of the **cna** package include functions for evaluating and benchmarking CNA's output. The functions randomAsf() and randomCsf() draw data-generating structures with one and multiple outcomes, respectively, as a basis for inverse searches. By means of is.submodel(), the CNA output can be scanned for correctness-preserving models. Moreover, is.inus() determines whether a solution is an INUS model, minimalize() brings non-INUS models into INUS form, and cyclic() checks whether causal models contain cyclic substructures, i.e. feedback loops. The package vignette, which presents the theoretical background of CNA and introduces to causal modeling with **cna**, has been updated accordingly. In particular, a section on correctness benchmarking has been added.

Details

Package:	cna
Type:	Package
Version:	2.2.3
Date:	2020-05-13
License:	GPL (>= 2)

Author(s)

Authors:
Mathias Ambuehl
`<mathias.ambuehl@consultag.ch>`

Michael Baumgartner
Department of Philosophy
University of Bergen
`<michael.baumgartner@uib.no>`

Maintainer:
Mathias Ambuehl

References

Baumgartner, Michael. 2009a. "Inferring Causal Complexity." *Sociological Methods & Research* 38(1):71-101.

Baumgartner, Michael. 2009b. “Uncovering Deterministic Causal Structures: A Boolean Approach.” *Synthese* 170(1):71-96.

Baumgartner, Michael. 2013. “Detecting Causal Chains in Small-n Data.” *Field Methods* 25 (1):3-24.

Baumgartner, Michael and Mathias Ambuehl. 2018. “Causal Modeling with Multi-Value and Fuzzy-Set Coincidence Analysis.” *Political Science Research and Methods* doi:10.1017/psrm.2018.45.

Baumgartner, Michael and Alrik Thiem. 2015. “Identifying Complex Causal Dependencies in Configurational Data with Coincidence Analysis”, *The R Journal* 7:176-184.

Mackie, John L. 1974. *The Cement of the Universe: A Study of Causation*. Oxford: Oxford University Press.

McCluskey, Edward J. 1965. *Introduction to the Theory of Switching Circuits*. Princeton: Princeton University Press.

Ragin, Charles C. 1987. *The Comparative Method*. Berkeley: University of California Press.

Ragin, Charles C. 2008. *Redesigning Social Inquiry: Fuzzy Sets and Beyond*. Chicago: University of Chicago Press.

allCombs *Generate all logically possible value configurations of a given set of factors*

Description

The function `allCombs` generates a data frame of all possible value configurations of `length(x)` factors, the first factor having `x[1]` values, the second `x[2]` values etc. The factors are labeled using capital letters.

Usage

```
allCombs(x)
```

Arguments

`x`	Integer vector with values >0

Details

In combination with `selectCases` and `makeFuzzy`, `allCombs` is useful for simulating data, which are needed for inverse search trials benchmarking the output of `cna`. In a nutshell, `allCombs` generates the space of all logically possible configurations of the factors in an analyzed factor set, `selectCases` selects those configurations from this space that are compatible with a given data-generating causal structure (i.e. the ground truth, which can be randomly generated using `randomConds`), `makeFuzzy` introduces noise into that data, and `is.submodel` checks whether the models returned by `cna` are true of the ground truth.

The **cna** package provides another function to the same effect, `full.tt`, which is more flexible than `allCombs`.

Value

A data frame.

See Also

selectCases, makeFuzzy, is.submodel, randomConds, full.tt

Examples

```
# Generate all logically possible configurations of 5 dichotomous factors named "A", "B",
# "C", "D", and "E".
allCombs(c(2, 2, 2, 2, 2)) - 1
# allCombs(c(2, 2, 2, 2, 2)) generates the value space for values 1 and 2, but as it is
# conventional to use values 0 and 1 for Boolean factors, 1 must be subtracted from
# every value output by allCombs(c(2, 2, 2, 2, 2)) to yield a Boolean data frame.

# Generate all logically possible configurations of 5 multi-value factors named "A", "B",
# "C", "D", and "E", such that A can take on 3 values {1,2,3}, B 4 values {1,2,3,4},
# C 3 values etc.
dat0 <- allCombs(c(3, 4, 3, 5, 3))
head(dat0)
nrow(dat0) # = 3*4*3*5*3

# Generate all configurations of 5 dichotomous factors that are compatible with the causal
# chain (A*b + a*B <-> C)*(C*d + c*D <-> E).
dat1 <- allCombs(c(2, 2, 2, 2, 2)) - 1
(dat2 <- selectCases("(A*b + a*B <-> C)*(C*d + c*D <-> E)", dat1))

# Generate all configurations of 5 multi-value factors that are compatible with the causal
# chain (A=2*B=1 + A=3*B=3 <-> C=1)*(C=1*D=2 + C=4*D=4 <-> E=3).
dat1 <- allCombs(c(3, 3, 4, 4, 3))
dat2 <- selectCases("(A=2*B=1 + A=3*B=3 <-> C=1)*(C=1*D=2 + C=4*D=4 <-> E=3)", dat1,
                    type = "mv")
nrow(dat1)
nrow(dat2)

# Generate all configurations of 5 fuzzy-set factors that are compatible with the causal
# structure A*b + C*D <-> E, such that con = .8 and cov = .8.
dat1 <- allCombs(c(2, 2, 2, 2, 2)) - 1
dat2 <- makeFuzzy(dat1, fuzzvalues = seq(0, 0.45, 0.01))
(dat3 <- selectCases1("A*b + C*D <-> E", con = .8, cov = .8, dat2))

# Inverse search for the data generating causal structure A*b + a*B + C*D <-> E from
# fuzzy-set data with non-perfect consistency and coverage scores.
set.seed(3)
groundTruth <- "A*b + a*B + C*D <-> E"
dat1 <- allCombs(c(2, 2, 2, 2, 2)) - 1
dat2 <- makeFuzzy(dat1, fuzzvalues = 0:4/10)
dat3 <- selectCases1(groundTruth, con = .8, cov = .8, dat2)
ana1 <- fscna(dat3, ordering = list("E"), strict = TRUE, con = .8, cov = .8)
any(is.submodel(asf(ana1)$condition, groundTruth))
```

cna *Perform Coincidence Analysis*

Description

The cna function performs Coincidence Analysis to identify atomic solution formulas (asf) consisting of minimally necessary disjunctions of minimally sufficient conditions of all outcomes in the data and combines the recovered asf to complex solution formulas (csf) representing multi-outcome structures, e.g. common-cause and/or causal-chain structures.

Usage

```
cna(x, type, ordering = NULL, strict = FALSE, con = 1, cov = 1, con.msc = con,
    notcols = NULL, rm.const.factors = TRUE, rm.dup.factors = TRUE,
    maxstep = c(3, 3, 9), inus.only = FALSE, only.minimal.msc = TRUE,
    only.minimal.asf = TRUE, maxSol = 1e6, suff.only = FALSE,
    what = if (suff.only) "m" else "ac", cutoff = 0.5,
    border = c("down", "up", "drop"), details = FALSE)
cscna(...)
mvcna(...)
fscna(...)

## S3 method for class 'cna'
print(x, what = x$what, digits = 3, nsolutions = 5,
      details = x$details, show.cases = NULL, ...)
```

Arguments

`x`	Data frame or `truthTab` (as output by `truthTab`).
`type`	Character vector specifying the type of `x`: `"cs"` (crisp-set), `"mv"` (multi-value), or `"fs"` (fuzzy-set).
`ordering`	List of character vectors specifying the causal ordering of the factors in `x`.
`strict`	Logical; if `TRUE`, factors on the same level of the causal ordering are *not* potential causes of each other; if `FALSE`, factors on the same level *are* potential causes of each other.
`con`	Numeric scalar between 0 and 1 to set the minimum consistency threshold every minimally sufficient condition (msc), atomic solution formula (asf), and complex solution formula (csf) must satisfy. (See also the argument `con.msc` below).
`cov`	Numeric scalar between 0 and 1 to set the minimum coverage threshold every asf and csf must satisfy.
`con.msc`	Numeric scalar between 0 and 1 to set the minimum consistency threshold every msc must satisfy. Allows for imposing a consistency threshold on msc that differs from the value `con` imposes on asf and csf. Defaults to `con`.

`maxstep`	Vector of three integers; the first specifies the maximum number of conjuncts in each disjunct of an asf, the second specifies the maximum number of disjuncts in an asf, the third specifies the maximum *complexity* of an asf. The complexity of an asf is the total number of exogenous factors in the asf.
`inus.only`	Logical; if `TRUE`, only disjunctive normal forms that are free of logical redundancies are retained as asf (see also `is.inus`). Defaults to `FALSE`.
`only.minimal.msc`	Logical; if `TRUE` (the default), only minimal conjunctions are retained as msc. If `FALSE`, sufficient conjunctions are not required to be minimal, in which case the number of msc will usually be much greater.
`only.minimal.asf`	Logical; if `TRUE` (the default), only minimal disjunctions are retained as asf. If `FALSE`, necessary disjunctions are not required to be minimal, in which case the number of asf will usually be much greater.
`maxSol`	Maximum number of asf calculated.
`suff.only`	Logical; if `TRUE`, the function only searches for msc and does not search for asf and csf.
`notcols`	Character vector of factors to be negated in `x`. If `notcols = "all"`, all factors in `x` are negated.
`rm.const.factors, rm.dup.factors`	Logical; if `TRUE` (default), factors with constant values are removed and all but the first of a set of duplicated factors are removed. These parameters are passed to `truthTab`.
`what`	Character string specifying what to print; `"t"` for the truth table, `"m"` for msc, `"a"` for asf, `"c"` for csf, and `"all"` for all. Defaults to `"ac"` if `suff.only = F`, and to `"m"` otherwise.
`cutoff`	Minimum membership score required for a factor to count as instantiated in the data and to be integrated in the analysis. Value in the unit interval (0,1). The default cutoff is 0.5. Only meaningful if `type="fs"`.
`border`	Character vector specifying whether factors with membership scores equal to `cutoff` are rounded up (`"up"`), rounded down (`"down"`) or dropped from the analysis (`"drop"`). Only meaningful if `type="fs"`.
`details`	Either `TRUE`/`FALSE`, or a character vector with possible elements `"inus"`, `"exhaustiveness"`, `"faithfulness"`, `"coherence"`, `"redundant"`. The strings can also be abbreviated, e.g. `"i"` for `"inus"`, `"e"` or `"exh"` for `"exhaustiveness"`, etc.
`digits`	Number of digits to print in consistency, coverage, exhaustiveness, faithfulness, and coherence scores.
`nsolutions`	Maximum number of msc, asf, and csf to print. Alternatively, `nsolutions = "all"` will print all solutions.
`show.cases`	Logical; if `TRUE`, the `truthTab`'s attribute "cases" is printed. See `print.truthTab`
`...`	In `cscna`, `mvcna`, `fscna`: any formal argument of `cna` except `type`. In `print.cna`: arguments passed to other `print`-methods.

Details

The first input x of the cna function is a data frame or a truthTab. To ensure that no misinterpretations of returned asf and csf can occur, users are advised to use only upper case letters as factor (column) names. Column names may contain numbers, but the first sign in a column name must be a letter. Only ASCII signs should be used for column and row names.

cna must be told what type of data x contains, unless x is a truthTab. In the latter case, the type of x is already defined. Data that feature factors taking values 1 or 0 only are called *crisp-set*, in which case the type argument takes its default value "cs". If the data contain at least one factor that takes more than two values, e.g. {1,2,3}, the data count as *multi-value*, which is indicated by type = "mv". Data featuring at least one factor taking real values from the interval [0,1] count as *fuzzy-set*, which is specified by type = "fs". (Note that mixing multi-value and fuzzy-set factors in one analysis is not (currently) supported). To abbreviate the specification of the data type using the type argument, the functions cscna(x,...), mvcna(x,...), and fscna(x,...) are available as shorthands for cna(x,type = "cs",...), cna(x,type = "mv",...), and cna(x,type = "fs",...), respectively.

A data frame or truthTab x with a corresponding type specification is the only mandatory input of the cna function. If no causal ordering is provided (see below), all factor values in x are treated as potential outcomes; more specifically, in case of "cs" and "fs" data, cna tests for all factors whether their presence (i.e. them taking the value 1) can be modeled as an outcome, and in case of "mv" data, cna tests for all factors whether any of their possible values can be modeled as an outcome. That is done by, first, identifying all minimally sufficient conditions (msc) that meet the threshold given by con.msc (resp. con, if con.msc = con) for each factor in x. Then, cna disjunctively combines these msc to minimally necessary conditions that meet the threshold given by cov such that the whole disjunction meets the threshold given by con. The resulting expressions are the atomic solution formulas (asf) for every factor value that can be modeled as outcome. The default value for con.msc, con, and cov is 1.

[Consistency and coverage measures have originally been introduced into the QCA protocol by Ragin (2006). Informally put, consistency reproduces the degree to which the behavior of an outcome obeys a corresponding sufficiency or necessity relationship or a whole causal model, whereas coverage reproduces the degree to which a sufficiency or necessity relationship or a whole model accounts for the behavior of the corresponding outcome. For details see the **cna** package vignette or Ragin (2006).]

cna builds msc and asf *from the bottom up*. That is, in a first phase, cna checks whether single factor values A, b, C, (where "A" stands for "A=1" and "b" for "B=0") or D=3, E=2, etc. (whose membership scores, in case of "fs" data, meet cutoff in at least one case) are sufficient for an outcome (where a factor value counts as sufficient iff it meets the threshold given by con.msc). Next, conjuncts of two factor values A*b, A*C, D=3*E=2 etc. (whose membership scores, in case of "fs" data, meet cutoff in at least one case) are tested for sufficiency. Then, conjuncts of three factors, and so on. Whenever a conjunction (or a single factor value) is found to be sufficient, all supersets of that conjunction contain redundancies and are, thus, not considered for the further analysis. The result of that first phase is a set of msc for every outcome. To recover certain target structures in cases of noisy data, it may be useful to allow cna to also consider sufficient conditions for further analysis that are not minimal. This can be accomplished by setting only.minimal.msc to FALSE. A concrete example illustrating the utility of only.minimal.msc is provided in the example section below. (The ordinary user is advised not to change the default value of this argument.)

In the next phase, minimally necessary disjunctions are built for each outcome by first testing whether single msc are necessary, then disjunctions of two msc, then of three, etc. (where a disjunction of msc counts as necessary iff it meets the threshold given by cov). Whenever a disjunction of msc (or a single msc) is found to be necessary, all supersets of that disjunction contain redundancies and are, thus, excluded from the further analysis. Finally, all and only those disjunctions of msc that meet both cov and con are issued as redundancy-free asf. To recover certain target structures in cases of noisy data, it may be useful to allow cna to also consider necessary conditions for further analysis that are not minimal. This can be accomplished by setting only.minimal.asf to FALSE, in which case *all* disjunctions of msc reaching the con and cov thresholds will be returned. (The ordinary user is advised not to change the default value of this argument.)

As the combinatorial search space for asf is potentially too large to be exhaustively scanned in reasonable time, the argument maxstep allows for setting an upper bound for the complexity of the generated asf. maxstep takes a vector of three integers c(i,j,k) as input, entailing that the generated asf have maximally j disjuncts with maximally i conjuncts each and a total of maximally k factor values (k is the maximal complexity). The default is maxstep = c(3,3,9).

Note that the default con and cov thresholds of 1 will often not yield any asf because real-life data tend to feature noise due to uncontrolled background influences. In such cases, users should gradually lower con and cov (e.g. in steps of 0.05) until cna finds solution formulas. con and cov should only be lowered below 0.75 with great caution. If thresholds of 0.75 do not result in solutions, the corresponding data feature such a high degree of noise that there is a severe risk of causal fallacies.

If cna finds asf, it combines them to complex solution formulas (csf). Asf with identical outcomes are not combined, for they do not represent a complex causal structure but model ambiguities with respect to one outcome. Asf with different outcomes can be concatenated to csf using two different signs: "*" and ",". If asf1 and asf2 have at least one factor in common, they are combined to "asf1 * asf2"; if they have no common factor, they are combined to "asf1, asf2". That is, csf with "*" as main operator represent *cohering* complex causal structures and the degree of coherence in the analyzed data is issued as coherence score (cf. coherence). Csf with "," as main operator represent *non-cohering* structures. For instance, the two asf (D + U <-> L) and (G + L <-> E) can be combined to the cohering csf "(D + U <-> L) * (G + L <-> E)", which represents a causal chain from D + U via L to E, whereas (D + U <-> L) and (G + F <-> E) yield the non-cohering csf "(D + U <-> L), (G + F <-> E)".

The default output of cna lists asf and csf with consistency, coverage, and complexity scores. But cna can calculate a number of further solution attributes: inus, exhaustiveness, faithfulness, coherence, and redundant, all of which are recovered by setting details to its non-default value TRUE. These attributes require explication (see also the package vignette).

complexity: Complexity corresponds to the number of exogenous factors in a solution. inus: The theory of causation underlying cna is called *INUS-theory* (Mackie 1974, ch. 3; Baumgartner 2008). Very roughly, it says that X is causally relevant to Y iff X is contained in a minimally necessary disjunction of minimally sufficient conditions of Y. It was originally designed for noise-free data that can be modeled with con = cov = 1. It turns out, however, that at consistency and coverage scores below 1 expressions can count as minimally necessary disjunctions of minimally sufficient conditions that, according to classical Boolean logic, could not possibly count as such at con = cov = 1. inus thus indicates whether or not a solution counts as an INUS solution relative to the strict criteria imposed by the INUS-theory for the case of con = cov = 1. If the user is only interested in INUS solutions, the argument inus.only is available; if inus.only = TRUE, only INUS solutions are built. The function behind the inus.only argument is also available as standalone function is.inus.

Exhaustiveness and faithfulness are two measures of model fit that quantify the degree of correspondence between the configurations that are, in principle, compatible with a solution and the configurations contained in the data from which that solution is derived. Roughly, exhaustiveness is high when *all* or most configurations *compatible* with a solution are in the data, whereas faithfulness is high when *no* or only few configurations that are *incompatible* with a solution are in the data. More specifically, `exhaustiveness` amounts to the ratio of the number of configurations in the data that are compatible with a solution to the number of configurations in total that are compatible with a solution. `faithfulness` amounts to the ratio of the number of configurations in the data that are compatible with a solution to the total number of configurations in the data. High exhaustiveness and faithfulness means that the configurations in the data are all and only the configurations that are compatible with the solution. Low exhaustiveness and/or faithfulness means that the data do not contain all configurations compatible with the solution and/or the data contain many configurations not compatible with the solution. In general, solutions with higher exhaustiveness and faithfulness scores are preferable over solutions with lower scores because they are better supported by the evidence in the data.

For details on `coherence` scores see `coherence`. Finally, `redundant`, which is only attributed to csf, determines whether a csf contains structurally redundant proper parts. That is the case if the csf has a proper part that is logically equivalent with the whole csf (cf. Baumgartner and Falk 2018). A csf with `redundant = TRUE` should not be causally interpreted. Rather, it must be further processed by `minimalizeCsf`, which eliminates redundancies from csf. The function identifying structural redundancies is also available as standalone function `redundant`.

`cna` does not need to be told which factor(s) are endogenous, it can infer that from the data. Still, when prior causal knowledge about an investigated process is available, `cna` can be prohibited from treating certain factors as potential causes of other factors by means of the argument `ordering`. If specified, that argument defines a causal ordering for the factors in `x`. For example, `ordering = list(c("A", "B"),"C")` determines that C is causally located *after* A and B, meaning that C is *not* a potential cause of A and B. In consequence, `cna` only checks whether values of A and B can be modeled as causes of values of C; the test for a causal dependency in the other direction is skipped. If the argument `ordering` is not specified or if it is given the `NULL` value (which is the argument's default value), `cna` searches for dependencies between all factors in `x`. An `ordering` does not need to explicitly mention all factors in an analyzed data frame. If only a subset of the factors are included in the `ordering`, the non-included factors are entailed to be causally before the included ones. Hence, `ordering = list("C")`, for instance, means that C is causally located after all other factors in the data, meaning that C is the ultimate outcome of the structure under scrutiny.

The argument `strict` determines whether the elements of one level in an ordering can be causally related or not. For example, if `ordering = list(c("A","B"),"C")` and `strict = TRUE`, then A and B—which are on the same level of the ordering—are excluded to be causally related and `cna` skips corresponding tests. By contrast, if `ordering = list(c("A","B"),"C")` and `strict = FALSE`, then `cna` also searches for dependencies among A and B. The default is `strict = FALSE`. If the user knows prior to the analysis that the data contain exactly one endogenous factor E and that the remaining exogenous factors are mutually causally independent, the appropriate function call should feature `cna(...,ordering = list("E"),strict = TRUE,...)`.

The argument `notcols` is used to calculate asf and csf for negative outcomes in data of `type "cs"` and `"fs"` (in `"mv"` data `notcols` has no meaningful interpretation and, correspondingly, issues an error message). If `notcols = "all"`, all factors in `x` are negated, i.e. their membership scores i are replaced by 1-i. If `notcols` is given a character vector of factors in `x`, only the factors in that vector are negated. For example, `notcols = c("A","B")` determines that only factors A and B are negated. The default is no negations, i.e. `notcols = NULL`.

`suff.only` is applicable whenever a complete `cna` analysis cannot be performed for reasons of computational complexity. In such a case, `suff.only = TRUE` forces `cna` to stop the analysis after the identification of msc, which will normally yield results even in cases when a complete analysis does not terminate. In that manner, it is possible to shed at least some light on the dependencies among the factors in `x`, in spite of an incomputable solution space.

`rm.const.factors` and `rm.dup.factors` are used to determine the handling of constant factors, i.e. factors with constant values in all cases (rows) in `x`, and of duplicated factors, i.e. factors that take identical value distributions in all cases in `x`. If `rm.const.factors = TRUE`, which is the default value, constant factors are removed from the data prior to the analysis, and if `rm.dup.factors = TRUE` (the default) all but the first of a set of duplicated factors are removed. From the perspective of configurational causal modeling, factors with constant values in all cases can neither be modeled as causes nor as outcomes; therefore, they can be removed prior to the analysis. Factors that take identical values in all cases cannot be distinguished configurationally, meaning they are one and the same factor as far as configurational causal modeling is concerned. Therefore, only one factor of a set of duplicated factors is standardly retained by `cna`.

The argument `what` can be specified both for the `cna` and the `print` function. It regulates what items of the output of `cna` are printed. If `what` is given the value “`t`”, the truth table is printed; if it is given an “`m`”, the msc are printed; if it is given an “`a`”, the asf are printed; if it is given a “`c`”, the csf are printed. `what = "all"` or `what = "tmac"` determine that all output items are printed. Note that `what` has no effect on the computations that will be performed when executing `cna`; it only determines how the result will be printed. The default output of `cna` is `what = "ac"`. It first returns the implemented ordering. Second, the asf and, third, the csf are reported. If csf are the same as asf, this is indicated by "Same as asf". In case of `suff.only = TRUE`, `what` defaults to `"m"`.

`cna` only includes factor configurations in the analysis that are actually instantiated in the data. The argument `cutoff` determines the minimum membership score required for a factor or a combination of factors to count as instantiated. It takes values in the unit interval [0,1] with a default of 0.5. `border` specifies whether factor combinations with membership scores equal to `cutoff` are rounded up (`border = "up"`), rounded down (`border = "down"`), which is the default, or dropped from the analysis (`border = "drop"`).

The arguments `digits`, `nsolutions`, and `show.cases` apply to the `print` function, which takes an object of class “cna” as first input. `digits` determines how many digits of consistency, coverage, coherence, exhaustiveness, and faithfulness scores are printed, while `nsolutions` fixes the number of conditions and solutions to print. `nsolutions` applies separately to minimally sufficient conditions, atomic solution formulas, and complex solution formulas. `nsolutions = "all"` recovers all minimally sufficient conditions, atomic and complex solution formulas. `show.cases` is applicable if the `what` argument is given the value “`t`”. In that case, `show.cases = TRUE` yields a truth table featuring a “cases” column, which assigns cases to configurations.

The option “spaces” controls how the conditions are rendered. The current setting is queried by typing `getOption("spaces")`. The option specifies characters that will be printed with a space before and after them. The default is `c("<->","->","+")`. A more compact output is obtained with `option(spaces = NULL)`.

Value

`cna` returns an object of class “cna”, which amounts to a list with the following components:

`call`:	the executed function call
`x`:	the processed data frame or truth table

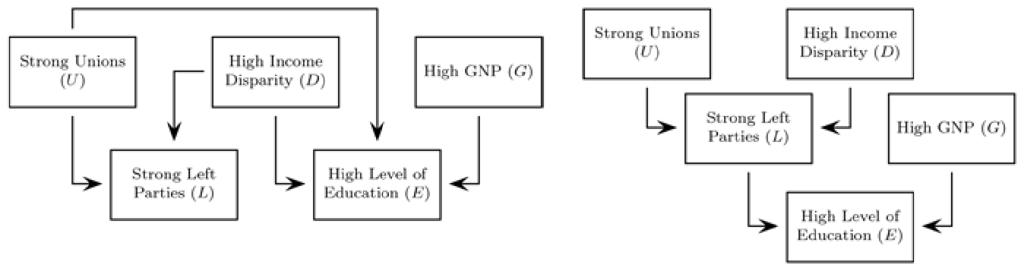
`ordering`:	the implemented ordering
`truthTab`:	the object of class "truthTab", as input to `cna`
`truthTab_out`:	the object of class "truthTab", after modification according to `notcols`
`solution`:	the solution object, which itself is composed of lists exhibiting msc, asf, and csf for all factors in `x`
`what`:	the values given to the `what` argument
`details`:	the calculated solution attributes

Contributors

Epple, Ruedi: development, testing
Thiem, Alrik: testing

Note

In the first example described below (in *Examples*), the two resulting complex solution formulas represent a common cause structure and a causal chain, respectively. The common cause structure is graphically depicted in figure (a) below, the causal chain in figure (b).


Strong Unions (U)
High Income Disparity (D)
High GNP (G)
Strong Left Parties (L)
High Level of Education (E)
Strong Unions (U)
High Income Disparity (D)
Strong Left Parties (L)
High GNP (G)
High Level of Education (E)

(a) (b)

References

Basurto, Xavier. 2013. "Linking Multi-Level Governance to Local Common-Pool Resource Theory using Fuzzy-Set Qualitative Comparative Analysis: Insights from Twenty Years of Biodiversity Conservation in Costa Rica." *Global Environmental Change* 23(3):573-87.

Baumgartner, Michael. 2008. "Regularity Theories Reassessed." *Philosophia* 36:327-354.

Baumgartner, Michael. 2009a. "Inferring Causal Complexity." *Sociological Methods & Research* 38(1):71-101.

Baumgartner, Michael. 2009b. "Uncovering Deterministic Causal Structures: A Boolean Approach." *Synthese* 170(1):71-96.

Baumgartner, Michael and Christoph Falk. 2018. "Boolean Difference-Making: A Modern Regularity Theory of Causation". *PhilSci Archive*. url: http://philsciarchive.pitt.edu/id/eprint/14876.

Hartmann, Christof, and Joerg Kemmerzell. 2010. "Understanding Variations in Party Bans in Africa." *Democratization* 17(4):642-65. DOI: 10.1080/13510347.2010.491189.

Krook, Mona Lena. 2010. "Women's Representation in Parliament: A Qualitative Comparative Analysis." *Political Studies* 58(5):886-908.

Mackie, John L. 1974. *The Cement of the Universe: A Study of Causation.* Oxford: Oxford University Press.

Ragin, Charles C. 2006. “Set Relations in Social Research: Evaluating Their Consistency and Coverage”. *Political Analysis* 14(3):291-310.

Wollebaek, Dag. 2010. “Volatility and Growth in Populations of Rural Associations.” *Rural Sociology* 75:144-166.

See Also

truthTab, condition, condTbl, selectCases, makeFuzzy, some, coherence, minimalizeCsf, randomConds, is.submodel, is.inus, redundant, full.tt, d.educate, d.women, d.pban, d.autonomy

Examples

```
# Ideal crisp-set data from Baumgartner (2009a) on education levels in western democracies
#----------------------------------------------------------------------------------------
# Exhaustive CNA without constraints on the search space; print atomic and complex
# solution formulas (default output).
cna.educate <- cna(d.educate)
cna.educate
# The two resulting complex solution formulas represent a common cause structure
# and a causal chain, respectively. The common cause structure is graphically depicted
# in (Note, figure (a)), the causal chain in (Note, figure (b)).

# Print only complex solution formulas.
print(cna.educate, what = "c")

# Print only atomic solution formulas.
print(cna.educate, what = "a")

# Print only minimally sufficient conditions.
print(cna.educate, what = "m")

# Print only the truth table.
print(cna.educate, what = "t")

# CNA with negations of the factors E and L.
cna(d.educate, notcols = c("E","L"))

# CNA with negations of all factors.
cna(d.educate, notcols = "all")

# Print msc, asf, and csf with all solution attributes.
cna(d.educate, what = "mac", details = TRUE)

# Add only the non-standard solution attributes "inus" and "faithfulness".
cna(d.educate, details = c("i", "f"))

# Print solutions without spaces before and after "+".
options(spaces = c("<->", "->" ))
cna(d.educate, details = c("i", "f"))

# Print solutions with spaces before and after "*".
options(spaces = c("<->", "->", "*" ))
```

```
cna(d.educate, details = c("i", "f"))

# Restore the default of the option "spaces".
options(spaces = c("<->", "->", "+"))


# Crisp-set data from Krook (2010) on representation of women in western-democratic parliaments
# ------------------------------------------------------------------------------------------
# This example shows that CNA can infer which factors are causes and which ones
# are effects from the data. Without being told which factor is the outcome,
# CNA reproduces the original QCA of Krook (2010).
ana1 <- cna(d.women, maxstep = c(3, 4, 9), details = c("e", "f"))
ana1

# The two resulting asf only reach an exhaustiveness score of 0.438, meaning that
# not all configurations that are compatible with the asf are contained in the data
# "d.women". Here is how to extract the configurations that are compatible with
# the first asf but are not contained in "d.women":
library(dplyr)
setdiff(tt2df(selectCases(asf(ana1)$condition[1], full.tt(d.women))),
        d.women)


# Highly ambiguous crisp-set data from Wollebaek (2010) on very high volatility of
# grassroots associations in Norway
# --------------------------------------------------------------------------------
# csCNA with ordering from Wollebaek (2010) [Beware: due to massive ambiguities, this analysis
# will take about 20 seconds to compute.]
cna(d.volatile, ordering = list("VO2"), maxstep = c(6, 6, 16))

# Using suff.only, CNA can be forced to abandon the analysis after minimization of sufficient
# conditions. [This analysis terminates quickly.]
cna(d.volatile, ordering = list("VO2"), maxstep = c(6, 6, 16), suff.only = TRUE)

# Similarly, by using the default maxstep, CNA can be forced to only search for asf and csf
# with reduced complexity. [This analysis also terminates quickly.]
cna(d.volatile, ordering = list("VO2"))


# Multi-value data from Hartmann & Kemmerzell (2010) on party bans in Africa
# ---------------------------------------------------------------------------
# mvCNA with causal ordering that corresponds to the ordering in Hartmann & Kemmerzell
# (2010); coverage cutoff at 0.95 (consistency cutoff at 1), maxstep at (6, 6, 10).
cna.pban <- mvcna(d.pban, ordering = list(c("C","F","T","V"),"PB"), cov = .95,
                  maxstep = c(6, 6, 10), what = "all")
cna.pban

# The previous function call yields a total of 14 asf and csf, only 5 of which are
# printed in the default output. Here is how to extract all 14 asf and csf.
asf(cna.pban)
csf(cna.pban)

# [Note that all of these 14 causal models reach considerably better consistency and
# coverage scores than the one model Hartmann & Kemmerzell (2010) present in their paper,
# which they generated using the TOSMANA software, version 1.3:
# T=0 + T=1 + C=2 + T=1*V=0 + T=2*V=0 <-> PB=1
mvcond("T=0 + T=1 + C=2 + T=1*V=0 + T=2*V=0 <-> PB = 1", d.pban)

# That is, not only does TOSMANA fail to recover model ambiguities in this case, it
# also issues a model whose fit is significantly below the models this data set would
# warrant.]

# Extract all minimally sufficient conditions.
msc(cna.pban)

# Alternatively, all msc, asf, and csf can be recovered by means of the nsolutions
# argument of the print function.
print(cna.pban, nsolutions = "all")

# Print the truth table with the "cases" column.
print(cna.pban, what = "t", show.cases = TRUE)

# Build solution formulas with maximally 4 disjuncts.
mvcna(d.pban, ordering = list(c("C","F","T","V"),"PB"), cov = .95, maxstep = c(4, 4, 10))

# Only print 2 digits of consistency and coverage scores.
print(cna.pban, digits = 2)

# Build all but print only two msc for each factor and two asf and csf.
print(mvcna(d.pban, ordering = list(c("C","F","T","V"),"PB"), cov = .95,
      maxstep = c(6, 6, 10), what = "all"), nsolutions = 2)

# Lowering the consistency instead of the coverage threshold yields further models with
# excellent fit scores; print only asf.
mvcna(d.pban, ordering = list(c("C","F","T","V"),"PB"), con = .93, what = "a",
      maxstep = c(6, 6, 10))

# Importing an ordering from prior causal knowledge is unnecessary for d.pban. PB
# is the only factor in that data that could possibly be an outcome.
mvcna(d.pban, cov = .95, maxstep = c(6, 6, 10))


# Fuzzy-set data from Basurto (2013) on autonomy of biodiversity institutions in Costa Rica
# ---------------------------------------------------------------------------------------
# Basurto investigates two outcomes: emergence of local autonomy and endurance thereof. The
# data for the first outcome is contained in rows 1-14 of d.autonomy, the data for the second
# outcome in rows 15-30. For each outcome, the author distinguishes between local ("EM",
# "SP", "CO"), national ("CI", "PO") and international ("RE", "CN", "DE") conditions. Here,
# we first apply fsCNA to replicate the analysis for the local conditions of the endurance of
# local autonomy.
dat1 <- d.autonomy[15:30, c("AU","EM","SP","CO")]
fscna(dat1, ordering = list("AU"), strict = TRUE, con = .9, cov = .9)

# The fsCNA model has significantly better consistency (and equal coverage) scores than the
# model presented by Basurto (p. 580): SP*EM + CO <-> AU, which he generated using the
# fs/QCA software.
```

```
fscond("SP*EM + CO <-> AU", dat1) # both EM and CO are redundant to account for AU

# If we allow for dependencies among the conditions by setting strict = FALSE, CNA reveals
# that SP is a common cause of both AU and EM:
fscna(dat1, ordering = list("AU"), strict = FALSE, con = .9, cov = .9)

# Here is the analysis for the international conditions of autonomy endurance, which
# yields the same model presented by Basurto (plus one model Basurto does not mention):
dat2 <- d.autonomy[15:30, c("AU","RE", "CN", "DE")]
fscna(dat2, ordering = list("AU"), con = .9, con.msc = .85, cov = .85)

# But there are other models (here printed with all solution attributes)
# that fare equally well.
fscna(dat2, ordering = list("AU"), con = .85, cov = .9, details = TRUE)

# Finally, here is an analysis of the whole data set, showing that across the whole period
# 1986-2006, the best causal model of local autonomy (AU) renders that outcome dependent
# only on local direct spending (SP):
fscna(d.autonomy, ordering = list("AU"), strict = TRUE, con = .85, cov = .9,
                maxstep = c(5, 5, 11), details = TRUE)

# Only build INUS solutions.
asf(fscna(d.autonomy, ordering = list("AU"), strict = TRUE, con = .85, cov = .9,
                    maxstep = c(5, 5, 11), details = TRUE, inus.only = TRUE))


# Highly ambiguous artificial data to illustrate exhaustiveness
# ---------------------------------------------------------------
mycond <- "(D + C*f <-> A)*(C*d + c*D <-> B)*(B*d + D*f <-> C)*(c*B + B*f <-> E)"
dat1 <- selectCases(mycond)
ana1 <- cna(dat1, details = TRUE)
# There are almost 2M csf. This is how to build the first 360 of them:
csf360 <- csf(ana1, 360)
# Most of these csf are compatible with more configurations than are contained in
# dat1. Only 32 of csf360 are perfectly exhaustive (i.e. all compatible
# configurations are contained in dat1):
subset(csf360, exhaustiveness == 1)

# Eliminate structural redundancies.
minimalizeCsf(subset(csf360, exhaustiveness == 1)$condition, dat1)


# Inverse search trials to assess the correctness of cna
# --------------------------------------------------------
# 1. Ideal mv data, i.e. perfect consistencies and coverages, without data fragmentation.
# Define the target and generate data on the target.
target <- "(A=1*B=2 + A=4*B=3 <-> C=1)*(C=4*D=1 + C=2*D=4 <-> E=4)"
dat1 <- allCombs(c(4, 4, 4, 4, 4))
dat2 <- selectCases(target, dat1, type = "mv")
# Analyze the simulated data with cna.
test1 <- mvcna(dat2)
# Eliminate possible structural redundancies.
test1 <- minimalizeCsf(test1)
```

```
# Check whether a correctness-preserving submodel of the target is among the
# returned solutions.
is.submodel(test1$condition, target)

# Same test as above with data fragmentation, i.e. with non-ideal data:
# only 100 of 472 observable configurations are actually
# observed. [Repeated runs will generate different data.]
dat3 <- some(dat2, n = 100, replace = TRUE)
test2 <- mvcna(dat3)
test2 <- minimalizeCsf(test2, 50)
is.submodel(test2$condition, target)

# 2. Fs data with imperfect consistencies (con = 0.8) and coverages (cov = 0.8);
# about 150 cases (depending on the seed). Randomly generated target asf.
# [Repeated runs will generate different targets and data.]
target <- randomAsf(full.tt(5), compl = c(2,3))
outcome <- as.vector(sapply(cna:::extract_asf(target), cna:::rhs))
# Simulate the data with con =  cov = 0.8.
dat1 <- allCombs(c(2, 2, 2, 2, 2)) - 1
dat2 <- some(truthTab(dat1), n = 200, replace = TRUE)
dat3 <- makeFuzzy(tt2df(dat2), fuzzvalues = seq(0, 0.45, 0.01))
dat4 <- selectCases1(target, con = .8, cov = .8, type = "fs", dat3)
# Analyze the simulated data with cna.
test3 <- fscna(dat4, ordering = list(outcome), strict = TRUE, con = .8, cov = .8)
# Check whether a correctness-preserving submodel of the target is among the
# returned solutions.
is.submodel(asf(test3)$condition, target)

# Same test as above with data fragmentation: only 80 of about 150 possible
# cases are actually observed. [Repeated runs will generate different data.]
dat5 <- some(dat4, n = 80, replace = TRUE)
fscna(dat5, ordering = list(outcome), strict = TRUE, con = .8, cov = .8)
test4 <- fscna(dat5, ordering = list(outcome), strict = TRUE, con = .8, cov = .8)
is.submodel(asf(test4)$condition, target)


# Illustration of only.minimal.msc = FALSE
# ----------------------------------------
# Simulate noisy data on the causal structure "a*B*d + A*c*D <-> E"
set.seed(1324557857)
mydata <- allCombs(rep(2, 5)) - 1
dat <- makeFuzzy(mydata, fuzzvalues = seq(0, 0.5, 0.01))
dat <- tt2df(selectCases1("a*B*d + A*c*D <-> E", con = .8, cov = .8, dat))

# In dat, "a*B*d + A*c*D <-> E" has the following con and cov scores:
as.condTbl(fscond("a*B*d + A*c*D <-> E", dat))

# The standard algorithm of cna will, however, not find this structure with
# con = cov = 0.8 because one of the disjuncts (a*B*d) does not meet the con
# threshold:
as.condTbl(fscond(c("a*B*d <-> E", "A*c*D <-> E"), dat))
fscna(dat, ordering=list("E"), strict = TRUE, con = .8, cov = .8)
```

```
# With the argument con.msc we can lower the con threshold for msc, but this does not
# recover "a*B*d + A*c*D <-> E" either:
cna2 <- fscna(dat, ordering=list("E"), strict = TRUE, con = .8, cov = .8, con.msc = .7)
cna2
msc(cna2)

# The reason is that "a*B -> E" and "c*D -> E" now also meet the con.msc threshold and,
# therefore, neither "a*B*d -> E" nor "A*c*D -> E" are contained in the msc---
# because of violated minimality. In a situation like this, lifting the minimality
# requirement via only.minimal.msc = FALSE allows cna to find the intended target:
fscna(dat, ordering=list("E"), strict=TRUE, con = .8, cov = .8, con.msc = .7,
      only.minimal.msc = FALSE)
```

---

coherence *Calculate the coherence of complex solution formulas*

Description

Calculates the coherence measure of complex solution formulas (csf).

Usage

```
coherence(cond, tt, type)
```

Arguments

`cond`	Character vector specifying an asf or csf.
`tt`	Data frame or `truthTab`.
`type`	Character vector specifying the type of `tt`: `"cs"` (crisp-set), `"mv"` (multi-value), or `"fs"` (fuzzy-set). Defaults to the type of `tt`, if `tt` is a `truthTab` or to `"cs"` otherwise.

Details

Coherence is a measure for model fit that is custom-built for complex solution formulas (csf). It measures the degree to which the atomic solution formulas (asf) combined in a csf cohere, i.e. are instantiated together in `tt` rather than independently of one another. More concretely, coherence is the ratio of the number of cases satisfying all asf contained in a csf to the number of cases satisfying at least one asf in the csf. For example, if the csf contains the three asf asf1, asf2, asf3, coherence amounts to | asf1 * asf2 * asf3 | / | asf1 + asf2 + asf3 |, where |...| expresses the cardinality of the set of cases instantiating the corresponding expression. For asf, `coherence` returns 1. For boolean conditions (see `condition`), the coherence measure is not defined and `coherence` hence retuns `NA`. For multiple csf that do not have a factor in common, `coherence` returns the minimum of the separate coherence scores.

Value

Numeric vector of coherence values.

See Also

cna, condition, selectCases, allCombs, full.tt, condTbl

Examples

```
# Perfect coherence.
dat1 <- selectCases("(A*b <-> C)*(C+D <-> E)")
coherence("(A*b <-> C)*(C + D <-> E)", dat1)
csf(cna(dat1, details = "c"))

# Non-perfect coherence.
dat2 <- selectCases("(a*B <-> C)*(C + D<->E)*(F*g <-> H)")
dat3 <- rbind(tt2df(dat2), c(0,1,0,1,1,1,0,1))
coherence("(a*B <-> C)*(C + D <-> E)*(F*g <-> H)", dat3)
csf(cna(dat3, con=.88, details = "c"))
```

`condition` *Uncover relevant properties of msc, asf, and csf in a data frame or* `truthTab`

Description

The `condition` function provides assistance to inspect the properties of msc, asf, and csf (as returned by `cna`) in a data frame or `truthTab`, but also of any other Boolean function. `condition` reveals which configurations and cases instantiate a given msc, asf, or csf and lists consistency and coverage scores.

Usage

```
condition(x, ...)

## Default S3 method:
condition(x, tt, type, add.data = FALSE,
          force.bool = FALSE, rm.parentheses = FALSE, ...)
## S3 method for class 'condTbl'
condition(x, tt, ...)
cscond(...)
mvcond(...)
fscond(...)

## S3 method for class 'condList'
print(x, ...)
## S3 method for class 'condList'
summary(object, ...)

## S3 method for class 'cond'
print(x, digits = 3, print.table = TRUE,
      show.cases = NULL, add.data = NULL, ...)
```

```
group.by.outcome(condlst, cases = TRUE)
```

Arguments

`x`	Character vector specifying a Boolean expression as `"A + B*C -> D"`, where `"A"`, `"B"`, `"C"`, `"D"` are column names in `tt`.
`tt`	Data frame or `truthTab` (see `truthTab`).
`type`	Character vector specifying the type of `tt`: `"cs"` (crisp-set), `"mv"` (multi-value), or `"fs"` (fuzzy-set). Defaults to the type of `tt`, if `tt` is a `truthTab` or to `"cs"` otherwise.
`add.data`	Logical; if `TRUE`, `tt` is attached to the output. Alternatively, the `tt` can be specified as the `add.data` argument in `print.cond`.
`force.bool`	Logical; if `TRUE`, `x` is interpreted as a mere Boolean function, not as a causal model.
`rm.parentheses`	Logical; if `TRUE`, parantheses around `x` are removed prior to evaluation.
`digits`	Number of digits to print in consistency and coverage scores.
`print.table`	Logical; if `TRUE`, the table assigning configurations and cases to conditions is printed.
`show.cases`	In `print.cond`: logical; if `TRUE`, the attribute "cases" of the `truthTab` is printed. Same default behavior as in `print.truthTab`.
`object`	Object of class "condList", as returned by `condition`.
`condlst`	List of objects, each of them of class "cond", as returned by `condition`.
`cases`	Logical; if `TRUE`, the returned data frame has a column named "cases".
`...`	In `cscond`, `mvcond`, `fscond`: any formal argument of `condition` except `type`.

Details

Depending on the processed data frame or `truthTab`, the solutions output by `cna` are often ambiguous; that is, it can happen that many solution formulas fit the data equally well. In such cases, the data alone are insufficient to single out one solution. While `cna` simply lists the possible solutions, the `condition` function is intended to provide assistance in comparing different minimally sufficient conditions (msc), atomic solution formulas (asf), and complex solution formulas (csf) in order to have a better basis for selecting among them.

Most importantly, the output of the `condition` function highlights in which configurations and cases in the data an msc, asf, and csf is instantiated. Thus, if the user has independent causal knowledge about particular configurations or cases, the information received from `condition` may be helpful in selecting the solutions that are consistent with that knowledge. Moreover, the `condition` function allows for directly contrasting consistency and coverage scores or frequencies of different conditions contained in returned asf.

The `condition` function is independent of `cna`. That is, any msc, asf, or csf—irrespective of whether they are output by `cna`—can be given as input to `condition`. Even Boolean expressions that do not have the syntax of CNA solution formulas can be passed to `condition`.

The first required input `x` of `condition` is a character vector consisting of Boolean formulas composed of factor names that are column names of `tt`, which is the second required input. `tt` can be a `truthTab` or a data frame. In the latter case, `condition` must be told what type of data `tt` contains, and the data frame will be converted to a `truthTab`. Data that feature factors taking values 1 or 0 only are called *crisp-set*, in which case the `type` argument takes its default value `"cs"`. If the data contain at least one factor that takes more than two values, e.g. {1,2,3}, the data count as *multi-value*, which is indicated by `type = "mv"`. Data featuring at least one factor taking real values from the interval [0,1] count as *fuzzy-set*, which is specified by `type = "fs"`. To abbreviate the specification of the data type, the functions `cscond(x,tt,...)`, `mvcond(x,tt,...)`, and `fscond(x,tt,...)` are available as shorthands for `condition(x,tt,type = "cs",...)`, `condition(x,tt,type = "mv",...)`, and `condition(x,tt,type = "fs",...)`, respectively.

Conjunction can be expressed by "`*`" or "`&`", disjunction by "`+`" or "`|`", negation can be expressed by "`-`" or "`!`" or, in case of crisp-set or fuzzy-set data, by changing upper case into lower case letters and vice versa, implication by "`->`", and equivalence by "`<->`". Examples are

- `A*b -> C,A+b*c+!(C+D),A*B*C + -(E*!B),C -> A*B + a*b`
- `(A=2*B=4 + A=3*B=1 <-> C=2)*(C=2*D=3 + C=1*D=4 <-> E=3)`
- `(A=2*B=4*!(A=3*B=1)) | !(C=2|D=4)*(C=2*D=3 + C=1*D=4 <-> E=3)`

Three types of conditions are distinguished:

- The type *boolean* comprises Boolean expressions that do not have the syntactic form of causal models, meaning the corresponding character strings in the argument `x` do not have an "`->`" or "`<->`" as main operator. Examples: `"A*B + C"` or `"-(A*B + -(C+d))"`. The expression is evaluated and written into a data frame with one column. Frequency is attached to this data frame as an attribute.
- The type *atomic* comprises expressions that have the syntactic form of atomic causal models, i.e. asf, meaning the corresponding character strings in the argument `x` have an "`->`" or "`<->`" as main operator. Examples: `"A*B + C -> D"` or `"A*B + C <-> D"`. The expressions on both sides of "`->`" and "`<->`" are evaluated and written into a data frame with two columns. Consistency and coverage are attached to these data frames as attributes.
- The type *complex* represents complex causal models, i.e. csf. Example: `"(A*B + a*b <-> C)*(C*d + c*D <-> E)"`. Each component must be a causal model of type *atomic*. These components are evaluated separately and the results stored in a list. Consistency and coverage of the complex expression are then attached to this list.

The types of the character strings in the input `x` are automatically discerned and thus do not need be specified by the user.

If `force.bool = TRUE`, expressions with "`->`" or "`<->`" are treated as type *boolean*, i.e. only their frequencies are calculated. Enclosing a character string representing a causal model in parentheses has the same effect as specifying `force.bool = TRUE`. `rm.parentheses = TRUE` removes parentheses around the expression prior to evaluation, and thus has the reverse effect of setting `force.bool = TRUE`.

If `add.data = TRUE`, `tt` is appended to the output such as to facilitate the analysis and evaluation of a model on the case level.

The `digits` argument of the `print` function determines how many digits of consistency and coverage scores are printed. If `print.table = FALSE`, the table assigning conditions to configurations and cases is omitted, i.e. only frequencies or consistency and coverage scores are returned. `row.names = TRUE` also lists the row names in `tt`. If rows in a `tt` are instantiated by many cases, those cases are not printed by default. They can be recovered by `show.cases = TRUE`.

`group.by.outcome` takes a `condlist` as input, i.e. a list of “cond” objects, as it is returned by `condition`, and combines the entries in that lists into a data frame with a larger number of columns. The additional attributes (consistencies etc.) are thereby removed.

Value

`condition` returns a list of objects, each of them corresponding to one element of the input vector `x`. The list has a class attribute “condList”, the list elements (i.e., the individual conditions) are of class “cond” and have a more specific class label “booleanCond”, “atomicCond” or “complexCond”, according to the condition type. The components of class “booleanCond” or “atomicCond” are amended data frames, those of class “complexCond” are lists of amended data frames.

`group.by.outcome` returns a list of data frames, one data frame for each factor appearing as an outcome in `condlst`.

`print` and `summary` methods

`print.condList` essentially executes `print.cond` successively for each list element/condition. All arguments in `print.condList` are thereby passed to `print.cond`, i.e. `digits`, `print.table`, `show.cases`, `add.data` can also be specified when printing the complete list of conditions.

The `summary` method for class “condList” is identical to `print`ing with `print.table = FALSE`.

The option “spaces” controls how the conditions are rendered in certain contexts. The current setting is queried by typing `getOption("spaces")`. The option specifies characters that will be printed with a space before and after them. The default is `c("<->","->","+")`. A more compact output is obtained with `option(spaces = NULL)`.

References

Emmenegger, Patrick. 2011. “Job Security Regulations in Western Democracies: A Fuzzy Set Analysis.” *European Journal of Political Research* 50(3):336-64.

Lam, Wai Fung, and Elinor Ostrom. 2010. “Analyzing the Dynamic Complexity of Development Interventions: Lessons from an Irrigation Experiment in Nepal.” *Policy Sciences* 43 (2):1-25.

Ragin, Charles. 2008. *Redesigning Social Inquiry: Fuzzy Sets and Beyond*. Chicago, IL: University of Chicago Press.

See Also

`cna`, `truthTab`, `condTbl`, `d.irrigate`

Examples

```
# Crisp-set data from Lam and Ostrom (2010) on the impact of development interventions
# ------------------------------------------------------------------------------------
# Build a truth table for d.irrigate.
irrigate.tt <- truthTab(d.irrigate)

# Any Boolean functions involving the factors "A", "R", "F", "L", "C", "W" in d.irrigate
# can be tested by condition.
condition("A*r + L*C", irrigate.tt)
condition(c("A*r + !(L*C)", "A*-(L | -F)", "C -> A*R + C*l"), irrigate.tt)
condition(c("A*r + L*C -> W", "!(A*L*R -> W)", "(A*R + C*l <-> F)*(W*a -> F)"),
          irrigate.tt)

# Group expressions with "->" by outcome.
irrigate.con <- condition(c("A*r + L*C -> W", "A*L*R -> W", "A*R + C*l -> F", "W*a -> F"),
                          irrigate.tt)
group.by.outcome(irrigate.con)

# Pass minimally sufficient conditions inferred by cna to condition.
irrigate.cna1 <- cna(d.irrigate, ordering = list(c("A","R","L"),c("F","C"),"W"), con = .9)
condition(msc(irrigate.cna1)$condition, irrigate.tt)

# Pass atomic solution formulas inferred by cna to condition.
irrigate.cna1 <- cna(d.irrigate, ordering = list(c("A","R","L"),c("F","C"),"W"), con = .9)
condition(asf(irrigate.cna1)$condition, irrigate.tt)

# Group by outcome.
irrigate.cna1.msc <- condition(msc(irrigate.cna1)$condition, irrigate.tt)
group.by.outcome(irrigate.cna1.msc)

irrigate.cna2 <- cna(d.irrigate, con = .9)
irrigate.cna2a.asf <- condition(asf(irrigate.cna2)$condition, irrigate.tt)
group.by.outcome(irrigate.cna2a.asf)

# Add data.
(irrigate.cna2b.asf <- condition(asf(irrigate.cna2)$condition, irrigate.tt,
                                     add.data = TRUE))

# No spaces before and after "+".
options(spaces = c("<->", "->" ))
irrigate.cna2b.asf

# No spaces at all.
options(spaces = NULL)
irrigate.cna2b.asf

# Restore the default spacing.
options(spaces = c("<->", "->", "+"))

# Print only consistency and coverage scores.
print(irrigate.cna2a.asf, print.table = FALSE)
summary(irrigate.cna2a.asf)

# Print only 2 digits of consistency and coverage scores.
print(irrigate.cna2b.asf, digits = 2)

# Instead of a truth table as output by truthTab, it is also possible to provide a data
# frame as second input.
condition("A*r + L*C", d.irrigate, type = "cs")
condition(c("A*r + L*C", "A*L -> F", "C -> A*R + C*l"), d.irrigate, type = "cs")
condition(c("A*r + L*C -> W", "A*L*R -> W", "A*R + C*l -> F", "W*a -> F"), d.irrigate,
          type = "cs")
```

```
# Fuzzy-set data from Emmenegger (2011) on the causes of high job security regulations
# ------------------------------------------------------------------------------------
# Compare the CNA solutions for outcome JSR to the solution presented by Emmenegger
# S*R*v + S*L*R*P + S*C*R*P + C*L*P*v -> JSR (p. 349), which he generated by fsQCA as
# implemented in the fs/QCA software, version 2.5.
jobsecurity.cna <- fscna(d.jobsecurity, ordering=list("JSR"), strict = TRUE, con = .97,
                         cov= .77, maxstep = c(4, 4, 15))
compare.sol <- fscond(c(asf(jobsecurity.cna)$condition, "S*R*v + S*L*R*P + S*C*R*P +
                         C*L*P*v -> JSR"), d.jobsecurity)
summary(compare.sol)
print(compare.sol, add.data = d.jobsecurity)
group.by.outcome(compare.sol)

# There exist even more high quality solutions for JSR.
jobsecurity.cna2 <- fscna(d.jobsecurity, ordering=list("JSR"), strict = TRUE, con = .95,
                          cov= .8, maxstep = c(4, 4, 15))
compare.sol2 <- fscond(c(asf(jobsecurity.cna2)$condition, "S*R*v + S*L*R*P + S*C*R*P +
                         C*L*P*v -> JSR"), d.jobsecurity)
summary(compare.sol2)
group.by.outcome(compare.sol2)


# Simulate multi-value data
# -------------------------
library(dplyr)
# Define the data generating structure.
groundTruth <- "(A=2*B=1 + A=3*B=3 <-> C=1)*(C=1*D=2 + C=2*D=3 <-> E=3)"
# Generate ideal data on groundTruth.
fullData <- allCombs(c(3, 3, 2, 3, 3))
idealData <- tt2df(selectCases(groundTruth, fullData, type = "mv"))
# Randomly add 15% inconsistent cases.
inconsistentCases <- setdiff(fullData, idealData)
realData <- rbind(idealData, inconsistentCases[sample(1:nrow(inconsistentCases),
                                               nrow(idealData)*0.15), ])
# Determine model fit of groundTruth and its submodels.
condition(groundTruth, realData, type = "mv")
mvcond(groundTruth, realData)
mvcond("A=2*B=1 + A=3*B=3 <-> C=1", realData)
mvcond("A=2*B=1 + A=3*B=3 <-> C=1", realData, force.bool = TRUE)
mvcond("(C=1*D=2 + C=2*D=3 <-> E=3)", realData)
mvcond("(C=1*D=2 + C=2*D=3 <-> E=3)", realData, rm.parentheses = TRUE)
mvcond("(C=1*D=2 +!(C=2*D=3 + A=1*B=1) <-> E=3)", realData)
# Manually calculate unique coverages, i.e. the ratio of an outcome's instances
# covered by individual msc alone (for details on unique coverage cf.
# Ragin 2008:63-68).
summary(mvcond("A=2*B=1 * -(A=3*B=3) <-> C=1", realData)) # unique coverage of A=2*B=1
summary(mvcond("-(A=2*B=1) * A=3*B=3 <-> C=1", realData)) # unique coverage of A=3*B=3
```

condTbl — *Extract conditions and solutions from an object of class "cna"*

Description

Given a solution object x produced by cna, msc(x) extracts all minimally sufficient conditions, asf(x) all atomic solution formulas, and csf(x,n) extracts at least n complex solution formulas. All solution attributes (details) that are saved in x are recovered as well. The three functions return a data frame with the additional class attribute condTbl.

as.condTbl reshapes the output produced by condition in such a way as to make it identical to the output returned by msc, asf, and csf.

condTbl executes condition and returns a concise summary table featuring consistencies and coverages.

Usage

```
msc(x, details = x$details)
asf(x, details = x$details, warn_details = TRUE)
csf(x, n = 20, tt = x$truthTab, details = x$details,
    asfx = asf(x, details, warn_details = FALSE))

## S3 method for class 'condTbl'
print(x, digits = 3, quote = FALSE, row.names = TRUE, ...)

condTbl(...)
as.condTbl(x, ...)
```

Arguments

x	Object of class “cna”. In as.condTbl, x is a list of evaluated conditions as returned by condition.
details	Either TRUE/FALSE or a character vector specifying which solution attributes to print (see cna). Note that msc and asf can only display attributes that are saved in x, i.e. those that have been requested in the details argument within the call of cna.
warn_details	Logical; if TRUE, a warning is issued when some attribute requested in details is not available in x (parameter for internal use).
n	The minimal number of csf to be calculated.
tt	A truthTab.
asfx	Object of class “condTbl” resulting from asf.
digits	Number of digits to print in consistency, coverage, exhaustiveness, faithfulness, and coherence scores.
quote, row.names	As in print.data.frame.
...	All arguments in condTbl are passed on to condition.

Details

Depending on the processed data, the solutions output by cna are often ambiguous, to the effect that many atomic and complex solutions fit the data equally well. To facilitate the inspection of the cna output, however, the latter standardly returns only 5 minimally sufficient conditions and 5 atomic and complex solution formulas for each outcome. msc can be used to extract *all* minimally sufficient conditions from an object x of class "cna", asf to extract *all* atomic solution formulas, and csf to extract at least n complex solution formulas from x. All solution attributes (details) that are saved in x are recovered as well. The outputs of msc, asf, and csf can be further processed by the condition function.

The argument digits applies to the print function. It determines how many digits of consistency, coverage, exhaustiveness, faithfulness, and coherence scores are printed. The default value is 3.

The function as.condTbl takes a list of objects of class "cond" that are returned by the condition function as input, and reshapes these objects in such a way as to make them identical to the output returned by msc, asf, and csf.

condTbl(...) is identical with as.condTbl(condition(...)).

Value

msc, asf, csf, and as.condTbl return objects of class "condTbl", a data.frame which features the following components:

outcome:	the outcomes
condition:	the relevant conditions or solutions
consistency:	the consistency scores
coverage:	the coverage scores
complexity:	the complexity scores
inus:	whether the solutions are inus
exhaustiveness:	the exhaustiveness scores
faithfulness:	the faithfulness scores
coherence:	the coherence scores
redundant:	whether the csf contain redundant proper parts

The latter five measures are optional and will be appended to the table according to the setting of the argument details.

References

Lam, Wai Fung, and Elinor Ostrom. 2010. "Analyzing the Dynamic Complexity of Development Interventions: Lessons from an Irrigation Experiment in Nepal." *Policy Sciences* 43 (2):1-25.

See Also

cna, truthTab, condition, minimalizeCsf, d.irrigate

Examples

```
# Crisp-set data from Lam and Ostrom (2010) on the impact of development interventions
# ------------------------------------------------------------------------------------
# CNA with causal ordering that corresponds to the ordering in Lam & Ostrom (2010); coverage
# cut-off at 0.9 (consistency cut-off at 1).
cna.irrigate <- cna(d.irrigate, ordering = list(c("A","R","F","L","C"),"W"), cov = .9,
                    maxstep = c(4, 4, 12), details = TRUE)
```

```
cna.irrigate

# The previous function call yields a total of 12 complex solution formulas, only
# 5 of which are returned in the default output.
# Here is how to extract all 12 complex solution formulas along with all
# solution attributes.
csf(cna.irrigate)
# With only the standard attributes plus exhaustiveness and faithfulness.
csf(cna.irrigate, details = c("e", "f"))

# Extract all atomic solution formulas.
asf(cna.irrigate)

# Extract all minimally sufficient conditions.
msc(cna.irrigate)

# Extract only the conditions (solutions).
csf(cna.irrigate)$condition
asf(cna.irrigate)$condition
msc(cna.irrigate)$condition

# A CNA of d.irrigate without a presupposed ordering is even more ambiguous.
cna2.irrigate <- cna(d.irrigate, cov = .9, maxstep = c(4,4,12), details = TRUE)

# To speed up the construction of complex solution formulas, first extract atomic solutions
# and then pass these asf to csf.
cna2.irrigate.asf <- asf(cna2.irrigate)
# By default, at least 20 csf are generated.
csf(cna2.irrigate, asfx = cna2.irrigate.asf, details = FALSE)
# Generate the first 191 csf.
csf(cna2.irrigate, asfx = cna2.irrigate.asf, 191, details = FALSE)
# Also extract exhaustiveness scores.
csf(cna2.irrigate, asfx = cna2.irrigate.asf, 191, details = "e")
# Generate all 684 csf.
csf(cna2.irrigate, asfx = cna2.irrigate.asf, 684)

# Return solution attributes with 5 digits.
print(cna2.irrigate.asf, digits = 5)

# Another example to the same effect.
print(csf(cna(d.irrigate, ordering = list(c("A","R","F","L","C"),"W"),
              maxstep = c(4, 4, 12), cov = 0.9)), digits = 5)

# Feed the outputs of msc, asf, and csf into the condition function to further inspect the
# properties of minimally sufficient conditions and atomic and complex solution formulas.
condition(msc(cna.irrigate)$condition, d.irrigate)
condition(asf(cna.irrigate)$condition, d.irrigate)
condition(csf(cna.irrigate)$condition, d.irrigate)

# Reshape the output of the condition function in such a way as to make it identical to the
# output returned by msc, asf, and csf.
as.condTbl(condition(msc(cna.irrigate)$condition, d.irrigate))
as.condTbl(condition(asf(cna.irrigate)$condition, d.irrigate))
as.condTbl(condition(csf(cna.irrigate)$condition, d.irrigate))

condTbl(csf(cna.irrigate)$condition, d.irrigate) # Same as preceding line
```

`cyclic` *Detect cyclic substructures in complex solution formulas (csf)*

Description

Given a character vector `x` specifying complex solution formula(s) (csf), `cyclic(x)` checks whether `x` contains cyclic substructures. The function can be used, for instance, to filter cyclic causal models out of `cna` solution objects (e.g. in order to reduce ambiguities).

Usage

```
cyclic(x, cycle.type = c("factor", "value"), use.names = TRUE, verbose = FALSE)
```

Arguments

`x`	Character vector specifying one or several csf.
`cycle.type`	Character string specifying what type of cycles to be detected: `"factor"` (the default) or `"value"`.
`use.names`	Logical; if `TRUE`, names are added to the result (see examples).
`verbose`	Logical; if `TRUE`, the checked causal paths are printed to the console.

Details

Detecting causal cycles is one of the most challenging tasks in causal data analysis—in all methodological traditions. In a nutshell, the reason is that factors in a cyclic structure are so highly interdependent that, even under optimal discovery conditions, the diversity of (observational) data tends to be too limited to draw informative conclusions about the data-generating structure. In consequence, various methods (most notably, Bayes nets methods, cf. Spirtes et al. 2000) assume that data-generating structures are acyclic.

`cna` outputs cyclic complex solutions formulas (csf) if they fit the data. Typically, however, the causal modeling of configurational data that can be modeled in terms of cycles is massively ambiguous. Therefore, if there are independent reasons to assume that the data are not generated by a cyclic structure, the function `cyclic` can be used to reduce the ambiguities in a `cna` output by filtering out all csf with cyclic substructures.

A causal structure has a cyclic substructure if, and only if, it contains a directed causal path from at least one cause back to itself. A regularity theory of causation in the vein of Mackie (1974) spells this criterion out as follows: a csf `x` has a cyclic substructure if, and only if, `x` contains a sequence <Z1, Z2,..., Zn> every element of which is an INUS condition of its successor and Z1=Zn. Accordingly, the function `cyclic` searches for sequences <Z1, Z2,..., Zn> of factors or factor values in a csf `x` such that (i) every Zi is contained in the antecedent (i.e. the left-hand side of "<->") of and atomic solution formula (asf) of Zi+1 in `x`, and (ii) Zn is identical to Z1. The function returns `TRUE` if, and only if, at least one such sequence (i.e. directed causal path) is found in `x`.

The `cycle.type` argument controls whether the sequence <Z1, Z2,..., Zn> is composed of factors (`cycle.type = "factor"`) or factor values (`cycle.type = "value"`). To illustrate, if `cycle.type = "factor"`, the following csf is considered cyclic: (A + B <-> C)*(c + D <-> A). The factor A (with value 1) appears in the antecedent of an asf of C (with value 1), and the factor C (with value 0) appears in the antecedent of an asf of A (with value 1). But if `cycle.type = "value"`, that same csf does not pass as cyclic. Although the factor value 1 of A appears in the antecedent of an asf of the factor value 1 of C, that same value of C does not appear in the antecedent of an asf of A; rather, the value 0 of C appears in the antecedent of A.

If `verbose = TRUE`, the sequences (paths) tested for cyclicity are output to the console. Note that the search for cycles is stopped as soon as one cyclic sequence (path) has been detected. Accordingly, not all sequences (paths) contained in `x` may be output to the console.

Value

A logical vector: `TRUE` for a csf with at least one cyclic substructure, `FALSE` for a csf without any cyclic substructures.

References

Mackie, John L. 1974. *The Cement of the Universe: A Study of Causation*. Oxford: Oxford University Press.

Spirtes, Peter, Clark Glymour, and Richard Scheines. 2000. *Causation, Prediction, and Search* (second ed.). Cambridge MA: MIT Press.

Examples

```
# cna infers two csf from the d.educate data, neither of which has a cyclic
# substructure.
cnaedu <- cna(d.educate)
cyclic(csf(cnaedu)$condition)

# At con = .82 and cov = .82, cna infers 605 csf from the d.jobsecurity data, all
# of which are cyclic.
cnajob <- fscna(d.jobsecurity, con = 0.82, cov = 0.82)
cyclic(csf(cnajob)$condition)  # first 20 csf
any(!cyclic(csf(cnajob, Inf)$condition))  # No acyclic csf!

# At con = .82 and cov = .82, cna infers 126 csf for the d.pacts data, some
# of which are cyclic, others are acyclic. If there are independent
# reasons to assume acyclicity, here is how to extract all acyclic csf.
cnapacts <- fscna(d.pacts, con = .82, cov = .82, inus.only = TRUE)
subset(csf(cnapacts, Inf), !cyclic(csf(cnapacts, Inf)$condition))

# With verbose = TRUE, the tested sequences (causal paths) are printed.
cyclic("(L=1 + G=1 <-> E=2)*(U=5 + D=3 <-> L=1)*(E=2*G=4 <-> D=3)", verbose = TRUE)
cyclic("(e*G + F*D + E*c*g*f <-> A)*(d + f*e + c*a <-> B)*(A*e + G*a*f <-> C)", verbose = TRUE)

# Argument cycle.type = "factor" or "value".
cyclic("(A*b + C -> D)*(d + E <-> A)")
cyclic("(A*b + C -> D)*(d + E <-> A)", cycle.type = "value")
cyclic("(L=1 + G=1 <-> E=2)*(U=5 + D=3 <-> L=2)*(E=2 + G=3 <-> D=3)")
cyclic("(L=1 + G=1 <-> E=2)*(U=5 + D=3 <-> L=2)*(E=2 + G=3 <-> D=3)", cycle.type = "v")

cyclic("a <-> A")
cyclic("a <-> A", cycle.type = "v")

sol1 <- "(A*X1 + Y1 <-> B)*(b*X2 + Y2 <-> C)*(C*X3 + Y3 <-> A)"
cyclic(sol1)
cyclic(sol1, cycle.type = "value")

sol2 <- "(A*X1 + Y1 <-> B)*(B*X2 + Y2 <-> C)*(C*X3 + Y3 <-> A)"
cyclic(sol2)
cyclic(sol2, cycle.type = "value")

# Argument use.names.
cyclic("a*b + C -> A", use.names = FALSE)

# More examples.
cyclic("(L + G <-> E)*(U + D <-> L)*(A <-> U)")
cyclic("(L + G <-> E)*(U + D <-> L)*(A <-> U)*(B <-> G)")
cyclic("(L + G <-> E)*(U + D <-> L)*(A <-> U)*(B <-> G)*(L <-> G)")
cyclic("(L + G <-> E)*(U + D <-> L)*(A <-> U)*(B <-> G)*(L <-> C)")
cyclic("(D -> A)*(A -> B)*(A -> C)*(B -> C)")

cyclic("(B + d*f <-> A)*(E + F*g <-> B)*(G*e + D*A <-> C)")
cyclic("(B*E + d*f <-> A)*(A + E*g + f <-> B)*(G*e + D*A <-> C)")
cyclic("(B + d*f <-> A)*(C + F*g <-> B)*(G*e + D*A <-> C)")
cyclic("(e*G + F*D + E*c*g*f <-> A)*(d + f*e + c*a <-> B)*(A*e + G*a*f <-> C)")
cyclic("(e*G + F*D + E*c*g*f <-> A)*(d + f*e + c*a <-> B)*(A*e + G*a*f <-> C)",
       verbose = TRUE)
```

d.autonomy — *Emergence and endurance of autonomy of biodiversity institutions in Costa Rica*

Description

This dataset is from Basurto (2013), who analyzes the causes of the emergence and endurance of autonomy among local institutions for biodiversity conservation in Costa Rica between 1986 and 2006.

Usage

```
d.autonomy
```

Format

The data frame contains 30 rows (cases), which are divided in two halfs: rows 1 to 14 comprise data on the emergence of local autonomy between 1986 and 1998, rows 15 to 30 comprise data on the endurance of local autonomy between 1998 and 2006. The data has the following 9 columns featuring fuzzy-set factors:

[, 1]	**AU**	local autonomy (ultimate outcome)
[, 2]	**EM**	local communal involvement through direct employment
[, 3]	**SP**	local direct spending
[, 4]	**CO**	co-management with local or regional stakeholders
[, 5]	**CI**	degree of influence of national civil service policies
[, 6]	**PO**	national participation in policy-making
[, 7]	**RE**	research-oriented partnerships
[, 8]	**CN**	conservation-oriented partnerships
[, 9]	**DE**	direct support by development organizations

Contributors

Thiem, Alrik: collection, documentation

Source

Basurto, Xavier. 2013. “Linking Multi-Level Governance to Local Common-Pool Resource Theory using Fuzzy-Set Qualitative Comparative Analysis: Insights from Twenty Years of Biodiversity Conservation in Costa Rica.” *Global Environmental Change* **23** (3):573-87.

`d.educate` *Artifical data on education levels and left-party strength*

Description

This artifical dataset of macrosociological factors on high levels of education is from Baumgartner (2009).

Usage

```
d.educate
```

Format

The data frame contains 8 rows (cases) and the following 5 columns featuring Boolean factors taking values 1 and 0 only:

[, 1]	**U**	existence of strong unions
[, 2]	**D**	high level of disparity
[, 3]	**L**	existence of strong left parties
[, 4]	**G**	high gross national product
[, 5]	**E**	high level of education

Source

Baumgartner, Michael. 2009. “Inferring Causal Complexity.” *Sociological Methods & Research* 38(1):71-101.

d.irrigate — *Data on the impact of development interventions on water adequacy in Nepal*

Description

This dataset is from Lam and Ostrom (2010), who analyze the effects of an irrigation experiment in Nepal.

Usage

```
d.irrigate
```

Format

The dataset contains 15 rows (cases) and the following 6 columns featuring Boolean factors taking values 1 and 0 only:

[, 1]	**A**	continual assistance on infrastructure improvement
[, 2]	**R**	existence of a set of formal rules for irrigation operation and maintenance
[, 3]	**F**	existence of provisions of fines
[, 4]	**L**	existence of consistent leadership
[, 5]	**C**	existence of collective action among farmers for system maintenance
[, 6]	**W**	persistent improvement in water adequacy at the tail end in winter

Source

Lam, Wai Fung, and Elinor Ostrom. 2010. “Analyzing the Dynamic Complexity of Development Interventions: Lessons from an Irrigation Experiment in Nepal.” *Policy Sciences* 43 (2):1-25.

d.jobsecurity — *Job security regulations in western democracies*

Description

This dataset is from Emmenegger (2011), who analyzes the determinants of high job security regulations in Western democracies using fsQCA.

Usage

```
d.jobsecurity
```

Format

The data frame contains 19 rows (cases) and the following 7 columns featuring fuzzy-set factors:

[, 1]	**S**	statism	("1" high, "0" not high)
[, 2]	**C**	non-market coordination	("1" high, "0" not high)
[, 3]	**L**	labour movement strength	("1" high, "0" not high)
[, 4]	**R**	Catholicism	("1" high, "0" not high)
[, 5]	**P**	religious party strength	("1" high, "0" not high)
[, 6]	**V**	institutional veto points	("1" many, "0" not many)
[, 7]	**JSR**	job security regulations	("1" high, "0" not high)

Contributors

Thiem, Alrik: collection, documentation

Note

The row names are the official International Organization for Standardization (ISO) country code elements as specified in ISO 3166-1-alpha-2.

Source

Emmenegger, Patrick. 2011. “Job Security Regulations in Western Democracies: A Fuzzy Set Analysis.” *European Journal of Political Research* 50(3):336-64.

`d.minaret` *Data on the voting outcome of the 2009 Swiss Minaret Initiative*

Description

This dataset is from Baumgartner and Epple (2014), who analyze the determinants of the outcome of the vote on the 2009 Swiss Minaret Initative.

Usage

```
d.minaret
```

Format

The data frame contains 26 rows (cases) and the following 6 columns featuring raw data:

[, 1]	**A**	rate of old xenophobia
[, 2]	**L**	left party strength
[, 3]	**S**	share of native speakers of Serbian, Croatian, or Albanian
[, 4]	**T**	strength of traditional economic sector
[, 5]	**X**	rate of new xenophobia
[, 6]	**M**	acceptance of Minaret Initiative

Contributors

Ruedi Epple: collection, documentation

Source

Baumgartner, Michael, and Ruedi Epple. 2014. "A Coincidence Analysis of a Causal Chain: The Swiss Minaret Vote." *Sociological Methods & Research* 43 (2):280-312.

---

d.pacts — *Data on the emergence of labor agreements in new democracies between 1994 and 2004*

Description

This dataset is from Aleman (2009), who analyzes the causes of the emergence of tripartite labor agreements among unions, employers, and government representatives in new democracies in Europe, Latin America, Africa, and Asia between 1994 and 2004.

Usage

```
d.pacts
```

Format

The data frame contains 78 rows (cases) and the following 5 columns listing membership scores in 5 fuzzy sets:

[, 1]	**PACT**	development of tripartite cooperation (ultimate outcome)
[, 2]	**W**	regulation of the wage setting process
[, 3]	**E**	regulation of the employment process
[, 4]	**L**	presence of a left government
[, 5]	**P**	presence of an encompassing labor organization (labor power)

Contributors

Thiem, Alrik: collection, documentation

Source

Aleman, Jose. 2009. "The Politics of Tripartite Cooperation in New Democracies: A Multi-level Analysis." *International Political Science Review* 30 (2):141-162.

---

d.pban — *Party ban provisions in sub-Saharan Africa*

Description

This dataset is from Hartmann and Kemmerzell (2010), who, among other things, analyze the causes of the emergence of party ban provisions in sub-Saharan Africa.

Usage

```
d.pban
```

Format

The data frame contains 48 rows (cases) and the following 5 columns, some of which feature multi-value factors:

[, 1]	**C**	colonial background ("2" British, "1" French, "0" other)
[, 2]	**F**	former regime type competition ("2" no, "1" limited, "0" multi-party)
[, 3]	**T**	transition mode ("2" managed, "1" pacted, "0" democracy before 1990)
[, 4]	**V**	ethnic violence ("1" yes, "0" no)
[, 5]	**PB**	introduction of party ban provisions ("1" yes, "0" no)

Source

Hartmann, Christof, and Joerg Kemmerzell. 2010. “Understanding Variations in Party Bans in Africa.” *Democratization* 17(4):642-65. DOI: 10.1080/13510347.2010.491189.

`d.performance` *Data on combinations of industry, corporate, and business-unit effects*

Description

This dataset is from Greckhammer et al. (2008), who analyze the causal conditions for superior (above average) business-unit performance of corporations in the manufacturing sector during the years 1995 to 1998.

Usage

```
d.performance
```

Format

The data frame contains 214 rows featuring configurations, one column reporting the frequencies of each configuration, and 8 columns listing the following Boolean factors:

[, 1]	**MU**	above average industry munificence
[, 2]	**DY**	high industry dynamism
[, 3]	**CO**	high industry competitiveness
[, 4]	**DIV**	high corporate diversification
[, 5]	**CRA**	above median corporate resource availability
[, 6]	**CI**	above median corporate capital intensity
[, 7]	**BUS**	large business-unit size
[, 8]	**SP**	above average business-unit performance (in the manufacturing sector)

Source

Greckhamer, Thomas, Vilmos F. Misangyi, Heather Elms, and Rodney Lacey. 2008. “Using Qualitative Comparative Analysis in Strategic Management Research: An Examination of Combinations of Industry, Corporate, and Business-Unit Effects.” *Organizational Research Methods* 11 (4):695-726.

`d.volatile` *Data on the volatility of grassroots associations in Norway between 1980 and 2000*

Description

This dataset is from Wollebaek (2010), who analyzes the causes of disbandings of grassroots associations in Norway.

Usage

```
d.volatile
```

Format

The data frame contains 22 rows (cases) and the following 9 columns featuring Boolean factors taking values 1 and 0 only:

[, 1]	**PG**	high population growth
[, 2]	**RB**	high rurbanization (i.e. people moving to previously sparsely populated areas that are not adjacent to a larger city)
[, 3]	**EL**	high increase in education levels
[, 4]	**SE**	high degree of secularization
[, 5]	**CS**	existence of Christian strongholds
[, 6]	**OD**	high organizational density
[, 7]	**PC**	existence of polycephality (i.e. municipalities with multiple centers)
[, 8]	**UP**	urban proximity
[, 9]	**VO2**	very high volatility of grassroots associations

Source

Wollebaek, Dag. 2010. “Volatility and Growth in Populations of Rural Associations.” *Rural Sociology* 75:144-166.

`d.women` *Data on high percentage of women’s represention in parliaments of western countries*

Description

This dataset is from Krook (2010), who analyzes the causal conditions for high women's representation in western-democratic parliaments.

Usage

```
d.women
```

Format

The data frame contains 22 rows (cases) and the following 6 columns featuring Boolean factors taking values 1 and 0 only:

[, 1]	**ES**	existence of a PR electoral system
[, 2]	**QU**	existence of quotas for women
[, 3]	**WS**	existence of social-democratic welfare system
[, 4]	**WM**	existence of autonomous women's movement
[, 5]	**LP**	strong left parties
[, 6]	**WNP**	high women's representation in parliament

Source

Krook, Mona Lena. 2010. "Women's Representation in Parliament: A Qualitative Comparative Analysis." *Political Studies* 58 (5):886-908.

`full.tt` *Generate all logically possible value configurations of a given set of factors*

Description

`full.tt` generates a `truthTab` with all logically possible value configurations of the factors defined in the input `x`. It is more flexible than `allCombs`. `x` can be a `truthTab`, a data frame, an integer, a list specifying the factors' value ranges, or a character vector featuring all admissible factor values.

Usage

```
full.tt(x, ...)

## Default S3 method:
full.tt(x, type = c("cs", "mv", "fs"), ...)
## S3 method for class 'truthTab'
full.tt(x, ...)
## S3 method for class 'tti'
full.tt(x, ...)
```

Arguments

`x`	A `truthTab`, a data frame, an integer, a list specifying the factors' value ranges, or a character vector featuring all admissible factor values (see the details and examples below).
`type`	Character vector specifying the type of `x`: `"cs"` (crisp-set), `"mv"` (multi-value), or `"fs"` (fuzzy-set); passed to `truthTab`; only required if `x` is a data frame or matrix.
`...`	Further arguments passed to methods.

Details

`full.tt` generates all logically possible value configurations of the factors defined in `x`, which can either be a character vector or an integer or a list or a data frame or a matrix.

- If `x` is a character vector, it can be a condition of any of the three types of conditions, *boolean*, *atomic* or *complex* (see `condition`). `x` must contain at least one factor. Factor names and admissible values are guessed from the Boolean formulas. If `x` contains multi-value factors, only those values are considered admissible that are explicitly contained in `x`. Accordingly, in case of multi-value factors, `full.tt` should be given the relevant factor definitions by means of a list (see below).
- If `x` is an integer and `<=26`, the output will be a full truth table of type `"cs"` with `x` factors. The first `x` capital letters of the alphabet will be used as the names of the factors.
- If `x` is a list, `x` is expected to have named elements each of which provides the factor names with corresponding vectors enumerating their admissible values (i.e. their value ranges). These values must be integers.
- If `x` is a `truthTab`, data frame, or matrix, `colnames(x)` are interpreted as factor names and the rows as enumerating the admissible values (i.e. as value ranges). If `x` is a data frame or a matrix, `x` is first converted to a `truthTab` (the function `truthTab` is called with `type` as specified in `full.tt`), and the `truthTab` method of `full.tt` is then applied to the result. The `truthTab` method uses all factors and factor values occurring in the `truthTab`. If `x` is of type `"fs"`, 0 and 1 are taken as the admissible values.

In combination with `selectCases`, `full.tt` is useful for simulating data, which are needed for inverse search trials benchmarking the output of `cna`. While `full.tt` generates the space of all logically possible configurations of the factors in an analyzed factor set, `selectCases` selects those configurations from this space that are compatible with a given data-generating causal structure (i.e. the ground truth), that is, it selects the empirically possible configurations.

The method for class "tti" is for internal use only.

Value

A `truthTab` of type `"cs"` or `"mv"` with the full enumeration of combinations of the factor values.

See Also

`truthTab`, `selectCases`, `allCombs`

Examples

```
# x is a character vector.
full.tt("A + B*c")
full.tt("A=1*C=3 + B=2*C=1 + A=3*B=1")
full.tt(c("A + b*C", "a*D"))
full.tt("!A*-(B + c) + F")

# x is a data frame.
full.tt(d.educate)
full.tt(d.jobsecurity, type = "fs")
full.tt(d.pban, type = "mv")

# x is a truthTab.
full.tt(cstt(d.educate))
full.tt(fstt(d.jobsecurity))
full.tt(mvtt(d.pban))

# x is an integer.
full.tt(6)

# x is a list.
full.tt(list(A = 0:1, B = 0:1, C = 0:1))  # cs
full.tt(list(A = 1:2, B = 0:1, C = 1:4))  # mv

# Simulating crisp-set data.
groundTruth.1 <- "(A*b + C*d <-> E)*(E*H + I*k <-> F)"
fullData <- full.tt(groundTruth.1)
idealData <- selectCases(groundTruth.1, fullData)
# Introduce 20% data fragmentation.
fragData <- idealData[-sample(1:nrow(idealData), nrow(idealData)*0.2), ]
# Introduce 10% random noise.
realData <- rbind(tt2df(fullData[sample(1:nrow(fullData), nrow(fragData)*0.1), ]), fragData)

# Simulating multi-value data.
groundTruth.2 <- "(JO=3 + TS=1*PE=3 <-> ES=1)*(ES=1*HI=4 + IQ=2*KT=5 <-> FA=1)"
fullData <- full.tt(list(JO=1:3, TS=1:2, PE=1:3, ES=1:2, HI=1:4, IQ=1:5, KT=1:5, FA=1:2))
idealData <- selectCases(groundTruth.2, fullData)
# Introduce 20% data fragmentation.
fragData <- idealData[-sample(1:nrow(idealData), nrow(idealData)*0.2), ]
# Introduce 10% random noise.
realData <- rbind(tt2df(fullData[sample(1:nrow(fullData), nrow(fragData)*0.1), ]), fragData)
```

`is.inus` *Test disjunctive normal forms for logical redundancies*

Description

`is.inus` checks for each element of a character vector specifying Boolean disjunctive normal forms (DNFs) whether it amounts to a minimally necessary disjunction of minimally sufficient conditions relative to all logically possible configurations of the factors contained in the DNF.

Usage

```
is.inus(cond, x = NULL)
```

Arguments

`cond`	Character vector specifying Boolean disjunctive normal forms (DNFs). Currently the permissible syntax is restricted to the operators `+`, `*` and `=` (in case of DNFs of type `"mv"`), with negation being expressed by lower case letters.
`x`	An optional argument providing a `truthTab`, a data frame, or a list specifying the factors' value ranges if `cond` contains multi-value factors; if `x` is not `NULL`, `is.inus` tests whether `cond` is redundancy-free relative to `full.tt(x)`, otherwise relative to `full.tt(cond)`.

Details

According to the regularity theory of causation underlying CNA, a Boolean dependency structure is causally interpretable only if it does not contain any redundant elements. Boolean dependency structures may feature various types of redundancies (Baumgartner and Falk 2018): redundancies in necessary and sufficient conditions or structural redundancies. Redundancies may obtain relative to an analyzed set of empirical data, which, typically, are fragmented and do not feature all logically possible configurations, or they may obtain for principled logical reasons, that is, relative to all configurations that are possible according to classical Boolean logic. While the function `cna` builds redundancy-free Boolean dependency structures based on empirical data, the function `is.inus` tests necessary and sufficient conditions for logical redundancies (`redundant` performs an analogous test for structural redundancies).

`is.inus` takes a character vector `cond` specifying Boolean disjunctive normal forms (DNFs) as input and checks whether these DNFs are redundancy-free according to Boolean logic, that is, minimally necessary disjunctions of minimally sufficient conditions. A necessary disjunction is *minimal* if, and only if, no proper sub-disjunction of it is necessary; and a sufficient conjunction is *minimal* if, and only if, no proper sub-conjunction of it is sufficient (Grasshoff and May 2001). In the function's default call with `x = NULL`, this minimality test is performed relative to `full.tt(cond)`; if `x` is not `NULL`, the test is performed relative to `full.tt(x)`. As `full.tt(cond)` and `full.tt(x)` coincide in case of binary factors, the argument `x` has no effect in the crisp-set and fuzzy-set cases and, hence, does not have to be specified. In case of multi-value factors, however, the argument `x` should be specified in order to define the factors' value ranges (see details below).

A `cond` with `is.inus(cond)==FALSE` can be freed of logical redundancies by means of the `minimalize` function.

Value

Logical vector of the same length as `cond`.

References

Baumgartner, Michael and Christoph Falk. 2018. “Boolean Difference-Making: A Modern Regularity Theory of Causation”. *PhilSci Archive*. url: http://philsciarchive.pitt.edu/id/eprint/14876.

Grasshoff, Gerd and Michael May. 2001. “Causal Regularities.” In W Spohn, M Ledwig, M Esfeld (eds.), *Current Issues in Causation*, pp. 85-114. Mentis, Paderborn.

See Also

condition, full.tt, redundant, minimalize, cna

Examples

```
# Crisp-set case
# --------------
is.inus(c("A", "A + B", "A + a*B", "A + a", "A*a"))

is.inus("F + f*G")
is.inus("F*G + f*H + G*H")
is.inus("F*G + f*g + H*F + H*G")


# Multi-value case
# ----------------
mvdata <- mvtt(setNames(allCombs(c(2, 3, 2)) -1, c("C", "F", "V")))
is.inus("C=1 + F=2*V=0", mvdata)
is.inus("C=1 + F=2*V=0", list(C=0:1, F=0:2, V=0:1))
# When x is NULL, is.inus is applied to full.tt("C=1 + F=2*V=0"), which has only
# one single row. That row is then interpreted to be the only possible configuration,
# in which case C=1 + F=2*V=0 is tautologous and, hence, non-minimal.
is.inus("C=1 + F=2*V=0")

is.inus("C=1 + C=0*C=2", mvtt(d.pban))    # contradictory
is.inus("C=0 + C=1 + C=2", mvtt(d.pban))  # tautologous


# Fuzzy-set case
# --------------
fsdata <- fstt(d.jobsecurity)
conds <- csf(cna(fsdata, con = 0.85, cov = 0.85))$condition
conds <- cna:::lhs(conds)
is.inus(conds, fsdata)
is.inus(c("S + s", "S + s*R", "S*s"), fsdata)
```

`is.submodel` *Identify correctness-preserving submodel relations*

Description

`is.submodel` checks for each element of a vector of `cna` solution formulas whether it is a submodel of a specified target model `y`. If `y` is the true model in an inverse search (i.e. the ground truth), `is.submodel` identifies the correct models in the `cna` output (see Baumgartner and Thiem 2017, Baumgartner and Ambuehl 2018).

Usage

```
is.submodel(x, y, strict = FALSE)
identical.model(x, y)
```

Arguments

`x`	Character vector of atomic and/or complex solution formulas (asf/csf). Must be of length 1 in `identical.model`.
`y`	Character string of length 1 specifying the target asf or csf.
`strict`	Logical; if `TRUE`, the elements of `x` only count as submodels of `y` if they are proper parts of `y` (i.e. not identical to `y`).

Details

To benchmark the reliability of a method of causal inference it must be tested to what degree the method recovers the true data-generating structure Δ or proper substructures of Δ from data of varying quality. Reliability benchmarking is done in so-called *inverse searches*, which reverse the order of causal discovery as normally conducted in scientific practice. An inverse search comprises three steps: (1) a causal structure Δ is drawn/presupposed (as ground truth), (2) artificial data δ is simulated from Δ, possibly featuring various deficiencies (e.g. noise, limited diversity, measurement error etc.), and (3) δ is processed by the benchmarked method in order to check whether its output meets the tested reliability benchmark (e.g. whether the output is true of or identical to Δ).

The main purpose of `is.submodel` is to execute step (3) of an inverse search that is tailor-made to test the reliability of `cna` [with `randomConds` and `selectCases` designed for steps (1) and (2), respectively]. A solution formula `x` being a submodel of a target formula `y` means that all the causal claims entailed by `x` are true of `y`, which is the case if a causal interpretation of `x` entails conjunctive and disjunctive causal relevance relations that are all likewise entailed by a causal interpretation of `y`. More specifically, `x` is a submodel of `y` if, and only if, the following conditions are satisfied: (i) all factor values causally relevant according to `x` are also causally relevant according to `y`, (ii) all factor values contained in two different disjuncts in `x` are also contained in two different disjuncts in `y`, (iii) all factor values contained in the same conjunct in `x` are also contained in the same conjunct in `y`, and (iv) if `x` is a csf with more than one asf, (i) to (iii) are satisfied for all asfs in `x`. For more details see Baumgartner and Thiem (2017) or Baumgartner and Ambuehl (2018, online appendix).

`is.submodel` requires two inputs `x` and `y`, where `x` is a character vector of `cna` solution formulas (asf or csf) and `y` is one asf or csf (i.e. a character string of length 1), viz. the target structure or ground truth. The function returns `TRUE` for elements of `x` that are a submodel of `y` according to the definition of submodel-hood given in the previous paragraph. If `strict = TRUE`, `x` counts as a submodel of `y` only if `x` is a proper part of `y` (i.e. `x` is not identical to `y`).

The function `identical.model` returns `TRUE` only if `x` (which must be of length 1) and `y` are identical. It can be used to test whether `y` is completely recovered in an inverse search.

Value

Logical vector of the same length as `x`.

References

Baumgartner, Michael and Mathias Ambuehl. 2018. "Causal Modeling with Multi-Value and Fuzzy-Set Coincidence Analysis." *Political Science Research and Methods*. doi:10.1017/psrm.2018.45.

Baumgartner, Michael and Alrik Thiem. 2017. "Often Trusted But Never (Properly) Tested: Evaluating Qualitative Comparative Analysis". *Sociological Methods & Research*. doi: 10.1177/0049124117701487.

See Also

randomConds, selectCases, cna.

Examples

```
# Binary expressions
# ------------------
trueModel.1 <- "(A*b + a*B <-> C)*(C*d + c*D <-> E)"
candidates.1 <- c("(A + B <-> C)*(C + c*D <-> E)", "(A + B <-> C)",
                 "(A <->  C)*(C <-> E)", "(C <-> E)")
candidates.2 <- c("(A*B + a*b <-> C)*(C*d + c*D <-> E)", "(A*b*D + a*B <-> C)",
                 "(A*b + a*B <-> C)*(C*A*D <-> E)", "(D <-> C)",
                 "(A*b + a*B + E <-> C)*(C*d + c*D <-> E)")

is.submodel(candidates.1, trueModel.1)
is.submodel(candidates.2, trueModel.1)
is.submodel(c(candidates.1, candidates.2), trueModel.1)

is.submodel("C + b*A <-> D", "A*b + C <-> D")
is.submodel("C + b*A <-> D", "A*b + C <-> D", strict = TRUE)
identical.model("C + b*A <-> D", "A*b + C <-> D")

target.1 <- "(A*b + a*B <-> C)*(C*d + c*D <-> E)"
testformula.1 <- "(A*b + a*B <-> C)*(C*d + c*D <-> E)*(A + B <-> C)"
is.submodel(testformula.1, target.1)

# Multi-value expressions
# -----------------------
trueModel.2 <- "(A=1*B=2 + B=3*A=2 <-> C=3)*(C=1 + D=3 <-> E=2)"
is.submodel("(A=1*B=2 + B=3 <-> C=3)*(D=3 <-> E=2)", trueModel.2)
is.submodel("(A=1*B=1 + B=3 <-> C=3)*(D=3 <-> E=2)", trueModel.2)
is.submodel(trueModel.2, trueModel.2)
is.submodel(trueModel.2, trueModel.2, strict = TRUE)

target.2 <- "C=2*D=1*B=3 + A=1 <-> E=5"
testformula.2 <- c("C=2 + D=1 <-> E=5","C=2 + D=1*B=3 <-> E=5","A=1+B=3*D=1*C=2 <-> E=5",
                "C=2 + D=1*B=3 + A=1 <-> E=5","C=2*B=3 + D=1 + B=3 + A=1 <-> E=5")
is.submodel(testformula.2, target.2)
identical.model(testformula.2[3], target.2)
identical.model(testformula.2[1], target.2)
```

`makeFuzzy` *Generate fuzzy-set data by simulating noise*

Description

Generates fuzzy-set data by simulating the addition of random noise from the uncontrolled causal background to a data frame featuring binary factors only.

Usage

```
makeFuzzy(x, fuzzvalues = c(0, 0.05, 0.1), ...)
```

Arguments

`x`	Data frame or `truthTab` featuring binary factors with values 1 and 0 only.
`fuzzvalues`	Values to be added to the 0's and subtracted from the 1's.
`...`	Additional arguments are passed to `truthTab`.

Details

In combination with `allCombs`, `full.tt` and `selectCases`, `makeFuzzy` is useful for simulating noisy data, which are needed for inverse search trials benchmarking the output of `cna`. `makeFuzzy` transforms a data frame or `truthTab` consisting of binary factors into a fuzzy-set `truthTab` by adding values selected at random from the argument `fuzzvalues` to the 0's and subtracting them from the 1's in the data frame. This transformation simulates the introduction of background noise into the data. `selectCases` can subsequently be applied to draw those fuzzy-set configurations from the resulting data that are compatible with a given data generating causal structure.

Value

A `truthTab` of type "fs".

See Also

`selectCases`, `allCombs`, `full.tt`, `truthTab`, `cna`, `tt2df`, `condition`

Examples

```
# Fuzzify a binary 6x3 matrix with default fuzzvalues.
X <- matrix(sample(0:1, 18, replace = TRUE), 6)
makeFuzzy(X)

# ... and with customized fuzzvalues.
makeFuzzy(X, fuzzvalues = 0:5/10)
makeFuzzy(X, fuzzvalues = seq(0, 0.45, 0.01))

# Generate all configurations of 5 fuzzy-set factors that are compatible with the causal
# structure A*b + C*D <-> E, such that con = .8 and cov = .8.
```

```
dat1 <- allCombs(c(2, 2, 2, 2, 2)) - 1
dat2 <- makeFuzzy(dat1, fuzzvalues = seq(0, 0.45, 0.01))
(dat3 <- selectCases1("A*b + C*D <-> E", con = .8, cov = .8, dat2))
condition("A*b + C*D <-> E", dat3)

# First, generate all configurations of 5 dichotomous factors that are compatible with
# the causal chain (A*b + a*B <-> C)*(C*d + c*D <-> E) and, second, introduce background
# noise.
dat1 <- full.tt(5)
dat2 <- selectCases("(A*b + a*B <-> C)*(C*d + c*D <-> E)", dat1)
(dat3 <- makeFuzzy(dat2, fuzzvalues = seq(0, 0.45, 0.01)))
condition("(A*b + a*B <-> C)*(C*d + c*D <-> E)", dat3)

# Inverse search for the data generating causal structure A*b + a*B + C*D <-> E from
# fuzzy-set data with non-perfect consistency and coverage scores.
dat1 <- full.tt(5)
set.seed(55)
dat2 <- makeFuzzy(dat1, fuzzvalues = 0:4/10)
dat3 <- selectCases1("A*b + a*B + C*D <-> E", con = .8, cov = .8, dat2)
fscna(dat3, ordering = list("E"), strict = TRUE, con = .8, cov = .8)
```

minimalize *Eliminate logical redundancies from Boolean expressions*

Description

minimalize eliminates logical redundancies from a Boolean expression cond based on all configurations of the factors in cond that are possible according to classical Boolean logic. That is, minimalize performs logical (i.e. not data-driven) redundancy elimination. The output is a set of redundancy-free DNFs that are logically equivalent to cond.

Usage

```
minimalize(cond, x = NULL, maxstep = c(4, 4, 12))
```

Arguments

cond	Character vector specifying Boolean expressions; the acceptable syntax is the same as that of condition.
x	A data frame, a truthTab, or a list determining the possible values for each factor in cond; x has no effect for a cond with only binary factors but is mandatory for a cond with multi-value factors (see details).
maxstep	Maximal complexity of the returned redundancy-free DNFs (see cna).

Details

The regularity theory of causation underlying CNA conceives of causes as parts of redundancy-free Boolean dependency structures. Boolean dependency structures tend to contain a host of redundancies. Redundancies may obtain relative to an analyzed set of empirical data, which, typically, are fragmented and do not feature all logically possible configurations, or they may obtain for principled logical reasons, that is, relative to all configurations that are possible according to Boolean logic. Whether a Boolean expression (in disjunctive normal form) contains the latter type of logical redundancies can be checked with the function `is.inus`. If `is.inus(cond)==FALSE`, `cond` contains logical redundancies.

`minimalize` eliminates logical redundancies from `cond` and outputs all redundancy-free disjunctive normal forms (DNF) (within some complexity range given by `maxstep`) that are logically equivalent with `cond`. If `cond` is redundancy-free, no reduction is possible and `minimalize` returns `cond` itself (possibly as an element of multiple logically equivalent redundancy-free DNFs). If `cond` is not redundancy-free, a `cna` with `con = 1` and `cov = 1` is performed relative to `full.tt(x)` (relative to `full.tt(cond)` if `x` is `NULL`). The output is the set of all redundancy-free DNFs in the complexity range given by `maxstep` that are logically equivalent to `cond`.

The purpose of the optional argument `x` is to determine the space of possible values of the factors in `cond`. If all factors in `cond` are binary, `x` is optional and without influence on the output of `minimalize`. If some factors in `cond` are multi-value, `minimalize` needs to be given the range of these values. `x` can be a data frame or `truthTab` listing all possible value configurations or simply a list of the possible values for each factor in `cond` (see examples).

The argument `maxstep`, which is identical to the corresponding argument in `cna`, specifies the maximal complexity of the returned DNF. `maxstep` expects a vector of three integers `c(i,j,k)` determining that the generated DNFs have maximally `j` disjuncts with maximally `i` conjuncts each and a total of maximally `k` factors. The default is `maxstep = c(4,4,12)`. If the complexity range of the search space given by `maxstep` is too low, it may happen that nothing is returned (accompanied by a corresponding warning message). In that case, the `maxstep` values need to be increased.

Value

A list of character vectors of the same length as `cond`. Each list element contains one or several redundancy-free disjunctive normal forms (DNFs) that are logically equivalent to `cond`.

See Also

`condition`, `is.inus`, `cna`, `full.tt`.

Examples

```
# Binary expressions
# ------------------
# DNFs as input.
minimalize(c("A", "A+B", "A + a*B", "A + a", "A*a"))
minimalize(c("F + f*G", "F*G + f*H + G*H", "F*G + f*g + H*F + H*G"))

# Any Boolean expressions (with variable syntax) are admissible inputs.
minimalize(c("!(A*B*C + a*b*c)", "A*!(B*d+E)->F", "-(A+-(E*F))<->H"))
```

```
# Proper redundancy elimination may require increasing the maxstep values.
minimalize("!(A*B*C*D*E+a*b*c*d*e)")
minimalize("!(A*B*C*D*E+a*b*c*d*e)", maxstep = c(3, 5, 15))


# Multi-value expressions
# -----------------------
# In case of expressions with multi-value factors, the relevant range of factor
# values must be specified by means of x. x can be a list or a truthTab:
values <- list(C = 0:3, F = 0:2, V = 0:4)
minimalize(c("C=1 + F=2*V=0", "C=1 + C=0*V=1"), values)
minimalize(c("C=1 + C=0 * C=2", "C=0 + C=1 + C=2"), mvtt(d.pban))


# Eliminating logical redundancies from non-inus asf inferred from real data
# --------------------------------------------------------------------------
fsdata <- fstt(d.jobsecurity)
conds <- asf(cna(fsdata, con = 0.8, cov = 0.8))$condition
conds <- cna:::lhs(conds)
noninus.conds <- conds[-which(is.inus(conds, fsdata))]
minimalize(noninus.conds)
```

minimalizeCsf *Eliminate structural redundancies from csf*

Description

minimalizeCsf eliminates structural redundancies from complex solutions formulas (csf) by recursively testing their component atomic solution formulas (asf) for redundancy and eliminating the redundant ones.

Usage

```
minimalizeCsf(x, ...)

## Default S3 method:
minimalizeCsf(x, data = full.tt(x), verbose = FALSE, ...)
## S3 method for class 'cna'
minimalizeCsf(x, n = 20, verbose = FALSE, ...)
## S3 method for class 'minimalizeCsf'
print(x, subset = 1:5, ...)
```

Arguments

x	In the default method, x is a character vector specifying csf. The cna method uses the strings representing the csf contained in an output object of cna (see details).
data	Data frame, matrix or truthTab with the data; optional if all factors in x are binary but required if some factors are multi-value.

`verbose`	Logical; if `TRUE` additional messages on the number of csf that are found to be reducible are printed.
`n`	Minimal number of csf to use.
`subset`	Integer vector specifying the numbers of csf to be displayed.
`...`	Further arguments passed to the methods.

Details

The core criterion that Boolean dependency structures must satisfy in order to be causally interpretable is *redundancy-freeness*. In atomic solution formulas (asf), both sufficient and necessary conditions are completely free of redundant elements. However, when asf are conjunctively combined to complex solution formulas (csf), new redundancies may arise. A csf may contain redundant parts. To illustrate, assume that a csf is composed of three asf: asf1 * asf2 * asf3. It can happen that the conjunction asf1 * asf2 * asf3 is logically equivalent to a proper part of itself, say, to asf1 * asf2. In that case, asf3 is a so-called *structural redundancy* in asf1 * asf2 * asf3 and must not be causally interpreted. See the **cna** package vignette or Baumgartner and Falk (2018) for more details.

`minimalizeCsf` recursively tests the `asf` contained in a `csf` for structural redundancies and eliminates the redundant ones. It takes a character vector `x` specifying csf as input and builds all redundancy-free csf that can be inferred from `x`. The function is especially useful in combination with `cna`, which builds csf by conjunctively concatenating asf. One of the `cna` solution attributes, which is accessed via `details = TRUE` or `details = "r"`, is called "redundant". If a csf output by `cna` has the attribute `redundant == TRUE`, that csf has at least one structurally redundant part. The `cna` function, however, does not identify those redundant parts. For this purpose, the `cna` object must be passed on to `minimalizeCsf`.

There are two possibilities to use `minimalizeCsf`. Either the csf to be tested for structural redundancies is passed to `minimalizeCsf` as a character vector (this is the default method), or `minimalizeCsf` is applied directly to the output of `cna`. In the latter case, the csf are extracted from the `cna`-object.

As a test for structural redundancies amounts to a test of logical equivalencies, it must be conducted relative to all logically possible configurations of the factors in `x`. That space of logical possibilities is generated by `full.tt(x)` if the `data` argument takes its default value. If all factors in `x` are binary, providing a non-default `data` value is optional and without influence on the output of `minimalizeCsf`. If some factors in `x` are multi-value, `minimalizeCsf` needs to be given the range of these values by means of the `data` argument. `data` can be a data frame or `truthTab` listing all possible value configurations.

Value

`minimalizeCsf` returns an object of class "minimalizeCsf", essentially a data frame.

Contributors

Falk, Christoph: identification and solution of the problem of structural redundancies

References

Baumgartner, Michael and Christoph Falk. 2018. “Boolean Difference-Making: A Modern Regularity Theory of Causation”. *PhilSci Archive*. url: http://philsciarchive.pitt.edu/id/eprint/14876.

See Also

csf, cna, redundant, full.tt.

Examples

```
# The default method.
minimalizeCsf("(f + a*D <-> C)*(C + A*B <-> D)*(c + a*E <-> F)")
minimalizeCsf("(f + a*D <-> C)*(C + A*B <-> D)*(c + a*E <-> F)",
              verbose = TRUE) # Same result, but with some messages.

# The cna method.
dat1 <- selectCases("(C + A*B <-> D)*(c + a*E <-> F)")
ana1 <- cna(dat1, details = c("r"))
csf(ana1)
# The attribute "redundant" taking the value TRUE in ana1 shows that this csf contains
# at least one redundant element. Only the application of minimalizeCsf() identifies
# the redundant element.
minimalizeCsf(ana1)

# Real data entailing a large number of csf with many redundancies.
tt.js <- fstt(d.jobsecurity)
cna.js <- cna(tt.js, con = .8, cov = .8)
minim100 <- minimalizeCsf(cna.js, n = 100) # may take a couple of seconds...
minim100  # By default the first 5 solutions are displayed.

# With mv data.
tt.pban <- mvtt(d.pban)
cna.pban <- cna(tt.pban, con = .75, cov = .75)
csf.pban <- csf(cna.pban, 100)
sol.pban <- csf.pban$condition

minim.pban <- minimalizeCsf(sol.pban, tt.pban)
as.character(minim.pban$condition)

# Alternatively, a more direct replication of the above using the cna method.
minim.pban <- minimalizeCsf(cna.pban, n = 100)
print(minim.pban, 1:50) # print the first 50 redundancy-free csf
```

`randomConds` *Generate random solution formulas*

Description

Based on a set of factors and a corresponding data type—given as a data frame or `truthTab`—, `randomAsf` generates a random atomic solution formula (asf) and `randomCsf` a random (acyclic) complex solution formula (csf).

Usage

```
randomAsf(x, outcome = NULL, compl = NULL, how = c("inus", "minimal"))
randomCsf(x, outcome = NULL, n.asf = NULL, compl = NULL)
```

Arguments

x	Data frame or truthTab; determines the number of factors, their names and their possible values.
outcome	Optional character vector (of length 1 in randomAsf) specifying the outcome factor(s) in the solution formula; must be a subset of names(x).
compl	Integer vector specifying the maximal complexity of the formula (i.e. number of factors in msc; number of msc in asf).
how	Character string, either "inus" or "minimal", specifying whether the generated solution formula is redundancy-free relative to full.tt(x) or relative to x (see details).
n.asf	Integer scalar specifying the number of asf in the csf. Is overridden by length(outcome) if outcome is not NULL. Note that n.asf is limited to ncol(x)-2.

Details

randomAsf and randomCsf can be used to randomly draw data generating structures (ground truths) in inverse search trials benchmarking the output of cna. In the regularity theoretic context in which the CNA method is embedded, a causal structure is a redundancy-free Boolean dependency structure. Hence, randomAsf and randomCsf both produce redundancy-free Boolean dependency structures. randomAsf generates structures with one outcome, i.e. atomic solution formulas (asf), randomCsf generates structures with multiple outcomes, i.e. complex solution formulas (csf), that are free of cyclic substructures. In a nutshell, randomAsf proceeds by, first, randomly drawing disjunctive normal forms (DNFs) and by, second, eliminating redundancies from these DNFs. randomCsf essentially consists in repeated executions of randomAsf.

The only mandatory argument of randomAsf and randomCsf is a data frame or a truthTab x defining the factors (with their possible values) from which the generated asf and csf shall be drawn. If asf and csf are built from multi-value or fuzzy-set factors, x must be a truthTab.

The optional argument outcome determines which factors in x shall be treated as outcomes. If outcome is at its default value NULL, randomAsf and randomCsf randomly draw factor(s) from x to be treated as outcome(s).

The argument compl controls the complexity of the generated asf and csf. More specifically, the *initial* complexity of asf and csf (i.e. the number of factors included in msc and the number of msc included in asf prior to redundancy elimination) is drawn from the vector compl. As this complexity might be reduced in the subsequent process of redundancy elimination, issued asf or csf will often have lower complexity than specified in compl. The default value of compl is determined by the number of columns in x. Assigning unduly high values to compl results in an error.

randomAsf has the additional argument how with the two possible values "inus" and "minimal". how = "inus" determines that the generated asf is redundancy-free relative to all logically possible configurations of the factors in x, i.e. relative to full.tt(x), whereas in case of how = "minimal" redundancy-freeness is imposed only relative to all configurations actually contained in x, i.e. relative to x itself. Typically "inus" should be used; the value "minimal" is relevant mainly in repeated randomAsf calls from within randomCsf. Moreover, setting how = "minimal" will return an error if x is a truthTab of type "fs".

The argument n.asf controls the number of asf in the generated csf. Its value is limited to ncol(x)-2 and overridden by length(outcome), if outcome is not NULL. Analogously to compl, n.asf specifies the number of asf prior to redundancy elimination, which, in turn, may further reduce these numbers. That is, n.asf provides an upper bound for the number of asf in the resulting csf.

Value

The randomly generated formula, a character string.

See Also

is.submodel, selectCases, full.tt, truthTab, cna.

Examples

```
# randomAsf
# ---------
# Asf generated from explicitly specified binary factors.
randomAsf(full.tt("H*I*T*R*K"))
randomAsf(full.tt("Johnny*Debby*Aurora*Mars*James*Sonja"))

# Asf generated from a specified number of binary factors.
randomAsf(full.tt(7))

# Asf generated from an existing data frame.
randomAsf(d.educate)

# Specify the outcome.
randomAsf(d.educate, outcome = "G")

# Specify the complexity.
randomAsf(full.tt(7), compl = 2)
randomAsf(full.tt(7), compl = 3:4)

# Redundancy-freeness relative to x instead of full.tt(x).
randomAsf(d.educate, outcome = "G", how = "minimal")

# Asf with multi-value factors (x must be given as a truthTab).
randomAsf(mvtt(allCombs(c(3,4,3,5,3,4))))

# Asf from fuzzy-set data (x must be given as a truthTab).
randomAsf(fstt(d.jobsecurity))
randomAsf(fstt(d.jobsecurity), outcome = "JSR")

# Generate 20 asf.
replicate(20, randomAsf(full.tt(7), compl = 2:3))


# randomCsf
# ---------
# Csf generated from explicitly specified binary factors.
randomCsf(full.tt("H*I*T*R*K*Q*P"))

# Csf generated from a specified number of binary factors.
randomCsf(full.tt(7))

# Specify the outcomes.
randomCsf(d.volatile, outcome = c("RB","SE"))

# Specify the complexity.
randomCsf(d.volatile, outcome = c("RB","SE"), compl = 2)
randomCsf(full.tt(7), compl = 3:4)

# Specify the number of asf.
randomCsf(full.tt(7), n.asf = 3)

# Csf with multi-value factors (x must be given as a truthTab).
randomCsf(mvtt(allCombs(c(3,4,3,5,3,4))))

# Generate 20 csf.
replicate(20, randomCsf(full.tt(7), n.asf = 2, compl = 2:3))


# Inverse searches
# ----------------
# === Ideal Data ===
# Draw the data generating structure. (Every run yields different
# targets and data.)
target <- randomCsf(full.tt(5), n.asf = 2)
target
# Select the cases compatible with the target.
x <- selectCases(target)
# Run CNA without an ordering.
mycna <- cna(x, maxstep = c(4, 4, 12), rm.dup.factors = FALSE)
# Extract the first 100 csf (depending on the seed, there may be
# more than 100 csf).
csfs <- csf(mycna, 100)
# Eliminate possible structural redundancies from the csf.
min.csfs <- minimalizeCsf(csfs$condition, x)$condition
# Check whether the target is completely returned.
any(unlist(lapply(min.csfs, identical.model, target)))

# === Data fragmentation (20% missing observations) ===
# Draw the data generating structure. (Every run yields different
# targets and data.)
target <- randomCsf(full.tt(7), n.asf = 2)
target
# Generate the complete data.
x <- tt2df(selectCases(target))
# Introduce fragmentation.
x <- x[-sample(1:nrow(x), nrow(x)*0.2), ]
# Run CNA without an ordering.
mycna <- cna(x, maxstep = c(4, 4, 12), rm.dup.factors = FALSE)
```

```
# Extract and minimize the first 100 csf (depending on the seed, there may be
# more than 100 csf).
csfs <- csf(mycna, 100)
min.csfs <- minimalizeCsf(csfs$condition, x)
# Check whether (a submodel of) the target is actually returned.
any(is.submodel(min.csfs$condition, target))

# === Data fragmentation and noise (20% missing observations, noise ratio of 0.05) ===
# Multi-value data.
# Draw the data generating structure. (Every run yields different
# targets and data.)
fullData <- mvtt(allCombs(c(4,4,4,4,4)))
target <- randomCsf(fullData, n.asf=2, compl = 2:3)
target
# Generate the complete data.
x <- tt2df(selectCases(target, fullData))
# Introduce fragmentation.
x <- x[-sample(1:nrow(x), nrow(x)*0.2), ]
# Introduce random noise.
x <- rbind(tt2df(fullData[sample(1:nrow(fullData), nrow(x)*0.05), ]), x)
# Run CNA without an ordering.
mycna <- mvcna(x, con = .75, cov = .75, maxstep = c(3, 3, 12), rm.dup.factors = F)
# Extract and minimize the first 100 csf (depending on the seed, there may be
# more than 100 csf).
csfs <- csf(mycna, 100)
min.csfs <- if(nrow(csfs)>0) {
              as.vector(minimalizeCsf(csfs$condition, mvtt(x))$condition)
            } else {NA}
# Check whether no causal fallacy (no false positive) is returned.
if(length(min.csfs)==1 && is.na(min.csfs)) {
      TRUE } else {any(is.submodel(min.csfs, target))}
```

redundant — *Identify structurally redundant asf in a csf*

Description

redundant takes a character vector cond containing complex solution formulas (csf) as input and tests for each element of cond whether the atomic solution formulas (asf) it consists of are structurally redundant.

Usage

```
redundant(cond, x = NULL, simplify = TRUE)
```

Arguments

`cond`	Character vector specifying complex solution formulas (csf); only strings of type csf are allowed, meaning conjunctions of one or more asf.
`x`	An optional argument providing a `truthTab`, a data frame, or a list specifying the factors' value ranges if `cond` contains multi-value factors; if `x` is not `NULL`, `cond` is tested for redundancy-freeness relative to `full.tt(x)`, otherwise relative to `full.tt(cond)`.
`simplify`	Logical; if `TRUE` the result for csfs with the same number of component asfs is presented as a matrix, otherwise all results are presented as a list of logical vectors.

Details

According to the regularity theory of causation underlying CNA, a Boolean dependency structure is causally interpretable only if it does not contain any redundant elements. Boolean dependency structures may feature various types of redundancies, one of which are so-called *structural redundancies*. A csf Φ has a structural redundancy if, and only if, reducing Φ by one or more of the asf it is composed of results in a csf Φ' that is logically equivalent to Φ. To illustrate, suppose that Φ is composed of three asf: asf1 * asf2 * asf3; and suppose that Φ is logically equivalent to Φ': asf1 * asf2. In that case, asf3 makes no difference to the behaviour of the factors in Φ and Φ'; it is structurally redundant and, accordingly, must not be causally interpreted. For more details see the **cna** package vignette or Baumgartner and Falk (2018).

The function `redundant` takes a character vector `cond` composed of csf as input an tests for each element of `cond` whether it is structurally redundant or not. As a test for structural redundancies amounts to a test of logical equivalencies, it must be conducted relative to all logically possible configurations of the factors in `cond`. That space of logical possibilities is generated by `full.tt(cond)` in case of `x = NULL`, and by `full.tt(x)` otherwise. If all factors in `cond` are binary, `x` is optional and without influence on the output of `redundant`. If some factors in `cond` are multi-value, `redundant` needs to be given the range of these values. `x` can be a data frame or `truthTab` listing all possible value configurations or a list of the possible values for each factor in `cond`.

If `redundant` returns `TRUE` for a csf, that csf must not be causally interpreted but further processed by `minimalizeCsf`.

Value

A list of logical vectors or a logical matrix.

If all `csf` in `cond` have the same number of `asf` and `simplify = TRUE`, the result is a logical matrix with `length(cond)` rows and the number of columns corresponds to the number of `asf` in each `csf`. In all other cases, a list of logical vectors of the same length as `cond` is returned.

Contributors

Falk, Christoph: identification and solution of the problem of structural redundancies

References

Baumgartner, Michael and Christoph Falk. 2018. "Boolean Difference-Making: A Modern Regularity Theory of Causation". *PhilSci Archive*. url: http://philsciarchive.pitt.edu/id/eprint/14876.

See Also

condition, full.tt, is.inus, csf, minimalizeCsf.

Examples

```
# Binary factors.
cond <- c("(f + a*D <-> C)*(C + A*B <-> D)*(c + a*E <-> F)", "f + a*D <-> C")
redundant(cond)

edu.sol <- csf(cna(d.educate))$condition
redundant(edu.sol, d.educate)

redundant(edu.sol, d.educate, simplify = FALSE)


# Multi-value factors.
tt.pban <- mvtt(d.pban)
cna.pban <- cna(tt.pban, con = .8, cov = .9)
csf.pban <- csf(cna.pban)
redundant(csf.pban$condition, tt.pban)
# If no truthTab is specified defining the factors' value ranges, the space of
# logically possible configurations is limited to the factor values contained in
# csf.pban, resulting in the structural redundancy of many asf.
redundant(csf.pban$condition)
```

---

selectCases — *Select the cases/configurations compatible with a data generating causal structure*

Description

`selectCases` selects the cases/configurations that are compatible with a Boolean function, in particular (but not exclusively), a data generating causal structure, from a data frame or `truthTab`.

`selectCases1` allows for setting consistency (con) and coverage (cov) thresholds. It then selects cases/configurations that are compatible with the data generating structure to degrees con and cov.

Usage

```
selectCases(cond, x = full.tt(cond), type, cutoff = 0.5,
            rm.dup.factors = FALSE, rm.const.factors = FALSE)
selectCases1(cond, x = full.tt(cond), type, con = 1, cov = 1,
             rm.dup.factors = FALSE, rm.const.factors = FALSE)
```

Arguments

cond	Character string specifying the Boolean function for which compatible cases are to be selected.
x	Data frame or `truthTab`; if not specified, `full.tt(cond)` is used.

`type`	Character vector specifying the type of `x`: `"cs"` (crisp-set), `"mv"` (multi-value), or `"fs"` (fuzzy-set). Defaults to the type of `x`, if `x` is a `truthTab` or to `"cs"` otherwise.
`cutoff`	Cutoff value in case of `"fs"` data.
`rm.dup.factors`	Logical; if `TRUE`, all but the first of a set of factors with identical value distributions are eliminated.
`rm.const.factors`	Logical; if `TRUE`, constant factors are eliminated.
`con, cov`	Numeric scalars between 0 and 1 to set the minimum consistency and coverage thresholds.

Details

In combination with `allCombs`, `full.tt`, `randomConds` and `makeFuzzy`, `selectCases` is useful for simulating data, which are needed for inverse search trials benchmarking the output of `cna`.

`selectCases` draws those cases/configurations from a data frame or `truthTab` `x` that are compatible with a data generating causal structure (or any other Boolean or set-theoretic function), which is given to `selectCases` as a character string `cond`. If the argument `x` is not specified, configurations are drawn from `full.tt(cond)`. `cond` can be a condition of any of the three types of conditions, *boolean*, *atomic* or *complex* (see `condition`). To illustrate, if the data generating structure is "A + B <-> C", then a case featuring A=1, B=0, and C=1 is selected by `selectCases`, whereas a case featuring A=1, B=0, and C=0 is not (because according to the data generating structure, A=1 must be associated with C=1, which is violated in the latter case). The type of the data frame is specified by the argument `type` taking `"cs"` (crisp-set), `"mv"` (multi-value), and `"fs"` (fuzzy-set) as values.

`selectCases1` allows for providing consistency (`con`) and coverage (`cov`) thresholds, such that some cases that are incompatible with `cond` are also drawn, as long as `con` and `cov` remain satisfied. The solution is identified by an algorithm aiming at finding a subset of maximal size meeting the `con` and `cov` requirements. In contrast to `selectCases`, `selectCases1` only accepts a condition of type *atomic* as its `cond` argument, i.e. an atomic solution formula. Data drawn by `selectCases1` can only be modeled with consistency = `con` and coverage = `cov`.

Value

A `truthTab`.

See Also

`allCombs`, `full.tt`, `randomConds`, `makeFuzzy`, `truthTab`, `condition`, `cna`, `d.jobsecurity`

Examples

```
# Generate all configurations of 5 dichotomous factors that are compatible with the causal
# chain (A*b + a*B <-> C) * (C*d + c*D <-> E).
groundTruth.1 <- "(A*b + a*B <-> C) * (C*d + c*D <-> E)"
(dat1 <- selectCases(groundTruth.1))
condition(groundTruth.1, dat1)

# Randomly draw a multi-value ground truth and generate all configurations compatible with it.
```

```
dat1 <- allCombs(c(3, 3, 4, 4, 3))
groundTruth.2 <- randomCsf(mvtt(dat1), n.asf=2)
(dat2 <- selectCases(groundTruth.2, dat1, type = "mv"))
condition(groundTruth.2, dat2)

# Generate all configurations of 5 fuzzy-set factors compatible with the causal structure
# A*b + C*D <-> E, such that con = .8 and cov = .8.
dat1 <- allCombs(c(2, 2, 2, 2, 2)) - 1
dat2 <- makeFuzzy(dat1, fuzzvalues = seq(0, 0.45, 0.01))
(dat3 <- selectCases1("A*b + C*D <-> E", con = .8, cov = .8, dat2))
condition("A*b + C*D <-> E", dat3)

# Inverse search for the data generating causal structure A*b + a*B + C*D <-> E from
# fuzzy-set data with non-perfect consistency and coverage scores.
dat1 <- allCombs(c(2, 2, 2, 2, 2)) - 1
set.seed(9)
dat2 <- makeFuzzy(dat1, fuzzvalues = 0:4/10)
dat3 <- selectCases1("A*b + a*B + C*D <-> E", con = .8, cov = .8, dat2)
fscna(dat3, ordering = list("E"), strict = TRUE, con = .8, cov = .8)

# Draw cases satisfying specific conditions from real-life fuzzy-set data.
tt.js <- fstt(d.jobsecurity)
selectCases("S -> C", tt.js)  # Cases with higher membership scores in C than in S.
selectCases("S -> C", d.jobsecurity, type = "fs")  # Same.
selectCases("S <-> C", tt.js) # Cases with identical membership scores in C and in S.
selectCases1("S -> C", con = .8, cov = .8, tt.js)  # selectCases1 makes no distinction
              #  between "->" and "<->".
condition("S -> C", selectCases1("S -> C", con = .8, cov = .8, tt.js))

# selectCases not only draws cases compatible with Boolean causal models. Any Boolean or
# set-theoretic function can be given as cond.
selectCases("C > B", allCombs(2:4), type = "mv")
selectCases("C=2 | B!=3", allCombs(2:4), type = "mv")
selectCases("A=1 * !(C=2 + B!=3)", allCombs(2:4), type = "mv")
```

`some` *Randomly select configurations from a data frame or* `truthTab`

Description

Randomly select configurations from a data frame or `truthTab` with or without replacement.

Usage

```
some(x, ...)

## S3 method for class 'data.frame'
some(x, n = 10, replace = TRUE, ...)
## S3 method for class 'truthTab'
some(x, n = 10, replace = TRUE, ...)
```

Arguments

x	Data frame or truthTab.
n	Sample size.
replace	Logical; if TRUE, configurations are sampled with replacement.
...	Not used.

Details

The function some randomly samples configurations from x, which is a data frame or truthTab. Such samples can, for instance, be used to simulate data fragmentation (limited diversity), i.e. the failure to observe/measure all configurations that are compatible with a data generating causal structure. They can also be used to simulate large-N data featuring multiple cases instantiating each configuration.

Value

A data frame or truthTab.

Note

The some generic function and the method for data frames are adopted from the **car** package. In particular, our data.frame-method has an additional argument replace, which is TRUE by default. It will by default not apply the same sampling scheme as the method in **car**.

References

Krook, Mona Lena. 2010. "Women's Representation in Parliament: A Qualitative Comparative Analysis." *Political Studies* 58(5):886-908.

See Also

truthTab, selectCases, allCombs, makeFuzzy, cna, d.women

Examples

```
# Randomly sample configurations from the dataset analyzed by Krook (2010).
tt.women <- truthTab(d.women)
some(tt.women, 20)
some(tt.women, 5, replace = FALSE)
some(tt.women, 5, replace = TRUE)

# Simulate limited diversity in data generated by the causal structure
# A=2*B=1 + C=3*D=4 <-> E=3.
dat1 <- allCombs(c(3, 3, 4, 4, 3))
dat2 <- selectCases("A=2*B=1 + C=3*D=4 <-> E=3", dat1, type = "mv")
(dat3 <- some(dat2, 150, replace = TRUE))
mvcna(dat3)

# Simulate large-N fuzzy-set data generated by the common-cause structure
# (A*b*C + B*c <-> D) * (A*B + a*C <-> E).
```

```
dat1 <- selectCases("(A*b*C + B*c <-> D) * (A*B + a*C <-> E)")
dat2 <- some(dat1, 250, replace = TRUE)
dat3 <- makeFuzzy(tt2df(dat2), fuzzvalues = seq(0, 0.45, 0.01))
fscna(dat3, ordering = list(c("D", "E")), strict = TRUE, con = .8, cov = .8)
```

truthTab *Assemble cases with identical configurations in a truth table*

Description

The `truthTab` function assembles cases with identical configurations from a crisp-set (cs), multi-value (mv), or fuzzy-set (fs) data frame in a table called a *truth table* (which is a very different type of object for CNA than for the related method of QCA).

Usage

```
truthTab(x, type = c("cs", "mv", "fs"), frequency = NULL,
         case.cutoff = 0, rm.dup.factors = TRUE, rm.const.factors = TRUE,
         .cases = NULL, verbose = TRUE)
cstt(...)
mvtt(...)
fstt(...)

## S3 method for class 'truthTab'
print(x, show.cases = NULL, ...)
```

Arguments

`x`	Data frame or matrix.
`type`	Character vector specifying the type of `x`: `"cs"` (crisp-set), `"mv"` (multi-value), or `"fs"` (fuzzy-set).
`frequency`	Numeric vector of length `nrow(x)`. All elements must be non-negative.
`case.cutoff`	Minimum number of occurrences (cases) of a configuration in `x`. Configurations with fewer than `case.cutoff` occurrences (cases) are not included in the truth table.
`rm.dup.factors`	Logical; if `TRUE`, all but the first of a set of factors with identical values in `x` are eliminated.
`rm.const.factors`	Logical; if `TRUE`, factors with constant values in `x` are eliminated.
`.cases`	Set case labels (row names): optional character vector of length `nrow(x)`.
`verbose`	Logical; if `TRUE`, some messages on the truth table are printed.
`show.cases`	Logical; if `TRUE`, the attribute "cases" is printed.
`...`	In `cstt`, `mvtt`, `fstt`: any formal argument of `truthTab` except `type`. In `print.truthTab`: arguments passed to `print.data.frame`.

Details

The first input `x` of the `truthTab` function is a data frame. To ensure that no misinterpretations of issued asf and csf can occur, users are advised to use only upper case letters as factor (column) names. Column names may contain numbers, but the first sign in a column name must be a letter. Only ASCII signs should be used for column and row names.

The `truthTab` function merges multiple rows of `x` featuring the same configuration into one row, such that each row of the resulting table, which is called a *truth table*, corresponds to one determinate configuration of the factors in `x`. The number of occurrences (cases) and an enumeration of the cases are saved as attributes "n" and "cases", respectively. The attribute "n" is always printed in the output of `truthTab`, the attribute "cases" is printed if the argument `show.cases` is `TRUE` in the `print` method.

The argument `type` specifies the type of data. `"cs"` stands for crisp-set data featuring factors that only take values 1 and 0; `"mv"` stands for multi-value data with factors that can take any non-negative integers as values; `"fs"` stands for fuzzy-set data comprising factors taking real values from the interval [0,1], which are interpreted as membership scores in fuzzy sets. To abbreviate the specification of the data type using the `type` argument, the functions `cstt(x,...)`, `mvtt(x,...)`, and `fstt(x,...)` are available as shorthands for `truthTab(x,type = "cs",...)`, `truthTab(x,type = "mv",...)`, and `truthTab(x,type = "fs",...)`, respectively.

Instead of multiply listing identical configurations in `x`, the `frequency` argument can be used to indicate the frequency of each configuration in the data frame. `frequency` takes a numeric vector of length `nrow(x)` as value. For instance, `truthTab(x,frequency = c(3,4,2,3))` determines that the first configuration in `x` is featured in 3 cases, the second in 4, the third in 2, and the fourth in 3 cases.

The `case.cutoff` argument is used to determine that configurations are only included in the truth table if they are instantiated at least as many times in `x` as the number assigned to `case.cutoff`. Or differently, configurations that are instantiated less than the number given to `case.cutoff` are excluded from the truth table. For instance, `truthTab(x,case.cutoff = 3)` entails that configurations with less than 3 cases are excluded.

`rm.dup.factors` and `rm.const.factors` allow for determining whether all but the first of a set of duplicated factors (i.e. factors with identical value distributions in `x`) are eliminated and whether constant factors (i.e. factors with constant values in all cases (rows) in `x`) are eliminated. From the perspective of configurational causal modeling, factors with constant values in all cases can neither be modeled as causes nor as outcomes; therefore, they can be removed prior to the analysis. Factors with identical value distributions cannot be distinguished configurationally, meaning they are one and the same factor as far as configurational causal modeling is concerned. Therefore, only one factor of a set of duplicated factors is standardly retained by `truthTab`.

`.cases` can be used to set case labels (row names). It is a character vector of length `nrow(x)`.

The `row.names` argument of the `print` function determines whether the case labels of `x` are printed or not. By default, `row.names` is `TRUE` unless the (comma-separated) list of the `cases` exceeds 20 characters in one row at least.

Value

An object of type "truthTab", i.e. a data.frame with additional attributes "type", "n" and "cases".

Note

For those users of **cna** that are familiar with Qualitative Comparative Analysis (QCA), it must be emphasized that a *truth table* is a very different type of object in the context of CNA than it is in the context of QCA. While a QCA truth table is a list indicating whether a minterm (i.e. a configuration of all exogenous factors) is sufficient for the outcome or not, a CNA truth table is simply an integrated representation of the input data that lists all configurations in the data exactly once. A CNA truth table does not express any relations of sufficiency whatsoever.

References

Aleman, Jose. 2009. “The Politics of Tripartite Cooperation in New Democracies: A Multi-level Analysis.” *International Political Science Review* 30 (2):141-162.

Greckhamer, Thomas, Vilmos F. Misangyi, Heather Elms, and Rodney Lacey. 2008. “Using Qualitative Comparative Analysis in Strategic Management Research: An Examination of Combinations of Industry, Corporate, and Business-Unit Effects.” *Organizational Research Methods* 11 (4):695-726.

Thiem, Alrik. 2018. “QCApro: Advanced Functionality for Performing and Evaluating Qualitative Comparative Analysis.” *R Package Version 1.1-2.* URL: http://www.alrik-thiem.net/software/.

See Also

`cna`, `condition`, `allCombs`, `d.performance`, `d.pacts`

Examples

```
# Manual input of cs data
# -----------------------
dat1 <- data.frame(
  A = c(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),
  B = c(1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0),
  C = c(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0),
  D = c(1,1,1,1,0,0,0,0,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,0,0,0),
  E = c(1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0,0,0)
)

# Default return of the truthTab function.
truthTab(dat1)

# Recovering the cases featuring each configuration by means of the print function.
print(truthTab(dat1), show.cases = TRUE)

# The same truth table as before can be generated by using the frequency argument while
# listing each configuration only once.
dat1 <- data.frame(
  A = c(1,1,1,1,1,1,0,0,0,0,0),
  B = c(1,1,1,0,0,0,1,1,1,0,0),
  C = c(1,1,1,1,1,1,1,1,1,0,0),
  D = c(1,0,0,1,0,0,1,1,0,1,0),
  E = c(1,1,0,1,1,0,1,0,1,1,0)
)
```

```
truthTab(dat1, frequency = c(4,3,1,3,4,1,10,1,3,3,3))

# Set (random) case labels.
print(truthTab(dat1, .cases = sample(letters, nrow(dat1), replace = FALSE)),
      show.cases = TRUE)

# Truth tables generated by truthTab can be input into the cna function.
dat1.tt <- truthTab(dat1, frequency = c(4,3,1,3,4,1,4,1,3,3,3))
cna(dat1.tt, con = .85, details = TRUE)

# By means of the case.cutoff argument configurations with less than 2 cases can
# be excluded (which yields perfect consistency and coverage scores for dat1).
dat1.tt <- truthTab(dat1, frequency = c(4,3,1,3,4,1,4,1,3,3,3), case.cutoff = 2)
cna(dat1.tt, details = TRUE)


# Simulating multi-value data with biased samples (exponential distribution)
# -------------------------------------------------------------------------
dat1 <- allCombs(c(3,3,3,3,3))
set.seed(32)
m <- nrow(dat1)
wei <- rexp(m)
dat2 <- dat1[sample(nrow(dat1), 100, replace = TRUE, prob = wei),]
truthTab(dat2, type = "mv") # 100 cases with 46 configurations instantiated only once.
mvtt(dat2, case.cutoff = 2) # removing the single instances.

# Duplicated factors are not eliminated, constant factors are not eliminated.
dat3 <- selectCases("(A=1+A=2+A=3 <-> C=2)*(B=3<->D=3)*(B=2<->D=2)*(A=2 + B=1 <-> E=2)",
                    dat1, type = "mv")
mvtt(dat3, rm.dup.factors = FALSE, rm.const.factors = FALSE)


# truthTab with fuzzy-set data from Aleman (2009)
# ----------------------------------------------
# Include all cases.
tt.pacts <- fstt(d.pacts)
fscna(tt.pacts, con = .93, cov = .86, details = TRUE)

# Only include configurations with at least 3 cases.
tt.pacts2 <- fstt(d.pacts, case.cutoff = 3)
fscna(tt.pacts2, con = .93, cov = .86, details = TRUE)


# Large-N data with crisp sets from Greckhamer et al. (2008)
#-------------------------------------------------------------
truthTab(d.performance[1:8], frequency = d.performance$frequency)

# Eliminate configurations with less than 5 cases.
truthTab(d.performance[1:8], frequency = d.performance$frequency, case.cutoff = 5)

# Various large-N CNAs of d.performance with varying case cut-offs.
cna(truthTab(d.performance[1:8], frequency = d.performance$frequency, case.cutoff = 4),
    ordering = list("SP"), con = .75, cov = .6)
```

```
cna(truthTab(d.performance[1:8], frequency = d.performance$frequency, case.cutoff = 5),
    ordering = list("SP"), con = .75, cov = .6)
cna(truthTab(d.performance[1:8], frequency = d.performance$frequency, case.cutoff = 10),
    ordering = list("SP"), con = .75, cov = .6)
print(cna(truthTab(d.performance[1:8], frequency = d.performance$frequency, case.cutoff = 15),
          ordering = list("SP"), con = .75, cov = .6, what = "a"), nsolutions = "all")
```

---

tt2df *Transform a truth table into a data frame*

Description

Transform a truth table into a data frame. This is the converse function of truthTab.

Usage

```
tt2df(tt)
```

Arguments

tt	A truthTab.

Details

Rows in the truthTab corresponding to several cases are rendered as multiple rows in the resulting data frame.

Value

A data frame.

See Also

truthTab

Examples

```
tt.educate <- truthTab(d.educate[1:2])
tt.educate
tt2df(tt.educate)

dat1 <- some(truthTab(allCombs(c(2, 2, 2, 2, 2)) - 1), n = 200, replace = TRUE)
dat2 <- selectCases("(A*b + a*B <-> C)*(C*d + c*D <-> E)", dat1)
dat2
tt2df(dat2)

dat3 <- data.frame(
  A = c(1,1,1,1,1,1,0,0,0,0,0),
  B = c(1,1,1,0,0,0,1,1,1,0,0),
  C = c(1,1,1,1,1,1,1,1,1,0,0),
  D = c(1,0,0,1,0,0,1,1,0,1,0),
  E = c(1,1,0,1,1,0,1,0,1,1,0)
  )
tt.dat3 <- truthTab(dat3, frequency = c(4,3,5,7,4,6,10,2,4,3,12))
tt2df(tt.dat3)
```

Index

∗Topic **datasets**
d.autonomy, 30
d.educate, 32
d.irrigate, 33
d.jobsecurity, 33
d.minaret, 34
d.pacts, 35
d.pban, 35
d.performance, 36
d.volatile, 37
d.women, 37
∗Topic **package**
cna-package, 2

allCombs, 4, *19*, *38*, *39*, *45*, *57*, *59*, *62*
as.condTbl *(*condTbl*)*, 24
asf *(*condTbl*)*, 24

cna, 6, *19*, *20*, *22*, *25*, *26*, *28*, *41–47*, *49–52*, *57*, *59*, *62*
cna-package, 2
coherence, *9*, *10*, *13*, 18
condition, *13*, *18*, *19*, 19, *25*, *26*, *39*, *42*, *45–47*, *56*, *57*, *62*
condTbl, *13*, *19*, *22*, 24
cscna *(*cna*)*, 6
cscond *(*condition*)*, 19
csf, *50*, *56*
csf *(*condTbl*)*, 24
cstt *(*truthTab*)*, 60
cyclic, 28

d.autonomy, *13*, 30
d.educate, *13*, 32
d.irrigate, *22*, *26*, 33
d.jobsecurity, 33, *57*
d.minaret, 34
d.pacts, 35, *62*
d.pban, *13*, 35
d.performance, 36, *62*
d.volatile, 37
d.women, *13*, 37, *59*

fscna *(*cna*)*, 6
fscond *(*condition*)*, 19
fstt *(*truthTab*)*, 60
full.tt, *4*, *5*, *13*, *19*, 38, *42*, *45*, *47*, *50*, *52*, *56*, *57*

group.by.outcome *(*condition*)*, 19

identical.model *(*is.submodel*)*, 42
is.inus, *7*, *9*, *13*, 40, *47*, *56*
is.submodel, *5*, *13*, 42, *52*

makeFuzzy, *5*, *13*, 45, *57*, *59*
minimalize, *41*, *42*, 46
minimalizeCsf, *10*, *13*, *26*, 48, *55*, *56*
msc *(*condTbl*)*, 24
mvcna *(*cna*)*, 6
mvcond *(*condition*)*, 19
mvtt *(*truthTab*)*, 60

print.cna *(*cna*)*, 6
print.cond *(*condition*)*, 19
print.condList *(*condition*)*, 19
print.condTbl *(*condTbl*)*, 24
print.data.frame, *25*, *60*
print.minimalizeCsf *(*minimalizeCsf*)*, 48
print.truthTab, *7*, *20*
print.truthTab *(*truthTab*)*, 60

randomAsf *(*randomConds*)*, 50
randomConds, *5*, *13*, *43*, *44*, 50, *57*
randomCsf *(*randomConds*)*, 50
redundant, *10*, *13*, *41*, *42*, *50*, 54

selectCases, *5*, *13*, *19*, *39*, *43–45*, *52*, 56, *59*
selectCases1 *(*selectCases*)*, 56
some, *13*, 58
summary.condList *(*condition*)*, 19

truthTab, *6*, *7*, *13*, *20*, *22*, *26*, *39*, *45*, *48*, *52*, *57*, *59*, 60, *64*
tt2df, *45*, 64