Package 'RNetica'

August 24, 2017

Version 0.5-1

Date 2017/08/24

Title R interface to Netica(R) Bayesian Network Engine

Author Russell Almond

Maintainer Russell Almond <ralmond@fsu.edu>

Depends R (>= 3.0), methods, utils

Imports grDevices, R.utils

Description
This provides an R interface to the Netica (http://norsys.com/) Bayesian network library API.

License Artistic-2.0 + file LICENSE

URL http://pluto.coe.fsu.edu/RNetica

Collate Session.R Networks.R Node.R Edges.R Inference.R Cases.R
Experience.R Continuous.R Random.R LoadFuns.R

R topics documented:

RNetica-package *R interface to Netica(R) Bayesian Network Engine*

Description

This provides an R interface to the Netica (http://norsys.com/) Bayesian network library API.

Details

The DESCRIPTION file: This package was not yet installed at build time.

This package provides an R interface to the Netica, in particular, it binds many of the functions in the Netica C API into the R language. RNetica can create and modify networks, enter evidence and extract the conditional probabilities from a Netica network.

License

While RNetica (the combination of R and C code that connects R and Netica) is free software, as is R, Netica is a commercial product. Users of RNetica will need to purchase a Netica API license key (which is different from the GUI license key) from Norsys(R) (`http://www.norsys.com/`).

Once you have a license key, you can use it in one of three ways. The currently (RNetica 0.5 and later) recommended way of using it is to create a Netica Session object that contains it: `DefaultNeticaSession <- NeticaS` This will store the key in a `NeticaSession` object. The special variable `DefaultNeticaSession` is used as a default for every function requiring a session argument, so can be used to skip the need for explicitly stating the session argument.

Two other mechanisms continue to be supported for backwards compatibility. First, the license key can be used as an argument to the function `StartNetica()`. This will create a session and store it in `NeticaDefaultSession`. If the variable `NeticaLicenseKey` in the R top-level environment is set before the call to `library(RNetica)`, `StartNetica()` will pick up the license key from that location.

Without the license key, the Netica shared library will be restricted to a student/demonstration mode with limited functionality. Note that all of the example code (and hence `R CMD check RNetica`) can be run using the limited version.

Note that the `NeticaSession` object stores the complete Netica license key. Do not share dumps of the session object (including the `.RData` file containing `DefaultNeticaSession`) with any third-party.

Index

Index: This package was not yet installed at build time.

RNetica Environment and Netica Objects

Netica exists in both as a stand alone graphical tool for building and manipulating Bayesian networks (the Netica GUI) and as a shared library for manipulating Bayesian networks (the Netica API). The RNetica package binds the API version of Netica to a series of R functions which do much of the work of manipulating the network. The file format for the GUI and API version of Netica is identical, so analysts can easily move back and forth between the two. Note that the RNetica environment is separate from other Netica environments that may be created using the Netica GUI (or API invoked from a different program); RNetica can only manipulate the networks that are currently loaded into its environment.

There are five objects which provide a handle for objects in the Netica session. These are:

`NeticaSession` This is a container for the overall Netica session. It is referenced when creating other Netica objects (`NeticaBN`s, `CaseStream`s and `NeticaRNG`) and contains the license key needed to activate Netica. Its field `$nets` is an environment which contains references to all of the networks which have been associate with this session.

`NeticaBN` This is a handle for a network object. `NeticaNode` objects are created within a network, and the `$nodes` field is an environment which contains node references, at least for those nodes which have been referenced in R code. Networks must have unique names within a session.

`NeticaNode` This is a handle for a particular Netica node. Nodes must have unique names within a network. Many inference functions are done based on nodes.

`CaseStream` This is a stream of Netica case data, values for particular nodes. There are two subclasses: `FileCaseStream` and `MemoryCaseStream`. As of version 5.04 of the Netica API, there are some issues with `MemoryCaseStream`s, so the `FileCaseStream`s should be used instead.

`NeticaRNG` This a random number generator used by Netica for generating random cases.

All of these follow the `envRefClass` protocol. In particular, their fields are referenced using '$'. Also, all of them have a method `$isActive` (which is called from the generic function `is.active`) which determines whether or not the pointer to the Netica object currently exists or not. Calling `stopSession` will render all Netica objects inactive.

In particular, when quitting and restarting R, the pointers will all be initialised to null, and all of the session, node and network objects will become inactive. Some examples of how to restart an RNetica session are provided below.

To connect R to Netica, it is necessary to create and start a `NeticaSession`. This is done by first calling the constructor `NeticaSession()` and then calling the function `startSession(`*session*`)`. If you have purchased a Netica license key from Norsys, this can be passed to the constructor with the argument `LicenseKey` given the value of the license key as a string. Note that the session object can be saved in the workspace, so that it can be used in future R session (it does not need to be recreated, but it must be restarted with a call to `startSession`). If it is saved to `DefaultNeticaSession`, this value will be used as a default by all of the functions that use the session as an argument.

Note that this is a change from how RNetica operated prior to version 0.5. In older versions of RNetica, the session pointer was held inside of the C code, and the function `StartNetica()` was invoked automatically when the `RNetica` package was attached. Nowt this needs to be done manually through a call to `startSession`.

The function `getDefaultSession()` emulates the behaviour of the previous version of RNetica. It is the default value for all of the functions which require a session argument. When invoked,

it looks for an object call DefaultNeticaSession in the global environment. If that exists, it is used, if not, a new NeticaSession is created. If the new session is created, it looks for a variable NeticaLicenseKey in the global environment. If that is present, it will use this as a license key. Finally, if the DefaultNeticaSession is not active, it will start it.

Note that it is almost certainly a mistake to have two sessions open at the same time. Users should either set the DefaultNeticaSession, and use the default, or always explicitly pass the session argument to functions that need it.

The following functions take a session argument: CaseFileDelimiter, CaseFileMissingCode, CaseFileStream, CaseMemoryStream, ClearAllErrors, CreateNetwork, GetNthNetwork, GetNamedNetworks, NeticaVersion, ReadNetworks, ReportErrors, StartNetica, StopNetica, startSession, and stopSession.

Netica Networks

NeticaBN objects are created through one of three functions: CreateNetwork(), ReadNetworks() and CopyNetworks. The first two both require a session argument, while the third uses the session from its *net* argument. When a network is created it is added as a symbol (using its name) to the $nets field of the session. It can then be referenced using *session*$nets$*netname* or *session*$nets[["*netname*"]]. The field $Session of the NeticaBN points to the NeticaSession object in which the network was created.

Note that *session*$nets cache may contain inactive network objects for one of two reasons: (1) it is a deleted network object, or (2) this is a session which has been restored from a file, and the Netica pointers have not been reconnected. In particular, quitting R will always deactivate the network.

For networks, the simplest solution is to save each network to a file using WriteNetworks(). If a NeticaBN object *net* is used in either a *net* <- ReadNetworks() or WriteNetworks(*net*) call, then the R object will be badged with the name of the last used filename. Thus, after saving and restoring a R session, the expression *net* <- ReadNetworks(*net*) will recreate *net* as an object pointing to a new network that is identical to the last saved version.

Netica Nodes

NeticaNode objects are created through NewDiscreteNode() or NewContinuousNode(), or retrieved from the network using NetworkFindNode(), NetworkAllNodes(), NetworkNodesInSet(), or one of a variety of other functions that return nodes. When a node is created it is added as a symbol (using its name) to the $node field of the network. It can then be referenced using *net*$nodes*nodename* or *net*$nodes[["*nodename*"]].

Note that if more than one network is loaded they may have identically named nodes that are not identical. For example, *net1* and *net2* may both have a node named "Proficiency". If the R variable *Proficiency* is bound to the NeticaNode object corresponding to the variable "Proficiency" in *net1*, it can only be used to access the instance of that variable in *net1*, not the one in *net2*.

Note that the NeticaNode object is created when then node is first references in R. In particular, this means when a network is loaded through a call to ReadNetworks, the R objects for the corresponding Netica nodes are not immediately created. The function NetworkAllNodes() returns a list of all nodes in the network, and as a side effect, creates NeticaNode object for all of the nodes found in the network. If the network has many nodes, it may be more efficient to just create R objects for the ones which are used. In this case the functions NetworkFindNode(), and NetworkNodesInSet are useful for finding (and creating R objects for) a subset of nodes.

The following procedure can be used to save and restore a Netica network across sessions. In the first session:

```
DefaultNeticaSession <- NeticaSession()
startSession(DefaultNeticaSession)
net <- CreateNetwork("myNet",DefaultNeticaSession)
# Work on the network.
WriteNetworks(net,"myNet.dne")
q("yes")
```

The variables `DefaultNeticaSession` and `net` will be saved in `.Rdata`. Then in the next R session

```
startSession(DefaultNeticaSession)
net <- ReadNetworks(net,DefaultNeticaSession)
net.nodes <- NetworkAllNodes(net)
```

This will read *net* from the place it was last saved. It will also create R objects for all of the nodes in *net*. This can now be access through *net*`$nodes` or the variable `net.nodes`.

Creating and Editing Networks

Operations with Bayesian networks generally proceed in two phases: Building network, and conducting inference. This section describes the most commonly used options for building networks. The following section describes the most commonly used options for inference.

First, the function `CreateNetwork()` is used to create an empty network. Multiple networks can be open within the RNetica environment, but each must have a unique name. Names must conform to Netica's `IDname` rules.

Nodes can be added to a network with the functions `NewDiscreteNode()` and `NewContinuousNode()`. Note that Netica makes an internal distinction between these two types of nodes and a node cannot be changed from one type to another. Nodes must all have a unique (within the network) name which must conform to the `IDname` rules.

Edges between nodes are created using the `AddLink(`*parent*`,`*child*`)` function. This forms a directed graph which must be acyclic (that is it must not be possible to follow a path along the direction of the arrows and return to the starting place). The function `NodeParents(`*child*`)` returns the current set of parents for the node *child* (nodes which have edges pointing towards *child*). `NodeParents(`*child*`)` may be set, which serves several purposes. First, it allows connections to be added and removed. Second, setting one of the parent locations to `NULL` produces a special *Stub* node, which serves as a placeholder for a later connection. Third, it allows one to reorder the nodes, which determines the order of the dimensions of the conditional probability table.

A completed Bayesian network has a conditional probability table (CPT) associated with each node. The CPT provides the conditional probability distributions of the node given the states of its parents in the graph. RNetica provides two functions for accessing and setting this CPT. The function `NodeProbs()` returns (or sets) the conditional probability table as a multi-dimensional array. However, using the array extractor "[" (`Extract.NeticaNode`) allows the conditional probability table to be manipulated as a data frame, where the first several columns provide the states of the parent

variables, and the remaining columns the probabilities of the the node being in each of those states given the parent configurations. This latter approach has a number of features for working with large tables and tables with complex structure.

Finally, when the network is complete, the function `WriteNetworks()` can be used to save it to a file, which can either be later read into RNetica, or can be used with the Netica GUI or other applications that use the Netica API.

Inference

The basic purpose for building a Bayesian network is to rapidly calculate conditional probabilities. In Netica language, one enters *findings* (conditions) on the known or hypothesised variables and then calculates *beliefs* (conditional probabilities) on certain variables of interest.

Netica, like most Bayesian network software, uses two different graphical representations, one for model construction and one for inference. The acyclic directed graph is use for model construction (previous section). The function `CompileNetwork()` builds the second graphical representation: the junction tree. The function `JunctionTreeReport()` provides information about the compiled representation.

While compiling can take a long time (depending on the size and connectivity of the network), repeated compilations appear to be harmless. There is an `UncompileNetwork()` function, but performing any editing operation (adding or removing nodes or edges) will automatically return the network to an uncompiled state. Netica tries to preserve finding information. In particular the function `AbsorbNodes()` provides a mechanism for removing nodes from a network without changing the joint probability (including influence of findings) of the remaining nodes. (The network must be recompiled after a call to `AbsorbNodes()` though.)

The principle way to enter observed evidence is setting `NodeFinding(`*node*`) <-` *value*. The function `NodeLikelihood()` can be used to enter *virtual evidence*, however, some care must be taken as it alters the meanings of several of the other functions.

The conditional (given the entered findings and likelihoods) probability distribution can be queried at any time using the function `NodeBeliefs(`*node*`)`. If the states of a node have been given numeric values using `NodeLevels(`*node*`)`, then `NodeExpectedValue(`*node*`)` will calculate the expected numeric value (and the standard deviation). The function `JointProbability(`*nodelist*`)` calculates the joint distribution over a collection of nodes, and the function `FindingsProbability(`*net*`)` calculates the prior probability of all the findings entered into the network. The function `MostProbableConfig(nodelist)` finds the mode of the joint probability distribution (given the current findings and likelihood).

Note that in the default state, when findings are entered, the beliefs about all other nodes in the network are then updated. This can be time consuming in large networks. The function `SetNetworkAutoUpdate()` can be used to change this to a lazy updating mode, when the evidence from the findings are only propagated when required for a call to `NodeBeliefs()` or a similar function. The function `WithoutAutoUpdate(`*net*`,`*expr*`)` is useful for setting findings in a large number of nodes in *net* without the overhead of belief updating.

Node Sets

The function `NodeSets()` allows the modeller to attach labels to the nodes in the network. For the most part, Netica ignores these labels, except that it will colour nodes from various sets different colours (`NetworkNodeSetColor()`). Aside from a few internal labels used by Netica, these node sets are reserved for user programming.

RNetica provides some functions that make node sets incredibly convenient ways to describe the intended usage of the nodes. In particular, the function `NetworkNodesInSet()` returns a list of all nodes which are tagged as being in a particular node set. For example, suppose that the modeller has marked a number of nodes as being in the node set `"ReportingVar"`. Then the following code would generate a report about the network:

```
net.ReportingVars <- NetworkNodesInSet(net, "ReportingVar")
lapply(net.ReportingVars, NodeBeliefs)
```

Warning

The current status of RNetica is that of a beta release. The code base is stable enough to do useful work, but more testing is still required. Users are advised to work in such a way that they can easily recover from problems.

In particular, because RNetica calls C code, there is a possibility that it will crash R. There is also a possibility that pointers embedded in `NeticaBN` and `NeticaNode` objects will become corrupted. If such problems occur, it is best to restart R and reload the networks.

Please send information about both serious and not-so-serious problems to the maintainer.

Legal Stuff

Netica and Norsys are registered trademarks of Norsys, LLC, used by permission.

Although Norsys is generally supportive of the RNetica project, it does not officially support RNetica, and all questions should be sent to the package maintainers.

Author(s)

Russell Almond
Maintainer: Russell Almond <almond@acm.org>

References

The general Netica manual can be found at: `http://www.norsys.com/WebHelp/NETICA.htm`

The Netica API documentation can be found at `http://norsys.com/onLineAPIManual/index.html`.

Almond, R. G. & Mislevy, R. J. (1999) Graphical models and computerized adaptive testing. *Applied Psychological Measurement*, 23, 223–238.

Almond, R. G., Mislevy, R. J., Steinberg, L. S., Yan, D. & Williamson, D. M. (2015) *Bayesian Networks in Educational Assessment*. Springer.

Examples

```
############################################################
## Network Construction:
sess <- NeticaSession()
startSession(sess)
```

```
abc <- CreateNetwork("ABC", session=sess)
A <- NewDiscreteNode(abc,"A",c("A1","A2","A3","A4"))
B <- NewDiscreteNode(abc,"B",c("B1","B2","B3"))
C <- NewDiscreteNode(abc,"C",c("C1","C2"))

AddLink(A,B)
NodeParents(C) <- list(A,B)

NodeProbs(A)<-c(.1,.2,.3,.4)
NodeProbs(B) <- normalize(matrix(1:12,4,3))
NodeProbs(C) <- normalize(array(1:24,c(4,3,2)))
abcFile <- tempfile("peanut",fileext=".dne")
WriteNetworks(abc,abcFile)

DeleteNetwork(abc)

######################################################################
##  Inference using the EM-SM algorithm (Almond & Mislevy, 1999).
## System/Student model
EMSMSystem <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","System.dne"), session=sess)


## Evidence model for Task 1a
EMTask1a <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","EMTask1a.dne"), session=sess)

## Evidence model for Task 2a
EMTask2a <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","EMTask2a.dne"), session=sess)

## Task 1a has a footprint of Skill1 and Skill2 (those are the
## referenced student model nodes.  So we want joint the footprint into
## a single clique.
MakeCliqueNode(NetworkFindNode(EMSMSystem, NetworkFootprint(EMTask1a)))
## The footprint for Task2 a is already a clique, so no need to do
## anything.

## Make a copy for student 1
student1 <- CopyNetworks(EMSMSystem,"student1")
## Monitor nodes for proficiency
student1.prof <- NetworkNodesInSet(student1,"Proficiency")

student1.t1a <- AdjoinNetwork(student1,EMTask1a)
## We are done with the original EMTask1a now
DeleteNetwork(EMTask1a)

## Now add findings
CompileNetwork(student1)
NodeFinding(student1.t1a$Obs1a1) <- "Right"
NodeFinding(student1.t1a$Obs1a2) <- "Right"

student1.probt1a <- JointProbability(student1.prof)
```

```
## Done with the observables, absorb them
AbsorbNodes(student1.t1a)
CompileNetwork(student1)
student1.probt1ax <- JointProbability(student1.prof)

## Now Task 2
student1.t2a <- AdjoinNetwork(student1,EMTask2a,"t2a")
DeleteNetwork(EMTask2a)

## Add findings
CompileNetwork(student1)
NodeFinding(student1.t2a$Obs2a) <- "Half"

AbsorbNodes(student1.t2a)
CompileNetwork(student1)
student1.probt1a2ax <- JointProbability(student1.prof)

DeleteNetwork(list(student1, EMSMSystem))
stopSession(sess)
```

`AbsorbNodes` *Delete a Netica nodes in a way that maintains the connectivity.*

Description

This function deletes `NeticaNode` connecting the parents of the deleted node to its children. If multiple nodes are passed as the argument, then all of the nodes are absorbed. The joint probability distribution over the remaining nodes should be the same as the marginal probability distribution over the remaining nodes before the nodes were deleted.

Usage

```
AbsorbNodes(nodes)
```

Arguments

`nodes`	A `NeticaNode` or list of `NeticaNodes` to be deleted.

Details

This function provides a way of removing a node without affecting the connectivity, or the joint probability of the remaining nodes. In particular, all of the relationship tested by `is.NodeRelated()` among the remaining nodes should remain true (or false) when we are done.

Value

Returns `NULL`.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: AbsorbNodes_bn()

See Also

NeticaNode, AddLink(), NodeChildren(), NodeParents(), ReverseLink(), is.NodeRelated()

Examples

```
sess <- NeticaSession()
startSession(sess)
anet <- CreateNetwork("Absorbent",sess)

xnodes <- NewDiscreteNode(anet,paste("X",1:5,sep="_"))
AddLink(xnodes[[1]],xnodes[[2]])
AddLink(xnodes[[2]],xnodes[[3]])
AddLink(xnodes[[3]],xnodes[[4]])
AddLink(xnodes[[3]],xnodes[[5]])

stopifnot(
 all(match(xnodes[4:5],NodeChildren(xnodes[[3]]),nomatch=0)>0),
 is.NodeRelated(xnodes[[2]],xnodes[[3]],"parent"),
 is.NodeRelated(xnodes[[2]],xnodes[[1]],"child")
)

## These are leaf nodes, shouldn't change topology, except locally.
AbsorbNodes(xnodes[4:5])
stopifnot(
  ## Nodes 4 and 5 are now deleted
  all(!is.active(xnodes[4:5])),
  all(anet$listNodes() == c("X_1","X_2","X_3")),
  length(NodeChildren(xnodes[[3]]))==0,
  is.NodeRelated(xnodes[[2]],xnodes[[3]],"parent"),
  is.NodeRelated(xnodes[[2]],xnodes[[1]],"child")
)

## This should connect X1->X3
AbsorbNodes(xnodes[[2]])
stopifnot(
  ## Node 2 is now deleted
  !is.active(xnodes[[2]]),
  length(NodeChildren(xnodes[[3]]))==0,
  is.NodeRelated(xnodes[[1]],xnodes[[3]],"parent"),
  is.NodeRelated(xnodes[[3]],xnodes[[1]],"child")
)

DeleteNetwork(anet)
stopSession(sess)
```

AddLink *Adds or removes a link between two nodes in a Netica network.*

Description

Add link adds an edge from Parent to Child. Delete Link removes that edge. This states that the distribution of child will be specified conditional on the value of parent. Consequently, adding or removing edges with affect the conditional probability tables associated with the Child node (see NodeProbs().)

Usage

```
AddLink(parent, child)
DeleteLink(parent, child)
```

Arguments

parent	A NeticaNode representing an independent variable to be added to the conditioning side of the relationship. The nodes parent and child must both be in the same network.
child	A NeticaNode representing dependent variable to be added to the conditioning side of the relationship.

Details

After adding a link parent --> child, it may be the case that parent is in NodeParents(child) and child is a member of NodeChildren(parent). If child already has other parents, then the new parent will be added to the end of the list. The order of the parents can be set by setting NodeParents(child).

In general, the Bayesian network must always be an acyclic directed graph. Therefore, if parent is a descendant of child (that is if is.NodeRelated(child), "descendant", child is TRUE), then Netica will generate an error.

The function DeleteLink() removes the relationship, and the parent and child nodes should no longer be in each other parent and child lists. The parent list of the child node is shortened (a stub node for later reconnection is not created as when NodeParents(child)[i] <- list(NULL)).

Value

The function AddLinK invisibly returns the index of the new parent in the parent list.

The function DeleteLink invisibly returns the child node.

Note

The Netica API specifies the first argument to DeleteLink_bn() as an index into the parent list. RNetica maps from the node to the index.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: AddLink_bn(), DeleteLink_bn()

See Also

NeticaNode, NodeParents(), NodeChildren(), is.NodeRelated()

Examples

```
sess <- NeticaSession()
startSession(sess)

abnet <- CreateNetwork("AABB", session=sess)
A <- NewDiscreteNode(abnet, "A")
B <- NewDiscreteNode(abnet, "B")

AddLink(A,B)

stopifnot(
  is.element(list(A),NodeParents(B)),
  is.element(list(B),NodeChildren(A))
)

DeleteLink(A,B)

stopifnot(
  !is.element(list(A),NodeParents(B)),
  !is.element(list(B),NodeChildren(A))
)

DeleteNetwork(abnet)
stopSession(sess)
```

AdjoinNetwork *Links an evidence model network to a system model network.*

Description

This function assumes that the two arguments are networks that were designed to be connected to one another. It copies the nodes from em into sm and then tries to resolve any stub links in the copied nodes by connecting them to nodes in sm.

Usage

```
AdjoinNetwork(sm, em, setname = character())
```

Arguments

sm	An active NeticaBN which contains the system state variables.
em	An active NeticaBN which contains variables that provide evidence about the system state.
setname	An optional character vector containing names of node sets (see NodeSets()). If supplied, all of the newly created nodes are added to the node sets. Note that all node set names must conform to the IDname rules.

Details

This follows the System Model–Evidence Model (or Hub-and-spoke) protocol laid out in Almond et al (1999) and Almond and Mislevy (1999). The idea is that the network sm is a complete network that encodes beliefs about the current status of a system. In particular, it often encodes the state of knowledge about a student and is then called a *student model*.

The second network em is an incomplete network: a fragment of a network, some of whose nodes could be stub nodes referring to nodes in the sm (see NodeInputNames() and NodeKind()). The idea is that the *evidence model* provides a set of observable values associated with some diagnostic procedure, in particular, a task on an assessment.

The function AdjoinNetwork(sm,em) copies all of the nodes from em to sm, modifying sm in the process (copy it first using CopyNetworks(sm) if this is not the intention). It then the parents of each node, emnode, in em looking for stub nodes (cases where NodeParents(emnode)[j] has been set to NULL for some parent. AdjoinNetworks(sm,em) then tries to find a matching parent by searching for a system model node, smnode named NodeInputNames(emnode)[j]. If it finds one, it sets NodeParents(emnode)[j] <- smnode; if not, it issues a warning.

The function AdjoinNetwork(sm,em) also copies node set information from the nodes in em to their copies in sm. The value of setname is concatenated with the current node sets of the nodes in em. This provides a handy way of identifying the evidence model from which the nodes came.

After findings are entered on the nodes in the evidence model, the can be eliminated using AbsorbNodes().

Value

A list containing the newly copied nodes (the instances of the em nodes now in sm).

Author(s)

Russell Almond

References

Almond, R. G. & Mislevy, R. J. (1999) Graphical models and computerized adaptive testing. *Applied Psychological Measurement*, 23, 223–238.

Almond, R., Herskovits, E., Mislevy, R. J., & Steinberg, L. S. (1999). Transfer of information between system and evidence models. In Artificial Intelligence and Statistics 99, Proceedings (pp. 181–186). Morgan-Kaufman

See Also

NeticaNode, AbsorbNodes(), JointProbability(), NodeSets(), CopyNodes(), NetworkFootprint()

Examples

```
sess <- NeticaSession()
startSession(sess)

## System/Student model
EMSMSystem <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","System.dne"), session=sess)


## Evidence model for Task 1a
EMTask1a <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","EMTask1a.dne"), session=sess)

## Evidence model for Task 2a
EMTask2a <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","EMTask2a.dne"), session=sess)

## Evidence model for Task 2b
EMTask2b <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","EMTask2b.dne"), session=sess)

## Task 1a has a footprint of Skill1 and Skill2 (those are the
## referenced student model nodes.  So we want joint the footprint into
## a single clique.
MakeCliqueNode(NetworkFindNode(EMSMSystem, NetworkFootprint(EMTask1a)))
## The footprint for Task2 a is already a clique, so no need to do
## anything.

## Make a copy for student 1
student1 <- CopyNetworks(EMSMSystem,"student1")
## Monitor nodes for proficiency
student1.prof <- NetworkNodesInSet(student1,"Proficiency")

student1.t1a <- AdjoinNetwork(student1,EMTask1a)
stopifnot(setequal(student1$listNodes(),
 c("CliqueNode1", "Obs1a1", "Obs1a2", "Skill1", "Skill2", "Skill3")))

## We are done with the original EMTask1a now
DeleteNetwork(EMTask1a)

## Now add findings
CompileNetwork(student1)
NodeFinding(student1.t1a$Obs1a1) <- "Right"
NodeFinding(student1.t1a$Obs1a2) <- "Right"

student1.probt1a <- JointProbability(student1.prof)
## Done with the observables, absorb them
AbsorbNodes(student1.t1a)
```

```
stopifnot(setequal(student1$listNodes(),
 c("CliqueNode1", "Skill1", "Skill2", "Skill3")))

CompileNetwork(student1)
student1.probt1ax <- JointProbability(student1.prof)

## This should be the same
stopifnot(
  sum(abs(student1.probt1a-student1.probt1ax)) <.0001
)

## Now Task 2
student1.t2a <- AdjoinNetwork(student1,EMTask2a,as.IDname("t2a"))
stopifnot(
  setequal(names(student1.t2a),names(NetworkNodesInSet(student1,"t2a")))
)
stopifnot(setequal(student1$listNodes(),
 c("CliqueNode1", "Obs2a", "Skill1", "Skill2", "Skill3")))
DeleteNetwork(EMTask2a)

## Add findings
CompileNetwork(student1)
NodeFinding(student1.t2a$Obs2a) <- "Half"

student1.probt1a2a <- JointProbability(student1.prof)

AbsorbNodes(student1.t2a)
stopifnot(setequal(student1$listNodes(),
 c("CliqueNode1", "Skill1", "Skill2", "Skill3")))
CompileNetwork(student1)
student1.probt1a2ax <- JointProbability(student1.prof)

## This should be the same
stopifnot(
  sum(abs(student1.probt1a2a-student1.probt1a2ax)) <.0001
)

## Adjoining networks twice should result in copies with incremented
##   numbers.
AdjoinNetwork(student1,EMTask2b)
AdjoinNetwork(student1,EMTask2b)
stopifnot(setequal(student1$listNodes(),
 c("CliqueNode1", "Obs2b", "Obs2b1", "Skill1", "Skill2", "Skill3")))

DeleteNetwork(student1)
DeleteNetwork(EMTask2b)
DeleteNetwork(EMSMSystem)
stopSession(sess)
```

CalcNodeState *Calculates the state of a node based on logical functions or formulae*

Description

The expression `CalcNodeState(node)` will return the state of node if it is known deterministically, and `NA` if the exact value is not known. The expression `CalcNodeValue(node)` will return the numeric value of the node (e.g., the value set with `NodeLevels(node)`.

Usage

```
CalcNodeState(node)
CalcNodeValue(node)
```

Arguments

node	An active `NeticaNode` object that references the node.

Details

According to the Netica manual, the way that the value of node could be known absolutely is if it was set directly a call to `NodeFinding(node)` or `NodeValue(node)`, or if the value can be calculated exactly through logical conditional probability tables (i.e., ones with just 0's and 1's) or formula (see `NodeEquation()`.

The expression `CalcNodeState(node)` is appropriate when node is discrete, or has been discretized through a call to `NodeLevels(node)`. Otherwise it will generate an error.

The expression `CalcNodeValue(node)` is appropriate when node is continuous, or the states have been assigned numeric values through a call to `NodeLevels(node)`. Otherwise it will generate an error.

Value

The expression `CalcNodeState(node)` will return a character scalar giving the name of the current state of node if it can be determined, otherwise it will return `NA`.

The expression `CalcNodeValue(node)` will return a numeric scalar giving the name of the current value of node if it can be determined, otherwise it will return `NA`.

Warning

This function is not behaving at all like what I expected. In particular, it is returning `NA` in many cases where I expect it to produce a value. I've queried Norsys about this, but use with caution until I get a clarification.

Author(s)

Russell Almond

References

`http://norsys.com/onLineAPIManual/index.html`: CalcNodeState_bn(), CalcNodeValue_bn()

See Also

NodeFinding(),NodeLevels(),NodeValue(),IsNodeDeterministic(),NodeEquation(),is.continuous(), NodeExpectedValue()

Examples

```
sess <- NeticaSession()
startSession(sess)

lights <- CreateNetwork("lights", session=sess)
switchs <- NewDiscreteNode(lights,paste("Switch",1:2,sep=""),c("Up","Down"))
bulb <- NewDiscreteNode(lights,"Bulb",c("On","Off"))

## Set up a two-way switch (Xor) network
AddLink(switchs[[1]],bulb)
AddLink(switchs[[2]],bulb)
## This sets up a logical table, so that the light is on iff
## both switches are in the same orientation.
bulb[] <-"Off"
bulb[Switch1="Up",Switch2="Up"]<-"On"
bulb[Switch1="Down",Switch2="Down"]<-"On"
switchs[[1]][] <- .5
switchs[[2]][] <- .5


CompileNetwork(lights)

## Bulb is a deterministic node.
stopifnot(IsNodeDeterministic(bulb))

## value of node is unknown, returns NA
stopifnot(is.na(CalcNodeState(bulb)))

NodeFinding(switchs[[1]]) <- "Up"
NodeFinding(switchs[[2]]) <- "Up"

stopifnot(CalcNodeState(switchs[[1]])=="Up")

stopifnot(CalcNodeState(bulb)=="On")

NodeLevels(bulb) <-c(1,0)
NodeLevels(switchs[[1]]) <-c(1,0)
NodeLevels(switchs[[2]]) <-c(1,0)

## I expect both of these to return 1, but they return NA
CalcNodeValue(bulb)
CalcNodeValue(switchs[[1]])

DeleteNetwork(lights)

stopSession(sess)
```

`CaseFileDelimiter` *Gets or sets special characters for case files.*

Description

The function `CaseFileDelimiter` sets the field delimiter used when writing case files. The function `CaseFileMissingCode` sets the character code used for missing values in case files. If called with a null argument, then the current value is returned.

Usage

```
CaseFileDelimiter(newdelimiter = NULL, session=getDefaultSession())
CaseFileMissingCode(newcode = NULL, session=getDefaultSession())
```

Arguments

`newdelimiter`	A character scalar containing the new delimiter. It must be either a comma, a space, or a tab.
`session`	An object of type `NeticaSession` which defines the reference to the Netica workspace.
`newcode`	The character to be used as a delimiter. It must be either an asterisk ("*"), a question mark ("?"), a space, (" ") or the empty string ("").

Details

Case files are essentially a comma separated value files, although tab and space are allowed as alternative delimiters. The space and empty string are only allowed as missing value codes when the delimiter is a comma.

The value of the delimiter is global, and applies to all case files written from this point on.

When the argument is null (the default) the current value is returned without changing it.

Value

The value of the delimiter or missing code before the function call as a string.

Note

The default R missing code "NA" does not work with Netica.

Author(s)

Russell G. Almond

References

http://norsys.com/onLineAPIManual/index.html: SetCaseFileDelimChar_ns(), SetMissingDataChar_ns()

See Also

WriteFindings, WriteFindings, read.CaseFile, CaseStream

Examples

```
sess <- NeticaSession()
startSession(sess)

defaultDelim <- CaseFileDelimiter(session=sess) # Get default
d1 <- CaseFileDelimiter("\t", session=sess)
d2 <- CaseFileDelimiter(" ", session=sess)
d3 <- CaseFileDelimiter(",", session=sess)

defaultMiss <- CaseFileMissingCode(session=sess) # Get default
m1 <- CaseFileMissingCode("*", session=sess)
m2 <- CaseFileMissingCode("?", session=sess)
m3 <- CaseFileMissingCode(" ", session=sess)
m4 <- CaseFileMissingCode("", session=sess)
## Not run:
 ## This should throw an error.
 CaseFileDelimiter(" ", session=sess)

## End(Not run)

m5 <- CaseFileMissingCode("?", session=sess)

d4<- CaseFileDelimiter(" ", session=sess)
## Not run:
  ## This should throw an error
  CaseFileMissingCode(" ", session=sess)

## End(Not run)
## But this is okay
CaseFileMissingCode("*", session=sess)

stopifnot(d1==defaultDelim, d2=="\t", d3==" ", d4==",")
stopifnot(m1==defaultMiss, m2=="*", m3=="?", m4==" ", m5=="")

## restore defaults
CaseFileDelimiter(defaultDelim, session=sess)
CaseFileMissingCode(defaultMiss, session=sess)

stopSession(sess)
```

CaseFileStream *A stream of cases for reading/writing Netica findings to a file*

Description

This is the constructor for FileCaseStream objects which provide a wrapper around a Netica stream which is used to read/write cases. In this subclass, the case stream is associated with a Netica case file ('.cas' extension). The function ReadFindings reads the findings from the stream and the function WriteFindings writes them out.

Usage

```
CaseFileStream(pathname, session=getDefaultSession())
is.CaseFileStream(x)
getCaseStreamPath(stream)
```

Arguments

pathname	A character scalar giving a path to the case file. Netica expects case files to end with the extension ".cas"
session	An object of type NeticaSession which defines the reference to the Netica workspace.
stream	A CaseFileStream object.
x	A object to be printed or whose type is to be determined.

Details

A FileCaseStream object is a subclass of the CaseStream object, which is an R wrapper around a Netica stream object, in this case one that reads or writes to a case file. Case files are tab (or comma, see CaseFileDelimiter) separated value files where columns represent variables and rows represent cases. Although the function WriteFindings always appends a new case to the end of a file (and hence does not need to keep the stream object open between calls), the function ReadFindings will read (by default) sequentially from the cases in the stream, and hence the stream needs to be kept open between calls.

The function CaseFileStream will open a stream in Netica and create a new FileCaseStream if necessary. The argument pathname should be the pathname of the case file in the file system. This file should be a file previously written by WriteFindings or be in the same format. The delimiter used should be the one given by CaseFileDelimiter, and the code used for missing values should be the value of CaseFileMissingCode.

The function CloseCaseStream closes an open case stream (and is harmless if the stream is already closed). Although RNetica tries to close open case streams when they are garbage collected, users should not count on this behavior and should close them manually. Also be aware that all case streams are automatically closed when R is closed or RNetica is unloaded. The function isCaseStreamOpen tests to see if the stream is open or closed, and the function OpenCaseStream reopens a previously closed case stream.

The functions getCaseStreamPath returns the path on which the FileCaseStream is focused.

Value

The function CaseFileStream returns a new, open FileCaseStream object.

The functions is.CaseFileStream returns a logical value indicating whether or not the argument is a CaseFileStream.

The function getCaseStreamPath returns a string giving the path of the file associated with stream, or NULL if the argument is not a CaseFileStream.

Note

Internally, a weak reference system is used to keep a list of Netica stream objects which need to be closed when RNetica is unloaded. Stream objects should also be forced closed when garbage collected. The weak reference system is somewhat experimental, so well designed code should manually close the streams when the program is through with it.

Stream objects are fragile, and will not survive saving and restoring an R session. However, the object retains information about itself, so that calling OpenCaseStream on the saved object, should reopen the stream. Note that any position information will be lost.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: NewFileStream_ns(), http://homepage.stat.uiowa.edu/~luke/R/references/weakfinex.html

See Also

See FileCaseStream for properties of file case stream objects and CaseStream for general properties of Netica streams.

CaseFileDelimiter, CaseFileMissingCode, CaseMemoryStream, WriteFindings, ReadFindings,

Examples

```
sess <- NeticaSession()
startSession(sess)

abc <- CreateNetwork("ABC", sess)
A <- NewDiscreteNode(abc,"A",c("A1","A2","A3","A4"))
B <- NewDiscreteNode(abc,"B",c("B1","B2","B3"))
C <- NewDiscreteNode(abc,"C",c("C1","C2"))

AddLink(A,B)
AddLink(A,C)
AddLink(B,C)

## Outputfilename
casefile <- tempfile("testcase",fileext=".cas")
```

```
filestream <- CaseFileStream(casefile, session=sess)
stopifnot(is.CaseFileStream(filestream),
          isCaseStreamOpen(filestream))

## Case 1
NodeFinding(A) <- "A1"
NodeFinding(B) <- "B1"
NodeFinding(C) <- "C1"
filestream <- WriteFindings(list(A,B,C),filestream,1001,1.0)
stopifnot(getCaseStreamLastId(filestream)==1001,
          abs(getCaseStreamLastFreq(filestream)-1.0) <.0001)

## Close it
filestream <- CloseCaseStream(filestream)
stopifnot (is.CaseFileStream(filestream),
           !isCaseStreamOpen(filestream))

## Reopen it
filestream <- OpenCaseStream(filestream)
stopifnot (is.CaseFileStream(filestream),
           isCaseStreamOpen(filestream))

##Case 1
RetractNetFindings(abc)
filestream <- ReadFindings(list(A,B,C),filestream,"FIRST")
stopifnot(getCaseStreamLastId(filestream)==1001,
          abs(getCaseStreamLastFreq(filestream)-1.0) <.0001)

##Clean Up
CloseCaseStream(filestream)
DeleteNetwork(abc)
stopSession(sess)
```

`CaseMemorytream` *A stream of cases for reading/writing Netica from memory*

Description

This object is subclass of `CaseStream` so it is a wrapper around a Netica stream which is used to read/write cases. In this subclass, the case stream is associated with a memory buffer that corresponds to an R `data.frame` object. The function `MemoryStreamContents` accesses the contents as a data frame.

Usage

```
CaseMemoryStream(data.frame, label=deparse(substitute(data.frame)), session=getDefaultSession())
is.MemoryCaseStream(x)
getCaseStreamDataFrameName(stream)
```

Arguments

data.frame	A data frame in which columns correspond to Netica nodes, and rows correspond to cases. See details.
label	A name for the stream object.
session	An object of type NeticaSession which defines the reference to the Netica workspace.
stream	A CaseStream object.
x	A object whose type is to be determined.

Details

A Netica case file has a format that very much resembles the output of write.table. The first row is a header row, which contains the names of the variables, the second and subsequent rows contain a set of findings: an assignment of values to the nodes indicated in the columns. There are no row numbers, and the separator and missing value codes are the values of CaseFileDelimiter(), and CaseFileMissingCode() respectively.

In addition to columns representing variables, two special columns are allowed. The column named "IDnum", if present should contain integers which correspond to ID numbers for the cases (this correspond to the id argument of WriteFindings). The column named "NumCases" should contain number values and this allows rows to be differentially weighted (this correspond to the freq argument of WriteFindings).

A simple way to convert a data frame into a set of cases for use with various Netica functions that use cases would be to write the data frame to a file of the proper format, and then create a CaseFileStream on the just written file. The MemoryCaseStream shortcuts that process by writing the data frame to a memory buffer and then creating a stream around the memory buffer. Like the CaseFileStream, the MemoryCaseStream is a subclass of CaseStream and follows the same conventions.

The function MemoryCaseStream opens a new memory stream using data.frame as the source. If data.frame is NULL a new memory stream for writing is created. The function CloseCaseStream closes an open case stream (and is harmless if the stream is already closed. Although RNetica tries to close open case streams when they are garbage collected, users should not count on this behavior and should close them manually. Also be aware that all case streams are automatically closed when R is closes or RNetica is unloaded. The function isCaseStreamOpen tests to see if the stream is open or closed. The function OpenCaseStream if called on a closed MemoryCaseStream will reopen the stream in Netica using the current value of MemoryStreamContents as the source. (If called on an open stream it will do nothing but issue a warning).

The function getCaseStreamDataFrameName provides the value of label when the stream was created.

Value

The function OpenMemoryCaseStream returns a new, open CaseFileStream object.

The functions is.MemoryCaseStream returns a logical value indicating whether or not the argument is a CaseFileStream.

The function getCaseStreamDataFrameName returns the value of label used when the stream was created, usually this is the name of the data.frame argument.

Netica Bugs

In version 5.04 of the Netica API, there is a problem with using Memory Streams that seems to affect the functions LearnCases and LearnCPTs. Until this problem is fixed, most uses of Memory Streams will require file streams instead. Write the case file using write.CaseFile, and then create a file stream using CaseFileStream.

Note

In version 0.5 of RNetica, this class was renamed. It is now called MemoryCaseStream and the constructor is called CaseMemoryStream (while previously the class and the filename had the same name). This matches the usage of FileCaseStream and its constructor CaseFileStream.

MemoryCaseStreams are most useful for small to medium size data frames. Larger data frames are probably better handled through case files.

Internally, a weak reference system is used to keep a list of Netica stream objects which need to be closed when RNetica is unloaded. Stream objects should also be forced closed when garbage collected. The weak reference system is somewhat experimental, so well designed code should manually close the streams when the program is through with it.

Stream objects are fragile, and will not survive saving and restoring an R session. However, the object retains information about itself, so that calling OpenCaseStream on the saved object, should reopen the stream. Note that any position information will be lost.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: NewMemoryStream_ns(), http://homepage.stat.uiowa.edu/~luke/R/references/weakfinex.html

See Also

CaseFileDelimiter, CaseFileMissingCode, WriteFindings, ReadFindings, MemoryStreamContents, CaseStream

Examples

```
sess <- NeticaSession()
startSession(sess)

abc <- CreateNetwork("ABC", session=sess)
A <- NewDiscreteNode(abc,"A",c("A1","A2","A3","A4"))
B <- NewDiscreteNode(abc,"B",c("B1","B2","B3"))
C <- NewDiscreteNode(abc,"C",c("C1","C2"))

AddLink(A,B)
AddLink(A,C)
AddLink(B,C)

## This is the file written in CaseFileStream help.
```

```
casefile <- file.path(library(help="RNetica")$path,
                          "testData","abctestcases.cas")
CaseFileDelimiter("\t", session=sess)
CaseFileMissingCode("*", session=sess)
cases <- read.CaseFile(casefile, session=sess)

memstream <- CaseMemoryStream(cases, session=sess)


##Case 1
memstream <- ReadFindings(list(A,B,C),memstream,"FIRST")
stopifnot(NodeFinding(A) == "A1",
          NodeFinding(B) == "B1",
          NodeFinding(C) == "C1",
          getCaseStreamLastId(memstream)==1001,
          abs(getCaseStreamLastFreq(memstream)-1.0) <.0001)

##Case 2
memstream <- ReadFindings(list(A,B,C),memstream,"NEXT")
stopifnot(NodeFinding(A) == "A2",
          NodeFinding(B) == "B2",
          NodeFinding(C) == "C2",
          getCaseStreamLastId(memstream)==1002,
          abs(getCaseStreamLastFreq(memstream)-2.0) <.0001)

##Case 3
memstream <- ReadFindings(list(A,B,C),memstream,"NEXT")
stopifnot(NodeFinding(A) == "A3",
          NodeFinding(B) == "B3",
          NodeFinding(C) == "@NO FINDING",
          getCaseStreamLastId(memstream)==1003,
          abs(getCaseStreamLastFreq(memstream)-1.0) <.0001)

## EOF
memstream <- ReadFindings(list(A,B,C),memstream,"NEXT")
stopifnot (is.na(getCaseStreamPos(memstream)))


##Clean Up
CloseCaseStream(memstream)
DeleteNetwork(abc)
stopSession(sess)
```

CaseStream-class *Class* "CaseStream"

Description

This object is a wrapper around a Netica stream which is used to read/write cases—sets of findings entered into a Netica network. There are two subclasses: FileCaseStream and MemoryCaseStream.

The function ReadFindings reads the findings from the stream and the function WriteFindings writes them out.

Details

A CaseStream object is an R wrapper around a Netica stream object. There are two subclasses: FileCaseStream objects are streams focused on a case file, and MemoryCaseStream objects are streams focused on a hunk of memory corresponding to an R data frame object.

Although the function WriteFindings always appends a new case to the end of a file (and hence does not need to keep the stream object open between calls), the function ReadFindings will read (by default) sequentially from the cases in the stream, and hence the stream needs to be kept open between calls.

The functions CaseFileStream and CaseMemoryStream create new streams and open them. The function OpenCaseStream will reopen a previously closed stream, and will issue a warning if the stream is already open. The function CloseCaseStream closes an open case stream (and is harmless if the stream is already closed). Although RNetica tries to close open case streams when they are garbage collected, users should not count on this behavior and should close them manually. Also be aware that all case streams are automatically closed when R is closes or RNetica is unloaded. The function isCaseStreamOpen tests to see if the stream is open or closed. The function WithOpenCaseStream executes an arbitrary R expression in a context where the stream is open, and then closed afterwards.

Netica internally keeps track of the current position of the stream when it is read or written. The functions getCaseStreamPos, getCaseStreamLastId and getCaseStreamLastFreq get information about the position in the file, the user generated id number and the frequency/weight assigned to the case at the time the stream was last read or written. In particular, the function ReadFindings returns a CaseStream object, which should be queried to find the ID and Frequencies read from the stream. When ReadFindings reaches the end of the stream, the value of getCaseStreamPos(*stream*) will be NA.

Extends

All reference classes extend and inherit methods from "envRefClass". Note that because this is a reference class unlike traditional S3 and S4 classes it can be destructively modified. Also fields (slots) are accessed using the '$' operator.

Fields

Note these should be regarded as read-only from user code.

Name: Object of class character an identifier for the case stream, derived from the filename for FileCaseStream objects, and from the name of the R object for MemoryCaseStream

Session: Object of class NeticaSession:: a back pointer to the NeticaSession object in which the stream was created.

Netica_Case_Stream: Object of class externalptr a link to the stream in internal Netica memory.

Case_Stream_Position: Object of class integer the number of the last read/writen record. This is NA if the end of the file has been reached.

`Case_Stream_Lastid`: Object of class `integer` the ID number of the last read/written record.

`Case_Stream_Lastfreq`: Object of class `numeric` giving the frequence of the last read/written record. This is used as a weight in learning applications.

Methods

`show()`: Provides a printed record.

`close()`: Closes the stream. Equivalent to `CloseCaseStream(`*stream*`)`.

`isOpen()`: Checks to see if the stream is currently open. Equivalent to `isCaseStreamOpen(`*stream*`)`.

`isActive()`: Equivalent to `isOpen()`, name is symmetric with other Netica reference objects.

`clearErrors(severity)`: Calls `clearErrors` on the `Session` object.

`reportErrors(maxreport, clear)`: Calls `reportErrors` on the `Session` object.

`initialize(Name, Session, ...)`: Internal initializer. User code should not call.

Note

The functions `ReadNetworks` and `WriteNetworks` also use Netica streams internally. However, as it is almost certainly a mistake to keep the stream open after the network has been read or written, no `NeticaCaseStream` object is created.

Internally, a weak reference system is used to keep a list of Netica stream objects which need to be closed when RNetica is unloaded. Stream objects should also be forced closed when garbage collected. The weak reference system is somewhat experimental, so well designed code should manually close the streams when the program is through with them.

Stream objects are fragile, and will not survive saving and restoring an R session. However, the object retains information about itself, so that calling `OpenCaseStream` on the saved object, should reopen the stream. Note that any position information will be lost.

Author(s)

Russell Almond

References

`http://norsys.com/onLineAPIManual/index.html`: NewFileStream_ns(),NewMemoryStream_ns(), DeleteStream_ns() `http://homepage.stat.uiowa.edu/~luke/R/references/weakfinex.html`

See Also

See `FileCaseStream` and `MemoryCaseStream` for specific details about the two subtypes. `CaseMemoryStream` and `CaseFileStream` are the two constructors.

`OpenCaseStream`, `CaseFileDelimiter`, `CaseFileMissingCode`, `WriteFindings`, `ReadFindings`, `WithOpenCaseStream`

Examples

```
sess <- NeticaSession()
startSession(sess)

abc <- CreateNetwork("ABC",sess)
A <- NewDiscreteNode(abc,"A",c("A1","A2","A3","A4"))
B <- NewDiscreteNode(abc,"B",c("B1","B2","B3"))
C <- NewDiscreteNode(abc,"C",c("C1","C2"))

AddLink(A,B)
AddLink(A,C)
AddLink(B,C)

## Outputfilename
casefile <- tempfile("testcase",fileext=".cas")

filestream <- CaseFileStream(casefile,sess)
stopifnot(is.NeticaCaseStream(filestream),
          isCaseStreamOpen(filestream))

## Case 1
NodeFinding(A) <- "A1"
NodeFinding(B) <- "B1"
NodeFinding(C) <- "C1"
filestream <- WriteFindings(list(A,B,C),filestream,1001,1.0)
stopifnot(getCaseStreamLastId(filestream)==1001,
          abs(getCaseStreamLastFreq(filestream)-1.0) <.0001)
pos1 <- getCaseStreamPos(filestream)
RetractNetFindings(abc)

## Case 2
NodeFinding(A) <- "A2"
NodeFinding(B) <- "B2"
NodeFinding(C) <- "C2"
## Double weight this case
filestream <- WriteFindings(list(A,B,C),filestream,1002,2.0)
pos2 <- getCaseStreamPos(filestream)
stopifnot(pos2>pos1,getCaseStreamLastId(filestream)==1002,
          abs(getCaseStreamLastFreq(filestream)-2.0) <.0001)
RetractNetFindings(abc)

## Case 3
NodeFinding(A) <- "A3"
NodeFinding(B) <- "B3"
## C will be missing
filestream <- WriteFindings(list(A,B,C),filestream,1003,1.0)
stopifnot(getCaseStreamLastId(filestream)==1003,
          abs(getCaseStreamLastFreq(filestream)-1.0) <.0001)
RetractNetFindings(abc)

## Close it
```

```
filestream <- CloseCaseStream(filestream)
stopifnot (is.NeticaCaseStream(filestream),
           !isCaseStreamOpen(filestream))

## Reopen it
filestream <- OpenCaseStream(filestream)
stopifnot (is.NeticaCaseStream(filestream),
           isCaseStreamOpen(filestream))

##Case 1
RetractNetFindings(abc)
filestream <- ReadFindings(list(A,B,C),filestream,"FIRST")
pos1a <- getCaseStreamPos(filestream)
stopifnot(pos1a==pos1,
          getCaseStreamLastId(filestream)==1001,
          abs(getCaseStreamLastFreq(filestream)-1.0) <.0001)

##Case 2
RetractNetFindings(abc)
filestream <- ReadFindings(list(A,B,C),filestream,"NEXT")
stopifnot(getCaseStreamPos(filestream)==pos2,
          getCaseStreamLastId(filestream)==1002,
          abs(getCaseStreamLastFreq(filestream)-2.0) <.0001)


##Clean Up
CloseCaseStream(filestream)
CloseCaseStream(filestream) ## This should issue a warning but be
## harmless.
DeleteNetwork(abc)
stopSession(sess)
```

---

cc *Concatenates lists without stripping attributes*

Description

OBSOLETE: This function was removed starting with RNetica 0.5 (it existed briefly as a workaround for a solution that change the object representation was the correct solution for.)

The base R function `c()` strips the attributes off of objects (particularly `NeticaNode` and `NeticaBN` objects). The function `cc` is a replacement which does not do that stripping.

Usage

```
cc(...)
## S3 method for class 'NeticaNode'
c(...)
## S3 method for class 'NeticaBN'
c(...)
```

Arguments

...	A list of objects. Generally it should be either NeticaNode or NeticaBN objects or lists of such objects.

Details

The base R c() function strips attributes from objects. For NeticaNode and NeticaBN objects, this removes the attributes that link the name to the Netica object and leaves just a string. This "feature" of S has been around since the days of the Blue Book and there is probably code that relies on this unexpected behavior.

The cc() function works around this by copying the arguments one at a time into a new list (slower but safer). Arguments which satisfy is(arg,"list") are treated as lists and add length(arg) elements to the lsit. All other arguments are treated as essentially lists of length 1, and the value is inserted in the appropriate place in the list.

The methods for NeticaNode and NeticaBN fix the c() function (which is generic) if the first argument is a singleton. Thus c(newNode,nodeList) and cc(newNode,nodeList) are identical. Note that these only fix half of the problem; c(nodeList,newNode) still calls the default method (or the method for lists) which strips the attributes of newNode. Instead use cc(nodeList,newNode) or c(nodeList,list(newNode))

Value

A list containing all of the values in the arguments. If there is a single, non-list argument, it will return a list with one element.

Author(s)

Russell Almond

See Also

NeticaNode,NeticaBN

Examples

```
## Not run:
anet <- CreateNetwork("anet")

nodeList <- NewDiscreteNode(anet,paste("oldNode",1:3,sep=""))
newNode <- NewDiscreteNode(anet,"newNode")

l1 <- c(newNode,nodeList)               #A list of nodes
stopifnot(is.list(l1),length(l1)==4L,sapply(l1,is.NeticaNode))

l2 <- c(nodeList,newNode)               #Doesn't work!!!
stopifnot(!all(sapply(l2,is.NeticaNode)))

l2a <- cc(nodeList,newNode)               #Does work!!!
stopifnot(all(sapply(l2a,is.NeticaNode)))
```

```
l2b <- c(nodeList,list(newNode))         #As does this
stopifnot(all(sapply(l2b,is.NeticaNode)))

l3 <- c(newNode)                        #List with one element
stopifnot(is.list(l3),length(l3)==1L,sapply(l3,is.NeticaNode))

l4 <- c(newNode,nodeList[[1]],nodeList[[3]])
stopifnot(is.list(l4),length(l4)==3L,sapply(l4,is.NeticaNode))

l5 <- c(newNode,nodeList[2:3],nodeList[[1]])
stopifnot(is.list(l5),length(l5)==4L,sapply(l5,is.NeticaNode))

## End(Not run)
```

`CliqueNode-class` *Class* `"CliqueNode"`

Description

A dummy node used to force it parents into the same clique in the junction tree. In particular, the node has a single state but its parents are listed in its `clique` field.

Extends

Class `"NeticaNode"`, directly.

All reference classes extend and inherit methods from `"envRefClass"`. Note that because this is a reference class unlike traditional S3 and S4 classes it can be destructively modified. Also fields (slots) are accessed using the '$' operator.

Methods

toString `signature(x = "CliqueNode")`: Provides a pretited representation.

Fields

Note these should be regarded as read-only from user code.

`Name`**:** Object of class `character` giving the Netica name of the node. Must follow the `IDname` rules.

`Netica_Node`**:** Object of class `externalptr` giving the address of the node in Netica's memory space.

`Net`**:** Object of class `NeticaBN`, a back reference to the network in which this node resides.

`discrete`**:** Always `TRUE` for clique nodes.

`clique`**:** A list of `NeticaNode` objects which are the parents of the clique node.

Class-Based Methods

`show()`: Prints a description of the node.

`initialize(..., clique)`: Internal initializer, should not be called directly by user code. Use `MakeCliqueNode` instead.

The following methods are inherited (from the `NeticaNode`): deactivate ("NeticaNode"), isActive ("NeticaNode"), show ("NeticaNode"), clearErrors ("NeticaNode"), reportErrors ("NeticaNode"), initialize ("NeticaNode")

Note

Clique nodes only last for the R session that was used to create them. After that, they will appear like ordinary nodes. They will still be present in the network, but the special `"clique"` attribute will be lost.

Currently Netica only allows virtual evidence at the node level (`NodeLikelihood()`). I'm lobbying to get Netica to support it at the clique level as well. At which point, this function becomes extremely useful.

Author(s)

Russell Almond

References

Almond, R. G. & Mislevy, R. J. (1999) Graphical models and computerized adaptive testing. *Applied Psychological Measurement*, 23, 223-238.

Almond, R., Herskovits, E., Mislevy, R. J., & Steinberg, L. S. (1999). Transfer of information between system and evidence models. In Artificial Intelligence and Statistics 99, Proceedings (pp. 181–186). Morgan-Kaufmann

`http://norsys.com/onLineAPIManual/index.html`: See the NeticaEx function `FormCliqueWith` is the documentation for JointProbability_bn()

See Also

`MakeCliqueNode()`, `NeticaNode`, `JointProbability()`, `AddLink()`, `JunctionTreeReport()`

Examples

```
sess <- NeticaSession()
startSession(sess)

EMSMSystem <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","System.dne"), session=sess)

CompileNetwork(EMSMSystem)
## Note that Skill1 and Skill2 are in different cliques
JunctionTreeReport(EMSMSystem)

Skills12 <- NetworkFindNode(EMSMSystem,c("Skill1","Skill2"))
```

```
cn <- MakeCliqueNode(Skills12)
cnclique <- GetClique(cn)

stopifnot(
  is.CliqueNode(cn),
  setequal(sapply(cnclique,NodeName),sapply(Skills12,NodeName))
)

CompileNetwork(EMSMSystem)
## Note that Skill1 and Skill2 are in different cliques
JunctionTreeReport(EMSMSystem)

DeleteNodes(cn) ## This clears the clique.

DeleteNetwork(EMSMSystem)
stopSession(sess)
```

`CompileNetwork` *Builds the junction tree for a Netica Network*

Description

Before Netica performs inference in a network, it needs to compile the network. This process consists of building a junction tree and conditional probability tables for the nodes of that tree. The function `CompileNetwork()` compiles the network and `UncompileNetwork()` undoes the compilation and frees the associated memory.

Usage

```
CompileNetwork(net)
UncompileNetwork(net)
is.NetworkCompiled(net)
```

Arguments

`net`	An active `NeticaBN` which will be compiled.

Details

Usually Bayesian network projects operate in two phases. In the construction phase, new nodes are added to the network, new connections made and conditional probability tables are set.

In the inference phase, findings are added to nodes and other nodes are queried about their current conditional probability tables.

The functions `CompileNetowrk()` and `UncompileNetwork()` move the networks between the two phases. The documentation for `EliminationOrder()` and `JunctionTreeReport()` provide more details about the compilation process. The function `NetworkCompiledSize()` provides information about the amount of storage used by the compiled network, but only after the network is compiled.

The function `is.NetworkCompiled()` tests to see if a network is compiled or not.

Value

The NeticaBN object net is returned invisibly.

Warning

I'm currently observing a bug that occurs under Windows if not all of the nodes have their CPTs set. Under Linux the function exhibits the expected behavior: It generates a warning about the unset CPTs and enters a uniform distribution for each one. Under Windows it reports the warning, but then generates an error "GetError_ns: deleted or damage report_ns passed". It is unclear if this a problem in Netica or RNetica.

To work around, simply set all tables before compiling.

Note

Calling NetworkCompiledSize() on an uncompiled network produces, an error, but also the sensible value of -1. The function is.NetworkCompiled() calls the same internal function as NetworkCompiledSize, but clears the error. This means it also clears any other errors that might be lurking in the system (see ReportErrors()).

I think calling CompileNetwork() twice on the same network is harmless. Adding a node to a network will automatically uncompile it.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: CompileNet_bn(), UncompileNet_bn(), SizeCompiledNet_bn(),

See Also

NeticaBN, HasNodeTable(), NodeFinding(), NodeBeliefs(), EliminationOrder(), JunctionTreeReport(), JointProbability(), MostProbableConfig(), FindingsProbability()

Examples

```
sess <- NeticaSession()
startSession(sess)

irt5 <- ReadNetworks(file.path(library(help="RNetica")$path,
                               "sampleNets","IRT5.dne"), session=sess)
stopifnot (!is.NetworkCompiled(irt5))

CompileNetwork(irt5) ## Ready to enter findings
stopifnot (is.NetworkCompiled(irt5))

UncompileNetwork(irt5) ## Ready to add more nodes
stopifnot (!is.NetworkCompiled(irt5))
```

```
DeleteNetwork(irt5)
stopSession(sess)
```

CopyNetworks *Makes copies of Netica networks.*

Description

Makes a copy of the networks in the list nets giving them the names in newnamelist. The options argument controls how much information is copied.

Usage

```
CopyNetworks(nets, newnamelist, options = character(0))
```

Arguments

nets	A list of NeticaBN objects.
newnamelist	A character vector of the same length as nets which gives the names for the newly created copies.
options	A character vector containing information about what to copy. The elements should be one of the values "no_nodes", "no_links", "no_tables", "no_visual".

Details

Copies each of the networks in the nets lists, giving it a new name from the newnamelist. It returns a list of the new networks. If the specified net does not exist, then a warning is issued and a NULL is returned instead of the corresponding NeticaBN object.

The options argument is passed to the options argument of the Netica API function CopyNet_bn(). Meanings for the various arguments can be found in the documentation for that function. Note that Netica expects a list of comma separated values. RNetica will collapse the options argument into a comma separated list, so the argument can be given either as a character vector of length 1 containing a comma separated list, or the elements of that list in separate elements of a character vector.

Value

A list of NeticaBN objects corresponding to the new networks, or if the length of nets is one, a single NeticaBN object is returned instead. A NULL is returned instead of the NeticaBN object if the corresponding element of nets does not exit.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: CopyNet_bn()

See Also

DeleteNetwork()

Examples

```
sess <- NeticaSession()
startSession(sess)

net1 <- CreateNetwork("Original", session=sess)
nets <- CreateNetwork(paste("Original",2:3,sep=""), session=sess)

copy1 <-CopyNetworks(net1,"Copy1")
stopifnot(is(copy1,"NeticaBN"))
stopifnot(copy1$Name == "Copy1")
stopifnot(copy1 != net1)

netc <- CopyNetworks(nets,paste("Copy",2:3,sep=""))
stopifnot(all(sapply(netc,is,"NeticaBN")))
stopifnot(netc$Name == c("Copy2","Copy3"))

DeleteNetwork(c(netc,nets,list(copy1,net1)))
stopSession(sess)
```

CopyNodes *Copies or duplicates nodes in a Netica network.*

Description

This function either copies nodes from one net to another or duplicates nodes within the same network.

Usage

```
CopyNodes(nodes, newnamelist = NULL, newnet = NULL, options = character(0))
```

Arguments

nodes	A list of active NeticaNode objects all from the same network.
newnamelist	If supplied, this should be character vector with the same length as nodes giving the new names for the nodes.
newnet	If supplied, it should be an active NeticaBN which is the destination for the new nodes. If this argument is NULL the nodes will be duplicated within the original network.

`options` A character vector of options, with each element being one of the options. Currently, the only supported options are `"no_tables"` (do not copy the conditional probability tables for the nodes) and `"no_links"` (do not duplicate the links, which implies do not copy tables).

Details

The nodes in the first argument will be copied into a new network as specified by `newnet`. If `newnet` is not specified or if it the same as the network from which `nodes` come, then the nodes will be duplicated instead of copied.

If the nodes are duplicated, then will be given new names. The default Netica behavior for new names is to append a number to the end of the node name, or to increment an existing number. If `newnamelist` is supplied, these names will be used instead of the add a number convention. Supplying `newnamelist` will change the names of the nodes when copying from one network to another.

When nodes are copied links going into the node are copied as well. Thus if there is a link `A -> B` in the network and `B` is copied into the same network, then there will a link `A -> B1` to the new node. If `B` is copied into a new network, the link will be there but not attached, as if `NodeParents(B1)[A] <- NULL` had been called.

The argument `options` allows control over what is copied. The currently supported options are:

- `"no_tables"` — The conditional probability tables of the nodes (see `NodeProbs()`) will not be copied, and new tables will need to be set in the new network.
- `"no_links"` — The links going into the `nodes` will not be copied. Note that the `"no_links"` option implies the `"no_tables"` option, so both do not need to be specified.

Value

A list containing the new nodes (or just the new node, if there is only one).

Note

There may be some information that is not copied. For example, the `NodeSets()` information is not copied.

Author(s)

Russell Almond

References

`http://norsys.com/onLineAPIManual/index.html`: CopyNodes_bn()

See Also

`CopyNetworks()`, `NeticaNode`, `NeticaBN()`, `NodeProbs()`, `NodeParents()`, `AbsorbNodes()`, `DeleteNodes()`

Examples

```
sess <- NeticaSession()
startSession(sess)
System <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","System.dne"), session=sess)

EMTask1a <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","EMTask1a.dne"), session=sess)

student1 <- CopyNetworks(System, "Student1")
student1.sysnodes <- NetworkAllNodes(student1)

student1.t1anodes <- CopyNodes(NetworkAllNodes(EMTask1a),newnet=student1)

## Copied, new nodes have the same names as the old nodes.
stopifnot(
  setequal(names(NetworkAllNodes(EMTask1a)),
           names(student1.t1anodes))
)

## The nodes in the evidence model have stub connections to the nodes in
## the system model.  Need to link them up.
stopifnot(
  any(sapply(NodeParents(student1.t1anodes[[1]]),NodeKind) == "Stub"),
  any(sapply(NodeParents(student1.t1anodes[[2]]),NodeKind) == "Stub")
)


student1.allnodes <- NetworkAllNodes(student1)
for (node in student1.t1anodes) {
  stubs <- sapply(NodeParents(node),NodeKind) == "Stub"
  NodeParents(node)[stubs] <- student1.allnodes[NodeInputNames(node)[stubs]]
}
stopifnot(
  sapply(NodeParents(student1.t1anodes[[1]]),NodeKind) != "Stub",
  sapply(NodeParents(student1.t1anodes[[2]]),NodeKind) !="Stub"
)

## Duplicate these nodes.
student1.t1xnodes <- CopyNodes(student1.t1anodes)

## Autonaming increments the numbers.
stopifnot(
  setequal(names(student1.t1xnodes),c("Obs1a3","Obs1a4"))
)

## Duplicate and rename.
student1.t1cnodes <- CopyNodes(student1.t1anodes,c("Obs1c1","Obs1c2"))

stopifnot(
  setequal(names(student1.t1cnodes),c("Obs1c1","Obs1c2"))
)
```

```
## Duplicated nodes have real not stub connections.
stopifnot(
  sapply(NodeParents(student1.t1cnodes[[1]]),NodeKind) != "Stub",
  sapply(NodeParents(student1.t1cnodes[[2]]),NodeKind) !="Stub"
)


DeleteNetwork(list(System,student1,EMTask1a))
stopSession(sess)
```

CPA *Representation of a conditional probability table as an array.*

Description

A conditional probability table for a node can be represented as a array with the first p dimensions representing the parent variables and the last dimension representing the states of the node. Given a set of values for the parent variables, the values in the last dimension contain the conditional probabilities corresponding conditional probabilities. A CPA is a special array object which represents a conditional probability table.

Usage

```
is.CPA(x)
as.CPA(x)
```

Arguments

x	Object to be tested or coerced into a CPA.

Details

One way to store a conditional probability table is as an array in which the first p dimensions represent the parent variables, and the $p + 1$ dimension represents the child variable. Here is an example with two parents variables, A and B, and a single child variable, C:

```
, , C=c1
```

	b1	b2	b3
a1	0.07	0.23	0.30
a2	0.12	0.25	0.31
a3	0.17	0.27	0.32
a4	0.20	0.29	0.33

```
, , C=c2
```

	b1	b2	b3

a1	0.93	0.77	0.70
a2	0.88	0.75	0.69
a3	0.83	0.73	0.68
a4	0.80	0.71	0.67

[Because R stores (and prints) arrays in column-major order, the last value (in this case tables) is the one that sums to 1.]

The CPA class is a subclass of the array class (formally, it is class c("CPA","array")). The CPA class interprets the dimnames of the array in terms of the conditional probability table. The first p values of names(dimnames(x)) are the input names of the edges (see NodeInputNames() or the variable names (or the parent variable, see NodeParents(), if the input names were not specified), and the last value is the name of the child variable. Each of the elements of dimnames(x) should give the state names (see NodeStates()) for the respective value. In particular, the conversion function as.CPF() relies on the existence of this meta-data, and as.CPA() will raise a warning if an array without the appropriate dimnames is supplied.

Although the intended interpretation is that of a conditional probability table, the normalization constraint is not enforced. Thus a CPA object could be used to store likelihoods, probability potentials, contingency table counts, or other similarly shaped objects. The function normalize scales the values of a CPA so that the normalization constraint is enforced.

The method NodeProbs() returns a CPA object.

The function as.CPA() is designed to convert between CPFs (that is, conditional probability tables stored as data frames) and CPAs. It assumes that the factors variables in the data frame represent the parent variables, and the numeric values represent the states of the child variable. It also assumes that the names of the numeric columns are of the form *varname*.*state*, and attempts to derive variable and state names from that.

If the argument to as.CPA(x) is an array, then it assumes that the dimnames(x) and names(dimnames(x)) are set to the states of the variables and the names of the variables respectively. A warning is issued if the names are missing.

Value

The function is.CPA() returns a logical value indicating whether or not the is(x,"CPA") is true.

The function as.CPA returns an object of class c("CPA","array"), which is essentially an array with the dimnames set to reflect the variable names and states.

Note

The obvious way to print a CPA would be to always show the child variable as the rows in the individual tables, with the parents corresponding to rows and tables. R, however, internally stores arrays in column-major order, and hence the rows in the printed tables always correspond to the second dimension. A new print method for CPA would be nice.

This is an S3 object, as it just an array with a special interpretation.

Author(s)

Russell Almond

See Also

NodeProbs(), Extract.NeticaNode, CPF, normalize()

Examples

```
arf <- data.frame(A=rep(c("a1","a2"),each=3),
                  B=rep(c("b1","b2","b3"),2),
                  C.c1=1:6, C.c2=7:12, C.c3=13:18, C.c4=19:24)
arfa <- as.CPA(arf)
stopifnot(
  is.CPA(arfa),
  all(dim(arfa)==c(2,3,4))
)

arr1 <- array(1:24,c(4,3,2),
            dimnames=list(A=c("a1","a2","a3","a4"),B=c("b1","b2","b3"),
                          C=c("c1","c2")))
arr1a <- as.CPF(arr1)
stopifnot(
  is.CPA(as.CPA(arr1a))
)

## Not run:
  as.CPF(node[])

## End(Not run)
```

CPF *Representation of a conditional probability table as a data frame.*

Description

A conditional probability table for a node can be represented as a data frame with a number of factor variables representing the parent variables and the remaining numeric values representing the conditional probabilities of the states of the nodes given the parent configuration. Each row represents one configuration and the corresponding conditional probabilities. A CPF is a special data.frame object which represents a conditional probability table.

Usage

```
is.CPF(x)
as.CPF(x)
```

Arguments

x	Object to be tested or coerced into a CPF.

Details

One way to store a conditional probability table is a table in which the first several columns indicate the states of the parent variables, and the last several columns indicate probabilities for several child variables. Consider the following example:

	A	B	C.c1	C.c2	C.c3	C.c4
[1,]	a1	b1	0.03	0.17	0.33	0.47
[2,]	a2	b1	0.05	0.18	0.32	0.45
[3,]	a1	b2	0.06	0.19	0.31	0.44
[4,]	a2	b2	0.08	0.19	0.31	0.42
[5,]	a1	b3	0.09	0.20	0.30	0.41
[6,]	a2	b3	0.10	0.20	0.30	0.40

In this case the first two columns correspond to parent variables A and B. The variable A has two possible states and the variable B has three. The child variable is C and it has for possible states. The numbers in each row give the conditional probabilities for those states give the state of the child variables.

The class `CPF` is a subclass of `data.frame` (formally, it is class `c("CPF","data.frame")`). Although the intended interpretation is that of a conditional probability table, the normalization constraint is not enforced. Thus a `CPF` object could be used to store likelihoods, probability potentials, contingency table counts, or other similarly shaped objects. The function `normalize` scales the numeric values of `CPF` so that each row is normalized.

The `[` method for a `NeticaNode` returns a `CPF` (if the node is not deterministic).

The function `as.CPF()` is designed to convert between `CPA`s (that is, conditional probability tables stored as arrays) and `CPF`s. In particular, `as.CPF` is designed to work with the output of `NodeProbs()` or a similarly formatted array. It assumes that `names(dimnames(x))` are the names of the variables, and `dimnames(x)` is a list of character vectors giving the names of the states of the variables. (See `CPA` for details.) This general method should work with any numeric array for which both `dimnames(x)` and `names(dimnames(x))` are specified.

The argument `x` of `as.CPF()` could also be a data frame, in which case it is permuted so that the factor variable are first and the class tag `"CDF"` is added to its class.

Value

The function `is.CPF()` returns a logical value indicating whether or not the `is(x,"CDF")` is true.

The function `as.CPF` returns an object of class `c("CPF","data.frame")`, which is essentially a data frame with the first couple of columns representing the parent variables, and the remaining columns representing the states of the child variable.

Note

The parent variable list is created with a call `expand.grid(dimnames(x)[1:(p-1)])`. This produces conditional probability tables where the first parent variable varies fastest. The Netica GUI displays tables in which the last parent variable varies fastest.

Note, this is an S3 class, as it is basically a data.frame with special structure.

Author(s)

Russell Almond

See Also

NodeProbs(), Extract.NeticaNode, CPA, normalize()

Examples

```
arf <- data.frame(A=rep(c("a1","a2"),each=3),
                  B=rep(c("b1","b2","b3"),2),
                  C.c1=1:6, C.c2=7:12, C.c3=13:18, C.c4=19:24)
arf <- as.CPF(arf)
stopifnot(is.CPF(arf))

arr <- array(1:24,c(2,3,4),
            dimnames=list(A=c("a1","a2"),B=c("b1","b2","b3"),
                          C=c("c1","c2","c3","c4")))
arrf <- as.CPF(arr)
stopifnot(
  is.CPF(arrf),
  all(levels(arrf$A)==c("a1","a2")),
  all(levels(arrf$B)==c("b1","b2","b3")),
  nrow(arrf)==6, ncol(arrf)==6
)

##Warning, this is not the same as arf, rows are permuted.
as.CPF(as.CPA(arf))

## Not run:
  as.CPF(NodeProbs(node))

## End(Not run)
```

CreateNetwork *Creates (destroys) a new Netica network.*

Description

CreateNetwork() makes a new empty network in Netica, returning new NeticaBN objects. DeleteNetwork() frees the memory associated with the named network inside of Netica.

Usage

```
CreateNetwork(names, session=getDefaultSession())
DeleteNetwork(nets)
```

Arguments

names	A character vector giving the name or names of the network to be created.
session	An object of type NeticaSession which defines the reference to the Netica workspace.
nets	A list of NeticaBN objects to be destroyed.

Details

The CreateNetwork method creates a new network for each of the names. Names must follow the IDname rules. It returns a NeticaBN object, or a list of such objects if the argument names has length greater than 1.

The DeleteNetwork method frees the Netica memory associated with each net in its argument. Note that the network will not be available for use after it is deleted. It returns the NeticaBN objects, but modified so that they are no longer active.

The function is.active(), checks to see if the network associated with a NeticaBN object still corresponds to a network loaded into Netica's memory.

These functions wrap the Netica API functions NewNet_bn() and DeleteNet_bn().

Value

A single NeticaBN object if the length of the argument is 1, and a list of such objects if the argument has length greater than 1. For DeleteNets() if a specified network does not exist, the corresponding element in the return list will be NULL.

Implementation Note

In RNetica version 0.5 and later, the NeticaBN is used to store the refernce to the network. The enclosing NeticaSession object contains a table of network names to NeticaBN objects giving the pointer. It will signal an error if a network with the given name already exists and is active (not deleted).

In RNetica version 0.4 and prior, the NeticaBN object used the name of the networks to store the pointer into the network.

Note

The function DeleteNetwork() implicitly deletes any nodes associated with the network. Therefore, any nodes associated with this network will become inactive (see is.active()).

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: NewNet_bn(), DeleteNet_bn()

See Also

NeticaBN CopyNetworks(), is.active()

Examples

```
sess <- NeticaSession()
startSession(sess)

net1 <- CreateNetwork("EmptyNet", session=sess)
stopifnot(is(net1,"NeticaBN"))
stopifnot(net1$Name=="EmptyNet")
stopifnot(is.active(net1))

netd <- DeleteNetwork(net1)
stopifnot(!is.active(netd))
stopifnot(!is.active(net1))
stopifnot(netd$Name=="EmptyNet")

stopSession(sess)
```

DeleteNodeTable *Deletes the conditional probability table of a Netica node.*

Description

This function completely removes the conditional probability table (CPT) associated with a node.

Usage

```
DeleteNodeTable(node)
```

Arguments

node	An active NeticaNode whose conditional probability table is to be tested.

Value

Returns the modified node invisibly.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: DeleteNodeTables_bn()

See Also

NeticaNode, NodeParents(), NodeInputNames(), HasNodeTable()

Examples

```
sess <- NeticaSession()
startSession(sess)

a1 <- CreateNetwork("AB1", session=sess)
A <- NewDiscreteNode(a1,"A",c("A1","A2"))

NodeProbs(A) <- c(0,1)
stopifnot(
  all(HasNodeTable(A))==TRUE
)

DeleteNodeTable(A)
stopifnot(
  all(HasNodeTable(A))==FALSE
)

DeleteNetwork(a1)
stopSession(sess)
```

---

dgetFromString *Serializes an R object to a string*

Description

The function dputToString converts an R object to a string which can then be turned back into an R object using dgetFromString.

Usage

```
dgetFromString(str)
dputToString(obj)
```

Arguments

str	A string containing a serialized object
obj	An object to be serialized

Details

These functions call the base R functions dget and dput using a string buffer as the connection. Thus, they serialize the R object and return a string value which can be stored in a NeticaNode (see NodeUserObj) or or NeticaBN (see NetworkUserObj).

Note that the object must be self-contained.

Value

The function dputToString returns a character scalar containing the serialized object. Note: Sometimes R "helpfully" adds line breaks, returning a vector of strings. This can be fixed by using paste(dputToString(obj),collapse=" ").

The function dgetFromString returns an arbitrary R object depending on what was stored in str.

Author(s)

Russell Almond

See Also

NodeUserObj), NetworkUserObj

Examples

```
x <- sample(1L:10L)

x1 <- dgetFromString(dputToString(x))

stopifnot(all(x==x1))
```

EliminationOrder *Retrieves or sets the elimination order used in compiling a Netica network.*

Description

The compilation process involves eliminating the nodes in the network one-by-one, different orders will produce junction trees of different sizes. The function EliminationOrder(net) returns the current elimination order associated with a network. The expression EliminationOrder(net) <- value sets the elimination order.

Usage

```
EliminationOrder(net)
EliminationOrder(net) <- value
```

Arguments

net	An active NeticaBN
value	Either NULL (to clear the elimination order) or a list of every node in net with no duplicates.

Details

Large cycles create problems for propagating probabilities in Bayesian networks. A solution to this problem is to fill-in chords (short cuts) in the cycles and then transform the network to a tree shape with the nodes of the tree representing cliques of the graph. This is commonly called a junction tree (although a junction tree additionally has nodes separating the cliques, called *sepsets* in Netica).

Finding the optimal pattern of fill-ins is an NP hard problem. A common way of approaching it is to eliminate the nodes from the network one-by-one and connect the neighbours of the eliminated node (if they were not already connected). In this case, the sequence of eliminated nodes will determine which edges are filled in, and hence the size of the final junction tree. Finding an optimal eliminator order is also NP hard, but simple heuristics (like the greedy algorithm) tend to do reasonably well in practice. (See Almond, 1995, for a complete description of the algorithm and heuristics solutions).

When Netica compiles a network (`CompileNetwork(net)`), it picks an elimination order, unless one has already been set. Unless the network has a particular difficult structure, then the Netica defaults should work pretty well. The function `JunctionTreeReport(net)` gives a report about the existing tree.

If the analyst has some clue about the structure of the network and wants to manually select the elimination order, this can be set through the form `EliminationOrder(net)<-nodelist`. Here `nodelist` should be a complete list of all of the nodes in `net` with no duplication. Alternatively, it can be set to `NULL`.

Setting the elimination order does not affect an already compiled network, it is only is applied when the network is next compiled.

Value

A list of all of the nodes in the network in elimination order if the elimination order is currently set, otherwise `NULL`.

The setter form returns `net` invisibly.

Note

The Netica documentation does not specify the heuristics for selecting the elimination order if no order is specified. I suspect it is some variation on the greedy algorithm, which works well in many cases.

Author(s)

Russell Almond

References

Almond, R.G. (1995) *Graphical Belief Modeling*. Chapman and Hall.

`http://norsys.com/onLineAPIManual/index.html`: GetNetElimOrder_bn(), SetNetElimOrder_bn(),

See Also

`NeticaBN`, `NetworkAllNodes()`, `CompileNetwork()`, `JunctionTreeReport()`

Examples

```
sess <- NeticaSession()
startSession(sess)

EMSMMotif <- ReadNetworks(file.path(library(help="RNetica")$path,
                          "sampleNets","EMSMMotif.dne"), session=sess)

## Should be null before we do anything.
stopifnot(
 is.null(EliminationOrder(EMSMMotif))
)

CompileNetwork(EMSMMotif)
## Now should have an elimination order.
stopifnot(
 length(EliminationOrder(EMSMMotif)) ==
 length(NetworkAllNodes(EMSMMotif)),
 NetworkCompiledSize(EMSMMotif) == 84
)
JunctionTreeReport(EMSMMotif)

## EMSMMotif is partitioned into observable and proficiency variables.
## Tell Netica to eliminate observable variables first.
EliminationOrder(EMSMMotif) <- c(NetworkNodesInSet(EMSMMotif,"Observable"),
                                 NetworkNodesInSet(EMSMMotif,"Proficiency"))
UncompileNetwork(EMSMMotif)
CompileNetwork(EMSMMotif)
stopifnot(
 length(EliminationOrder(EMSMMotif)) ==
 length(NetworkAllNodes(EMSMMotif)),
 NetworkCompiledSize(EMSMMotif) == 84
)
JunctionTreeReport(EMSMMotif)

## Clear elimination order.
EliminationOrder(EMSMMotif) <- NULL
stopifnot(
 is.null(EliminationOrder(EMSMMotif))
)

DeleteNetwork(EMSMMotif)
stopSession(sess)
```

`EnterFindings` *Enters findings for multiple nodes in a Netica network.*

Description

This function takes two arguments, a network and a list of nodes and the corresponding findings. It sets all of the findings at once.

Usage

```
EnterFindings(net, findings)
```

Arguments

net	An active and compiled NeticaBN.
findings	An integer or character vector giving the findings. The names(*findings*) should be names of nodes in net. The values of findings should be corresponding states either expressed as a character string or as an integer index into the list of states for that node. (See NodeFinding(*node*).

Details

This function enters findings for multiple nodes at the same time. It offers two improvements over repeated calls to NodeFinding(). First, it finds the nodes by name in the network, making it easier to work with data in the form of key–value pairs that might come from other systems. Second, it wraps the calls to NodeFinding() in a call to WithoutAutoUpdate() which should only propagate the new findings after all values have been entered.

Value

The value of net is returned invisibly.

Author(s)

Russell Almond

See Also

NeticaBN, NodeBeliefs(), EnterNegativeFinding(), NodeFinding(), RetractNodeFinding(), NodeLikelihood(), EnterGaussianFinding(), EnterIntervalFinding(), JointProbability(), NodeValue(), MostProbableConfig(), FindingsProbability()

Examples

```
sess <- NeticaSession()
startSession(sess)

Motif <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","EMSMMotif.dne"), session=sess)

CompileNetwork(Motif)
obs <- c(Obs1a1="Right",Obs1a2="Wrong",
         Obs1b1="Right",Obs1b2="Wrong",
         Obs2a="Half", Obs2b="Half")

EnterFindings(Motif,obs)
JointProbability(NetworkNodesInSet(Motif,"Proficiency"))

DeleteNetwork(Motif)
stopSession(sess)
```

EnterGaussianFinding *Enter a numeric finding with uncertainty*

Description

This function a likelihood for a node that follows a Gaussian distribution with a given mean and standard deviation. This is entered as virtual evidence.

Usage

```
EnterGaussianFinding(node, mean, sem, retractFirst = TRUE)
```

Arguments

node	An active NeticaNode object that references the node. Node should be continuous, or have numeric value ranges assigned to it using NodeLevels(*node*).
mean	A numeric scalar giving the observed value (mean of the normal).
sem	A nonnegative numeric scalar giving the standard error of measurement for the observed finding (standard deviation of the normal).
retractFirst	A logical value. If true, any previous findings will be retracted first.

Details

The node must a continuous node that has been discretized using NodeLevels(node). The probabilities for each state are calculated based on a Gaussian distribution with the given mean and sem (SD).

Value

Return the node argument invisibly.

Warning

This function is not behaving at all like what I expected. In particular, I expect that it would behave like a normal likelihood, but instead it seems to be behaving as if I typed the expression NodeValue(node)<-mean. I've queried Norsys about this. Use with caution until I get a clarification.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: EnterGaussianFinding_bn(),

See Also

EnterNegativeFinding(), EnterFindings(), RetractNodeFinding(), NodeLikelihood(), NodeFinding(), EnterIntervalFinding(), NodeValue()

Examples

```
sess <- NeticaSession()
startSession(sess)

cirt5 <- CreateNetwork("ContinuousIRT5", session=sess)

theta <- NewContinuousNode(cirt5,"Theta")
NodeLevels(theta) <- c(-5,-2.5,-1.5,-0.5,0.5,1.5,2.5,5)
theta[] <- rep(1/NodeNumStates(theta),NodeNumStates(theta))

CompileNetwork(cirt5) ## Ready to enter findings

EnterGaussianFinding(theta,0,1)
NodeBeliefs(theta)

## I expect this to look like:
diff(pnorm(c(-5,-2.5,-1.5,-0.5,0.5,1.5,2.5,5)))
## But it doesn't!

DeleteNetwork(cirt5)
stopSession(sess)
```

EnterIntervalFinding *Enter finding of value within an interval*

Description

Sets the finding associate with node to an interval.

Usage

```
EnterIntervalFinding(node, low, high, retractFirst = TRUE)
```

Arguments

node	An active NeticaNode object that references the node.
low	Lower bound of interval.
high	Upper bound of interval.
retractFirst	A logical value. If true, any previous findings will be retracted first.

Details

The node must a continuous node that has been discretized using NodeLevels(*node*). The probabilities for each state are calculated based on a uniform distribution with the given low and high endpoints.

Value

Return the node argument invisibly.

Warning

This function is not behaving at all like what I expected. In particular, I expect that it would behave like a normal likelihood, but instead it seems to be behaving as if I typed the expression NodeValue(node)<-low. I've queried Norsys about this. Use with caution until I get a clarification.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: EnterIntervalFinding_bn()

See Also

EnterNegativeFinding(), EnterFindings(), RetractNodeFinding(), NodeLikelihood(), NodeFinding(), EnterGaussianFinding(), NodeValue()

Examples

```
sess <- NeticaSession()
startSession(sess)

cirt5 <- CreateNetwork("ContinuousIRT5", session=sess)

theta <- NewContinuousNode(cirt5,"Theta")
NodeLevels(theta) <- c(-5,-2.5,-1.5,-0.5,0.5,1.5,2.5,5)
theta[] <- rep(1/NodeNumStates(theta),NodeNumStates(theta))

CompileNetwork(cirt5) ## Ready to enter findings

EnterIntervalFinding(theta,-1,1)
NodeBeliefs(theta)

## I expect the middle three values to be non-negative, but that is not
## what I get!

DeleteNetwork(cirt5)
stopSession(sess)
```

EnterNegativeFinding *Sets findings for a Netica node to a list of ruled out values.*

Description

This is conceptually equivalent to setting NodeFinding(*node*)<-not(*eliminatedVals*) (although this will not work as NodeFinding does not accept set values). It essentially eliminates any of the eliminatedVals as possible values (assigns them zero probability).

Usage

```
EnterNegativeFinding(node, eliminatedVals)
```

Arguments

node	An active NeticaNode whose value was observed or hypothesized.
eliminatedVals	A character or integer vector indicating the values to be ruled out. Character values should be one of the values in NodeStates(node). Integer values should be between 1 and NodeNumStates(*node*) inclusive.

Details

This function essentially asserts that $Pr(node \in eliminatedVals) = 0$. Thus, it rules out the values in the eliminatedVals set. Note that the length of this set should be less than the number of states, or all possibilities will have been eliminated.

Note calling EngerNegativeFining(node, ...) clears any previous findings (including virtual findings set through NodeLikelihood() or simple finding set through NodeFinding(*node*)<-*value*). The function RetractNodeFinding(node) will clear the current finding without setting it to a new value.

Value

This function returns node invisibly.

Note

If SetNetworkAutoUpdate() has been set to TRUE, then this function could take some time as each finding is individually propagated. Consider wrapping multiple calls setting NodeFinding() in WithoutAutoUpdate(net, ...).

Unlike the Netica function EnterFindingNot_bn() the function EnterNegativeFinding() internally calls RetractFindings. So there is no need to do this manually. Also, the internal Netica function multiplies multiple calls to EnterFindingNod_bn() add to the list of negative findings, while in the R version takes the entire list.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: EnterFindingNot_bn()

See Also

NeticaBN, NodeBeliefs(), NodeFinding(), RetractNodeFinding(), NodeLikelihood()

Examples

```
sess <- NeticaSession()
startSession(sess)
irt5 <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","IRT5.dne"), session=sess)

irt5.theta <- NetworkFindNode(irt5,"Theta")
irt5.x <- NetworkFindNode(irt5,paste("Item",1:5,sep="_"))

CompileNetwork(irt5) ## Ready to enter findings

## Calculated new expected beliefs
renormed <- NodeProbs(irt5.theta)
renormed[c("neg1","neg2")] <- 0
renormed <- renormed/sum(renormed)

## Negative finding
EnterNegativeFinding(irt5.theta,c("neg1","neg2")) ## Rule out negatives.
stopifnot(
  NodeFinding(irt5.theta) == "@NEGATIVE FINDINGS",
  sum(abs(NodeLikelihood(irt5.theta) - c(1,1,1,0,0))) < 1e-6,
  sum(abs(NodeBeliefs(irt5.theta) - renormed)) < 1.e-6
)


DeleteNetwork(irt5)
stopSession(sess)
```

`Extract.NeticaNode` *Extracts portions of the conditional probability table of a Netica node.*

Description

Provides an efficient mechanism for extracting or setting portions of large conditional probability tables. In particular, allows setting many rows a CPT to the same value.

Usage

```
## S4 method for signature 'NeticaNode'
x[i, j,..., drop=FALSE]
## S4 method for signature 'NeticaNode'
x[[i, j, ...]]
## S4 replacement method for signature 'NeticaNode'
x[i, j, ...] <- value
EVERY_STATE
```

Arguments

x	An active, discrete NeticaNode whose conditional probability table is to be accessed.
i,j,...	Indices specifying rows of the table to extract or replace. If a single index, i, is given, it should be a data frame selecting the parent states, or an integer pointing at a configuration. If multiple indexes are given, the number of indexes should correspond to the number of parent states of the variable. The values should either be character strings (corresponding to parent variable states), or numeric (indexes to parent states). In character strings, the special value "*" is allowed to select all values of that variable. In numeric indexes, the special value EVERY_STATE indicates that all states are selected. Leaving the index position blank is the same as specifying "*" or EVERY_STATE.
drop	If true and a single row is selected, that row will be returned as a numeric vector instead of a conditional probability frame (CPF).
value	Either a numeric vector with length NodeNumStates(x) giving the conditional probabilities for the specified rows in the table or a character scalar (discrete node) or numeric scalar (continuous node) giving the value that should be given probability 1.

Details

The function NodeProbs(*node*) allows one to access the entire conditional probability at once as a conditional probability array (CPA). Although the built-in R array replacement mechanisms allow one to make various kinds of edits, it is relatively inefficient. In particular, to set a single row of an array, the entire table is read into R and then written back to Netica.

This function allows the syntax node[...] to be used to access only a portion of the table. There are many different ways ... can be interpreted, which are described below.

In this access model the value EVERY_STATE or the character value "*" has a special meaning of match every level of that state variable. Netica supports this as a shortcut method for specifying conditional probability tables with many similar values. However, when reading the conditional probability tables from Netica they are expanded and no attempt is made to collapse over identical rows.

A second difference is that node[...] returns the conditional probability table in data frame (CPF) format. This is particularly convenient because that format does not need to cover every parent configuration, thus it is ideal for holding subset of the complete table.

A third difference is that a number of special values are allowed for the probability table. First, if the node is deterministic, the value of a parent configuration can be set to the state name instead of a probability vector. This creates a deterministic conditional probability table full of 1's and 0's. For continuous nodes, the nodes value for a parent configuration (assuming all discrete or discretized parents) can be set directly. Finally, if the last column of the conditional probabilities is not supplied, it will be computed. This is particularly handy for binary nodes.

Normally, the expression `node[...]` produces a data frame either in `CPF` format, or with the probabilities replaced by a single column of values. If `drop==TRUE` or equivalently if `node[[...]]` was the expression, only the matrix of probabilities or the vector of values will be returned. The expression `node[[...]] <- value` is not supported.

The sections below describe the various indexing options.

Value

For the form `node[...]` the return value is a data frame in the `CPF` format giving the conditional probability table. If the node is deterministic (`IsNodeDeterministic(`*node*`)==TRUE`), then the probabilities will be replaced with a single column giving the value of the node. If the node is discrete, then the value will be a factor. If the node is continuous, then the value will be a real vector.

If `drop==TRUE` or an expression of the form `node[[...]]` was called, then the return value will be a matrix of probabilities (the last several columns of the data frame). If the node is deterministic, then the result will instead be either a factor (discrete node) or real vector (continuous node) giving the value of the node for each parent configuration.

The form `node[...]<-`*value* returns `node` invisibly.

Selecting Rows Using Data Frames

This selection uses the syntax `node[df]` or `node[df]<-value`, where `df` is a data frame or a matrix. It is assumed that the columns represent the variables, and the rows represent the selected configurations of the parent variables.

In this configuration, the number of rows of `df` and `value` should match (or the length of `value` should equal the number of rows if one of the special values is used). When the value is being queried rather than set, the number of rows in the result may be greater than the number of rows in `df` because of `EVERY_STATE` expansion.

There are three different ways that `df` could be represented:

1. It can be a data frame filled with factor variables whose levels correspond to the states of the corresponding parent node.
2. It can be a matrix or data frame of type character whose values correspond to the state names of the corresponding parent variables, or possibly the special value `"*"` meaning that all values of that parent should be matched.
3. It can be a matrix of data frame of integers whose values correspond to the state indexes of the parent variables. In this case the special value `EVERY_STATE` can be supplied indicating that all values should be matched. Otherwise, it should be a number between 1 and the number of states of that variable, inclusive.

The number of columns in `df` should be the same as the number of parent variables for `node`. If `df` has column names, then all columns should be named. In this case the parent variables will be

match by the NodeInputNames(node) if they exist, or the names of the parent variables if they do not (see ParentStates(node) for more details). Otherwise, positional selection is used.

Selecting Rows Using Array-type Selection

The second way that rows from the conditional probability table can be selected is using an analogue of the selection mechanisms supported by R for selecting cells from an array. Essentially, the rows of the conditional probability table are treated as if they are the elements of an array whose dimnames correspond to ParentStates{node}. In particular the number of dimensions corresponds to the number of parent variables, and the extent of each dimension corresponds to the number of states of the corresponding parent variable.

In this selection mode, the length of ... should correspond to the number of parent variables (that is, there should be one fewer comma, than parent variables). Each element can be one of three things:

1. A character or factor vector selecting the appropriate states of the parent variable.
2. An integer vector selecting the appropriate states of the parent variable by position.
3. One of the special values EVERY_STATE, "*" or blank indicating that all values of the appropriate variable should be selected.

The order of the entries should be the same as the order of the parent variables in NodeParents{*node*}. The selection looks very similar to selection using a data frame, where the data frame consists of applying expand.grid(...).

Once again EVERY_STATE or "*" entries are treated specially inside of Netica, which allows every matching row of the table to be simultaneously set to the same probabilities.

Note that negative selections and logical selections are not currently supported.

Selecting Rows Using Named Parents

As with R array index selection, the dimensions of the selection in the ... argument can be specified using named arguments. If one of the elements of ... is named, they all should be named. The names should correspond to ParentNames(node), that is the NodeInputNames(node) are used if available, and the names of the parent nodes are used as a fallback.

As before the value for a parent variable can be set to a value or a vector of possible values as either an integer, factor or character value. The special values EVERY_STATE and "*" are interpreted as before. If the value of a parent variable is unspecified, this is equivalent to using the value EVERY_STATE.

Selecting Rows Using a Single Integer

If ... is a single integer, it is treated as an index into the possible configurations. These are defined by expand.grid(ParentStates(node)). Each index refers to a row in that table. This is particularly meant for running through loops on all values, although working with value as a data frame or using NodeProbs may be faster in those cases.

There is some ambiguity when there is a single parent variable about whether the array-type selection or the index was intended, but both are identical, so there should be no conflict.

Special Meaning for NULL selection

If ... is NULL, that is if the calling expression looks like node[] then the intention is that all rows of the conditional probability table are to be selected. This is the only meaningful selection type if there are no parent variables. It also provides a fast and convenient way to set all rows of the conditional probability table to the same value (if value) has a single row, or to retrieve the complete conditional probability table in CPF format.

If value is a data frame with both factor and numeric variables, then it takes on a different meaning. In this case, the factor variables are used as if they were the selection argument (the ...) and the remaining numeric values the probabilities.

Setting Value to a Probability Matrix

In general the replacement value should be a matrix. The number of columns should match the number of states of node (see below for the behavior if the number of columns is one less than the number of states). It should have the same number of rows as the number of rows in the selection after any expansion has been applied for vector valued arguments, but not counting the special values EVERY_STATE or "*" (or blank entries in the list).

Netica has a special shortcut for EVERY_STATE and all matching rows are set to the same probability value. This means that the number of rows in the value must match the selection counting the special values as if they selected a single row. In particular, if node has one or more parent variables and value is a matrix with more than one row, node[] <- value will generate a error, because the selection has only one row (with every value set to EVERY_STATE).

When value is an undimensioned vector, the function will do its best to figure out if it should be treated as a row or a column vector. In the case of unusual behavior, expressing value as a matrix should make the programmer's intention clear.

Setting Deterministic Values

When a node is deterministic, that is all probabilities are 0 or 1, then it is meaningful to talk about the conditional value of a node instead of the conditional probability table. The expression node[...] displays the conditional probability table in a special way when the node is deterministic. In this case it displays the value as a single variable giving the state of the child variable given the configuration of the parents. In the case of discrete nodes, this is a factor variable giving the state. In the case of continuous nodes, this is a numeric vector giving the value.

The same conventions can be used in setting the conditional probability of a node. In the expression node[...] <- value if value is a factor or character vector then the selected configurations are set to deterministic probabilities with the indicated value given probability of 1 and all others with probability 0. It is possible to set some rows of a conditional probability table to be deterministic and others to have unrestricted probabilities, however, the deterministic rows will then print out as unconstrained probabilities with 0 and 1 values.

Continuous nodes (nodes for which is.continuous(*node*) == TRUE) use a variation of this system. Here the value is an arbitrary numeric value. For this to be meaningful, it is assumed that all of the parents of node are either discrete or have been discretized.

Warning: Setting an unconditional discrete node to a constant value, that is executing an expression like node[] <- *value* is almost certainly a mistake. Probably what is intended by that expression is NodeFinding(*node*) <- *value*. In particular, if the former expression is used and the later

someone attempts to set NodeFinding(*node*) <- *value1*, where *value1* != *value*, this will produce a contradiction (probability zero event) and all kinds of error will follow.

Automatic normalization

If the number of columns in value is one less than the number of states in node, then is assumed that the probability values should be calculated for the last state via normalization, that is it is assigned all of the remaining probability not assigned in the first couple of columns. In particular, the value is internally translated via the expression: `value <- cbind(value,1-apply(value,1,sum))`.

This is particularly useful when the node is binary (has exactly 2 states). Then the replacement only needs to specify the probability for the first one. For example `node[] <- .5` would set the probability distribution of node to the uniform distribution if node is binary.

There is some potential for confusion if value is not specified as a matrix. In particular, if the number of states of the child value is one more than the number of configurations of the parents, it is unclear whether this is an attempt to set the node value of a discrete node or an unnormalized probability. It should be possible by specifying value as a matrix or one row or one column to clarify the intent.

Note

I have tried to anticipate most of the ways that somebody might want to index the conditional probability table, not to mention all of the peculiar ways that R overloads the extraction operator. Negative selections are not allowed. I have almost certainly missed some combinations, and some untested combinations might preform rather strangely. Undoubtedly somebody will come to rely on that strangeness and it will never get fixed.

Factor variables do not easily handle the use of `"*"` as a wildcard. To make this work, a construction like `factor(varstates, c(1:3,EVERY_STATE), labels=c("a1","a2","a3","*"))`.

Internally R uses 1-based indexing and Netica uses 0-based indexing. RNetica makes the translation inside of the C layer, so these function should be called with R-style 1-based indexing.

I'm having weird race conditions when trying to set the value of EVERY_STATE (I can't figure out how to call the C function to set its value after the C code is loaded but before the namespace is exported. So for now the exported EVERY_STATE is different from the internal Netica value (which is RNetica:::EVERY_STATE, at least in the current implementation). This should not be a visible change to the user.

This documentation file is longer than *War and Peace*.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeProbs_bn(), SetNodeProbs_bn(), GetNodeFuncState_bn(), SetNodeFuncState_bn(), GetNodeFuncReal_bn(), SetNodeFuncReal_bn(),

See Also

NeticaNode, NodeParents(), NodeInputNames(), NodeStates(), ParentStates(), CPF, CPA

Examples

```
## Setup
sess <- NeticaSession()
startSession(sess)
xnet <- CreateNetwork("X", session=sess)

A <- NewDiscreteNode(xnet,"A",c("A1","A2","A3","A4"))
Aalt <- NewDiscreteNode(xnet,"Aalt",c("A1","A2","A3","A4"))
B <- NewDiscreteNode(xnet,"B",c("B1","B2","B3"))
B2 <- NewDiscreteNode(xnet,"B2",c("B1","B2"))
Balt <- NewDiscreteNode(xnet,"Balt",c("B1","B2","B3"))
C2 <- NewDiscreteNode(xnet,"C2",c("C1","C2"))
C3 <- NewDiscreteNode(xnet,"C3",c("C1","C2","C3"))
C4 <- NewDiscreteNode(xnet,"C4",c("C1","C2","C3","C4"))
Cont <- NewContinuousNode(xnet,"Cont")
CC <- NewContinuousNode(xnet,"CC")
CCC <- NewContinuousNode(xnet,"CCC")

### Tests for various setting modes.

## Null before we set any probabilities anything
stopifnot(
  all(is.na(C2[])), length(C2[]) == 2,
  all(is.na(Cont[])), length(Cont[])==1
)

NodeProbs(C2) <- c(1,0)
stopifnot(
  C2[]=="C1"
)
## This is just a demonstration of the syntax, in practice
## the expression NodeFinding(C2) <- "C2" is usually better.
C2[] <- "C2"
stopifnot(
  NodeProbs(C2)==c(0,1)
)
C3[] <- 3
stopifnot(
  C3[] == "C3"
)

## Setting value of continuous node
Cont[] <- 145.4
stopifnot( abs(Cont[] - 145.4) < .0001)

## Setting value with probabilities
C2[] <- c(.3,.7)
stopifnot( sum(abs(NodeProbs(C2)-c(.3,.7))) < .0001)
C3[] <- c(1,2,1)/4
stopifnot( sum(abs(NodeProbs(C3)-c(.25,.5,.25))) < .0001)
```

```
## Automatic normalization
C2[] <- .25
stopifnot( abs(sum(NodeProbs(C2)-c(.25,.75))) < .0001)
C3[] <- c(1,1)/3
stopifnot( abs(sum(NodeProbs(C3)-1/3)) < .0001)


### Now some one parent cases
AddLink(A,B)
AddLink(A,B2)

stopifnot(
  nrow(B[])==NodeNumStates(A),
  ncol(B[])==1+NodeNumStates(B),
  nrow(B[[]])==NodeNumStates(A),
  ncol(B[[]])==NodeNumStates(B),
  all(is.na(B[][,2:(1+NodeNumStates(B))])),
  all(is.na(B[[]]))
)

NodeProbs(B) <- normalize(matrix(1:12,4))
Brow1 <- B[1]
stopifnot(
  nrow(Brow1)==1,ncol(Brow1)==4,
  sum(abs(Brow1[,2:4]-c(1,5,9)/15))<.00001
)
Brow12 <- B[1:2]
stopifnot(
  nrow(Brow12)==2,ncol(Brow12)==4,
  sum(abs(Brow12[2,2:4]-c(2,6,10)/18))<.00001
)

Brow4 <- B["A4"]
stopifnot(
  nrow(Brow4)==1,ncol(Brow4)==4,
  sum(abs(Brow4[,2:4]-c(1,2,3)/6))<.00001
)
Brow34 <- B[c("A3","A4")]
stopifnot(
  nrow(Brow34)==2,ncol(Brow34)==4,
  abs(sum(Brow4[1,2:4]-c(3,7,11)/21))<.00001
)
Ball <- B["*"]
stopifnot(
  nrow(Ball)==4,ncol(Ball)==4
)
Ball <- B[EVERY_STATE]
stopifnot(
  nrow(Ball)==4,ncol(Ball)==4
)

Brow24 <- B[data.frame(A=factor(c("A2","A4"),NodeStates(A)))]
stopifnot(
```

```
  nrow(Brow24)==2,ncol(Brow24)==4,
  sum(abs(Brow24[2,2:4]-c(1,2,3)/6))<.00001
)

## Set all rows to the same value.
B[] <- matrix(c(1,1,1)/3,1)
stopifnot(
  abs(NodeProbs(B)-1/3)<.0001
)
B[EVERY_STATE] <- matrix(c(1,2,1)/4,1)
stopifnot(
  abs(NodeProbs(B)[3,]-c(.25,.5,.25))<.0001
)
B["*"] <- matrix(c(1,2,3)/6,1)
stopifnot(
  abs(NodeProbs(B)[2,]-c(1/6,1/3,.5))<.0001
)


## Setting to exact values
B2[1:2] <- "B1"
B2[3] <- "B2"
B2[4] <- "B2"
B2tab <- B2[]
stopifnot(
  IsNodeDeterministic(B2),
  nrow(B2tab)==4,ncol(B2tab)==2,
  length(B2[[]]) == 4,
  B2[[]] == c("B1","B1","B2","B2"),
  as.integer(B2tab[,2]) == c(1,1,2,2)
)
## Setting one value to non-deterministic changes the way the table is
## displayed.
B2[2] <- c(.5,.5)
B2tab <- B2[]
stopifnot(
  !IsNodeDeterministic(B2),
  nrow(B2tab)==4,ncol(B2tab)==3,
  sum(abs(B2tab[2,2:3]- c(.5,.5))) < .001,
  B2tab[1,2:3] == c(1,0),
  B2[[3]] == c(0,1)
)

## Self-normalizing setting
## Not run:
## This will generate an error because it is trying to set all four
## configurations to the same value but it is given four values.
B2[] <- c(.1,.2,.3,.4)

## End(Not run)

B2[1:4] <- c(.1,.2,.3,.4)
stopifnot(
```

```
  sum(abs(NodeProbs(B2)[,2]-c(.9,.8,.7,.6))) < .001
)
B2[1:2] <- .5  ## Set both values to the same thing
B2[3:4] <- c(.6,.7) ## set to normalizing probs
stopifnot(
  sum(abs(NodeProbs(B2)[,2]-c(.5,.5,.4,.3))) < .001
)
## Beware!  This next form assumes you are setting the rows to the same
## thing.
B2[3:4] <- c(.2,.8) ## Ambiguous instructions
stopifnot(
  sum(abs(NodeProbs(B2)[,2]-c(.5,.5,.8,.8))) < .001
)
## Using a matrix makes intent clear
B2[3:4] <- matrix(c(.2,.8),2) ## set to normalizing probs
stopifnot(
  sum(abs(NodeProbs(B2)[,2]-c(.5,.5,.8,.2))) < .001
)

## Data frame as value
## First do a blank extraction to get general shape.
B2frame <- B2[]
## Now manipulate it however
B2frame[,2:3] <- 1:8
## And set it back
B2[] <- normalize(B2frame)
stopifnot(
  sum(abs(NodeProbs(B2)[,1]-c(1/6,2/8,3/10,4/12))) <.001
)

B2frame1 <-B2frame[B2frame$A=="A3",]
B2frame1[,2:3] <- c(4,6)/10
B2[] <- B2frame1  ## Only row 3 affected
stopifnot(
  sum(abs(NodeProbs(B2)[,1]-c(1/6,2/8,4/10,4/12))) <.001
)

## Continuous node with one discrete parent
AddLink(A,Cont)  ##Notice how old value is replicated
stopifnot(
  nrow(Cont[]) ==4, ncol(Cont[]) == 2,
  length(Cont[[]]) == 4,
  abs(Cont[][,2]-145.4) <.0001,
  abs(Cont[[3]]-145.4) <.0001
)
AddLink(A,CC)
stopifnot(
  nrow(CC[]) ==4, ncol(CC[]) == 2,
  is.na(CC[][,2])
)

Cont[] <- 7
stopifnot(
```

```
  abs(Cont[[]]-7) <.0001
)
Cont[2] <- 3.2
stopifnot(
  abs(Cont[[]]-c(7,3.2,7,7)) <.0001
)

Cont[1:2] <- 0
Cont[3:4] <- c(8,1)
stopifnot(
  abs(Cont[[]]-c(0,0,8,1)) <.0001,
  abs(Cont[3:4,drop=TRUE]-c(8,1)) < .0001
)


## Two parent case
AddLink(A,C2)
AddLink(B,C2)

C2[] <- c(.5,.5)
stopifnot(
  nrow(C2[])==12, ncol(C2[])==4,
  sum(abs(C2[[]]-.5)) < .0001
)


AddLink(A,C4)
AddLink(B,C4)
stopifnot(
  nrow(C4[])==12, ncol(C4[])==6,
  all(is.na(C4[[]]))
)

NodeProbs(C4) <- normalize(array(1:48,c(4,3,4)))


## Data Frame/matrix Selection

dfsel <- data.frame(A=factor(c("A2","A3"),levels=NodeStates(A)),
                    B=factor(c("B1","B3"),levels=NodeStates(B)))

C21.33 <- C4[dfsel]
stopifnot(
   nrow(C21.33)==2, ncol(C21.33)==6,
   C21.33[1,1] == "A2",
   C21.33[2,2] == "B3",
   abs(C21.33[1,3]-2/80) < .0001,
   abs(C21.33[2,4]-23/116) < .0001
)

dfselbak <- data.frame(B=factor(c("B3","B2"),levels=NodeStates(B)),
                       A=factor(c("A1","A4"),levels=NodeStates(A)))
C13.42 <- C4[dfselbak]
```

```
stopifnot(
   nrow(C13.42)==2, ncol(C13.42)==6,
   C13.42[1,1] == "A1",
   C13.42[2,2] == "B2",
   abs(C13.42[1,3]-9/108) < .0001,
   abs(C13.42[2,4]-20/104) < .0001
)

C2[dfsel] <- matrix(c(.7,.6,.3,.4),2)
C2[dfselbak] <- c(.9,.1)
stopifnot(
  sum(abs(C2[[]][,1] - c(.5,.7,.5,.5, .5,.5,.5,.9, .9,.5,.6,.5))) < .0001
)
## Test for error with using variables in selection inside of a
## function.
testSel <- function(node,sel1,sel2, val) {
 localselvar <- data.frame(sel1,sel2)
 names(localselvar) <- ParentNames(node)
 node[localselvar]
 node[localselvar]<-val
 invisible(node)
}

testSel(C2,factor(c("A2","A3"),levels=NodeStates(A)),
           factor(c("B1","B3"),levels=NodeStates(B)),
           matrix(c(.7,.6,.3,.4),2))


## Array-like selection
stopifnot(
  sum(abs(C4[[2,3]]-c(10,22,34,46)/112))<.0001,
  sum(abs(C4[[B=2,A=4]]-c(8,20,32,44)/104))<.0001
)

C1.23 <- C4[1,2:3]
stopifnot(
  nrow(C1.23)==2, ncol(C1.23)==6,
  sum(abs(C1.23[,3] - c(5/92 ,9/108))) <.0001
)
C2[] <- .5
C2[1,2:3] <- .99
stopifnot(
  sum(abs(C2[[]][,1] - c(.5,.5,.5,.5, .99,.5,.5,.5, .99,.5,.5,.5))) < .0001
)

C1.23 <- C4["A1",c("B2","B3")]
stopifnot(
  nrow(C1.23)==2, ncol(C1.23)==6,
  sum(abs(C1.23[,3] - c(5/92 ,9/108))) <.0001
)
C2[] <- .5
C2["A1",c("B2","B3")] <- .99
stopifnot(
```

```
  sum(abs(C2[[]][,1] - c(.5,.5,.5,.5, .99,.5,.5,.5, .99,.5,.5,.5))) < .0001
)

C34.12 <- C4[3:4,1:2]
stopifnot(
  nrow(C34.12)==4, ncol(C34.12)==6,
  sum(abs(C34.12[,3] - c(3/84,4/88, 7/100, 8/104))) <.0001
)
C2[] <- .5
C2[3:4,1:2] <- .99
stopifnot(
  sum(abs(C2[[]][,1] - c(.5,.5,.99,.99, .5,.5,.99,.99, .5,.5,.5,.5))) < .0001
)

## Wildcards

C1. <- C4[1,EVERY_STATE]
stopifnot(
  nrow(C1.) == 3, ncol(C1.)==6,
  sum(abs(C1.[,3] -c(1/76, 5/92, 9/108))) < .0001
)
C2[] <-.5
C2[1,EVERY_STATE] <- "C1"
stopifnot(
  sum(abs(C2[[]][,1] - c(1,.5,.5,.5, 1,.5,.5,.5, 1,.5,.5,.5))) < .0001
)

C.2 <- C4[EVERY_STATE,2]
stopifnot(
  nrow(C.2) == 4, ncol(C.2)==6,
  sum(abs(C.2[,3] -c(5/92, 6/96, 7/100, 8/104))) < .0001
)
C2[] <-.5
C2[EVERY_STATE,2] <- "C2"
stopifnot(
  sum(abs(C2[[]][,1] - c(.5,.5,.5,.5, 0,0,0,0, .5,.5,.5,.5))) < .0001
)

C1. <- C4["A1","*"]
stopifnot(
  nrow(C1.) == 3, ncol(C1.)==6,
  sum(abs(C1.[,3] -c(1/76, 5/92, 9/108))) < .0001
)
C2[] <-.5
C2["A1","*"] <- "C1"
stopifnot(
  sum(abs(C2[[]][,1] - c(1,.5,.5,.5, 1,.5,.5,.5, 1,.5,.5,.5))) < .0001
)

C.2 <- C4["*","B2"]
stopifnot(
  nrow(C.2) == 4, ncol(C.2)==6,
  sum(abs(C.2[,3] -c(5/92, 6/96, 7/100, 8/104))) < .0001
```

```
)
C2[] <-.5
C2["*","B2"] <- "C2"
stopifnot(
  sum(abs(C2[[]][,1] - c(.5,.5,.5,.5, 0,0,0,0, .5,.5,.5,.5))) < .0001
)

## Missing parent values

C1. <- C4[1,]
stopifnot(
  nrow(C1.) == 3, ncol(C1.)==6,
  sum(abs(C1.[,3] -c(1/76, 5/92, 9/108))) < .0001
)
C2[] <-.5
C2[1,] <- "C1"
stopifnot(
  sum(abs(C2[[]][,1] - c(1,.5,.5,.5, 1,.5,.5,.5, 1,.5,.5,.5))) < .0001
)

C.2 <- C4[,2]
stopifnot(
  nrow(C.2) == 4, ncol(C.2)==6,
  sum(abs(C.2[,3] -c(5/92, 6/96, 7/100, 8/104))) < .0001
)
C2[] <-.5
C2[,2] <- "C2"
stopifnot(
  sum(abs(C2[[]][,1] - c(.5,.5,.5,.5, 0,0,0,0, .5,.5,.5,.5))) < .0001
)

C1. <- C4[A=1]
stopifnot(
  nrow(C1.) == 3, ncol(C1.)==6,
  sum(abs(C1.[,3] -c(1/76, 5/92, 9/108))) < .0001
)
C2[] <-.5
C2[A=1] <- "C1"
stopifnot(
  sum(abs(C2[[]][,1] - c(1,.5,.5,.5, 1,.5,.5,.5, 1,.5,.5,.5))) < .0001
)

C.2 <- C4[B="B2"]
stopifnot(
  nrow(C.2) == 4, ncol(C.2)==6,
  sum(abs(C.2[,3] -c(5/92, 6/96, 7/100, 8/104))) < .0001
)
C2[] <-.5
C2[B="B2"] <- "C2"
stopifnot(
  sum(abs(C2[[]][,1] - c(.5,.5,.5,.5, 0,0,0,0, .5,.5,.5,.5))) < .0001
)
```

```
## Data frame as value

dfset <- data.frame(A=factor(c("A2","A3"),levels=NodeStates(A)),
                    B=factor(c("B1","B3"),levels=NodeStates(B)),
                    C.C1=c(1,0), C.C2=c(0,1))
C2[] <- .5
C2[] <- dfset
stopifnot(
  sum(abs(C2[[]][,1] - c(.5,1,.5,.5, .5,.5,.5,.5, .5,.5,0,.5))) < .0001
)

## Continuous Child node
AddLink(B2,Cont)
stopifnot(
  nrow(Cont[])==8, ncol(Cont[])==3,
  sum(abs(Cont[[]]-c(0,0,8,1))) < .0001
)

AddLink(A,CCC)
AddLink(B,CCC)
stopifnot(
  nrow(CCC[])==12, ncol(CCC[])==3,
  all(is.na(CCC[[]]))
)

Cont[] <- 0
Cont[1,1] <- 1.1
Cont[2:3,2] <- c(2.2,3.2)
Cont["A4","*"] <- 4
## Not run:
## Can't set to multiple values when using * selection.
Cont["A4","*"] <- c(4.1,4.2) ## Generates an error

## End(Not run)
stopifnot(
  sum(abs(Cont[[]]-c(1.1,0,0,4,0,2.2,3.2,4))) < .0001,
  abs(Cont[["A1","B1"]]-1.1) <.0001,
  sum(abs(Cont[[B=2,A=2:3]]-c(2.2,3.2))) < .0001,
  sum(abs(Cont[[A=4]] -4)) < .0001
)

## Set by integer count
## 12 rows in A*B combinations
for (i in 1:12) {
  CCC[i] <- i
  C2[i] <- i/100
}
stopifnot(
  sum(abs(CCC[[]]-t(matrix(1:24,3,4)))) <.0001,
  sum(abs(C2[[]][,1]-t(matrix(1:24/100,3,4)))) <.001
)
for (i in 1:12) {
  stopifnot(
```

```
    abs(CCC[[i]] - i) <.0001,
    abs(C2[[i]][1] - i/100) <.0001
  )
}

### Try some things with three parents, just to make sure that works
### too.
C2tab <- C2[[]]
AddLink(C3,C2)
C2.1tab <- C2[[,,"C1"]]
stopifnot(all.equal(C2tab,C2.1tab),
          all.equal(C2tab,C2[[,,"C1"]]),
          all.equal(C2tab,C2[[C="C3"]]))

stopifnot(all(abs(C2[["A1","B1","C1"]]-NodeProbs(C2)[1,1,1,])<.0001),
          all.equal(C2["A1",,],C2[A="A1"]),
          all.equal(C2[,"B2",],C2[B="B2"]),
          all.equal(C2["A1","B2",],C2[B="B2",A="A1"]))


DeleteNetwork(xnet)
stopSession(sess)
```

FadeCPT *Fades a Netica Conditional Probability Table*

Description

This function fades a Netica conditional probability table associated with a node (that is, it makes it closer to uniform). This is used when learning conditional probabilities over time, so that newer observations will have more weight than older ones.

Usage

```
FadeCPT(node, degree = 0.2)
```

Arguments

node	A NeticaNode object.
degree	A scalar value between 0 and 1 providing the amount of fading to be done. A degree of 1 produces a uniform distribution and a degree of 0 leaves the CPT unchanged.

Details

This is essentially an exponential filter, with `1-degree` as the retained weight. Calling it once with degree of $1 - d$ and again with degree $1 - f$ is equivalent to calling it once with degree $1 - df$.

If prob are the current probabilities associated with a row of the CPT, and expr is the current experience, then the new probabilities will be newprob = normalize(prob* exper * (1-degree) + degree), and the new experience will be the normalization constant.

This function is often used together with LearnFindings to down weight old cases when the conditional probabilities are thought to be changing slowly over time.

Value

This function returns the node object.

Note

Frequently the degree is made time dependent. If dt is the time elapsed since the last observation, the degree is frequently an expression like 1-expt(R,dt), where R is a constant less than 1 which controls how quickly the CPT is faded.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: FadeCPTable_bn()

See Also

NodeExperience, NodeProbs, LearnFindings

Examples

```
sess <- NeticaSession()
startSession(sess)

aaa <- CreateNetwork("AAA", session=sess)
A <- NewDiscreteNode(aaa,paste("A",1:5,sep=""),c("true","false"))

for( i in 1:length(A)) {
  NodeProbs(A[[i]]) <- c(.8,.2)
  NodeExperience(A[[i]]) <- 10
}

deg <- .2
expected <- NodeProbs(A[[1]])*10*(1-deg)+deg

FadeCPT(A[[1]], deg)
stopifnot(
  sum(abs(NodeProbs(A[[1]])-expected/sum(expected))) < .0001,
  abs(NodeExperience(A[[1]])-sum(expected)) < .001
)

## Fading by deg then by deg2 is the same as fading by
```

```
## 1-(1-deg)*(1-deg2)
deg2 <- .3
FadeCPT(A[[1]],deg2)
FadeCPT(A[[2]], 1-(1-deg)*(1-deg2))
stopifnot (
  sum(abs(NodeProbs(A[[1]]) - NodeProbs(A[[2]]))) < .0001
)

## Fade by two time units.
lambda <- .8
FadeCPT(A[[3]],1-lambda^2)

## Special cases
FadeCPT(A[[4]],0)
FadeCPT(A[[5]],1)

stopifnot (
  sum(abs(NodeProbs(A[[4]]) -c(.8,.2))) < .0001,
  sum(abs(NodeProbs(A[[5]]) -c(.5,.5))) < .0001
)

DeleteNetwork(aaa)
stopSession(sess)
```

FileCaseStream-class *Class* "FileCaseStream"

Description

This object is subclass of CaseStream so it is a wrapper around a Netica stream which is used to read/write cases. In this subclass, the case stream is associated with a Netica case file ('.cas' extension). The function CaseFileStream is the constructor. The function ReadFindings reads the findings from the stream and the function WriteFindings writes them out.

Extends

Class "CaseStream", directly.

All reference classes extend and inherit methods from "envRefClass". Note that because this is a reference class unlike traditional S3 and S4 classes it can be destructively modified. Also fields (slots) are accessed using the '$' operator.

Fields

Note these should be regarded as read-only from user code.

Name: Object of class character used in printed representation. Default is basename(*Case_Stream_Path*).

Session: Object of class NeticaSession a link to the session in which this case stream was created.

`Netica_Case_Stream`: Object of class `externalptr` a pointer to the case stream in Netica memory.

`Case_Stream_Position`: Object of class `integer` the number of the last read/writen record. This is `NA` if the end of the file has been reached.

`Case_Stream_Lastid`: Object of class `integer` the ID number of the last read/written record.

`Case_Stream_Lastfreq`: Object of class `numeric` giving the frequence of the last read/written record. This is used as a weight in learning applications.

Methods

`open()`: Opens a connection too the file in Netica.

`show()`: Provides a description of the field

`initialize(Name, Session, Case_Stream_Path, ...)`: internal constructor; user code should use `CaseFileStream`.

The following methods are inherited (from `CaseStream`): close ("CaseStream"), isActive ("CaseStream"), isOpen ("CaseStream"), show ("CaseStream"), clearErrors ("CaseStream"), reportErrors ("CaseStream"), initialize ("CaseStream")

Note

In version 0.5 of RNetica, this class was renamed. It is now called `FileCaseStream` but the constructor is still called `CaseFileStream` (while previously the class and the filename had the same name). This matches the usage of `FileCaseStream` and its constructor `CaseFileStream`. It is also now a reference class instead of an informal S3 class. This is only likely to be a problem for code that was using the hard coded class name.

Stream objects are fragile, and will not survive saving and restoring an R session. However, the object retains information about itself, so that calling `OpenCaseStream` on the saved object, should reopen the stream. Note that any position information will be lost.

Author(s)

Russell Almond

References

`http://norsys.com/onLineAPIManual/index.html`: NewFileStream_ns(), DeleteStream_ns() `http://homepage.stat.uiowa.edu/~luke/R/references/weakfinex.html`

See Also

See `CaseStream` for the superclass and `MemoryCaseStream` for a sibling class. The function `CaseFileStream` is the constructor.

`OpenCaseStream`, `CaseFileDelimiter`, `CaseFileMissingCode`, `WriteFindings`, `ReadFindings`, `WithOpenCaseStream`

Examples

```
sess <- NeticaSession()
startSession(sess)

abc <- CreateNetwork("ABC", session=sess)
A <- NewDiscreteNode(abc,"A",c("A1","A2","A3","A4"))
B <- NewDiscreteNode(abc,"B",c("B1","B2","B3"))
C <- NewDiscreteNode(abc,"C",c("C1","C2"))

AddLink(A,B)
AddLink(A,C)
AddLink(B,C)

## Outputfilename
casefile <- tempfile("testcase",fileext=".cas")

filestream <- CaseFileStream(casefile, session=sess)
stopifnot(is.CaseFileStream(filestream),
          isCaseStreamOpen(filestream))

## Case 1
NodeFinding(A) <- "A1"
NodeFinding(B) <- "B1"
NodeFinding(C) <- "C1"
filestream <- WriteFindings(list(A,B,C),filestream,1001,1.0)
stopifnot(getCaseStreamLastId(filestream)==1001,
          abs(getCaseStreamLastFreq(filestream)-1.0) <.0001)

## Close it
filestream <- CloseCaseStream(filestream)
stopifnot (is.CaseFileStream(filestream),
           !isCaseStreamOpen(filestream))

## Reopen it
filestream <- OpenCaseStream(filestream)
stopifnot (is.CaseFileStream(filestream),
           isCaseStreamOpen(filestream))

##Case 1
RetractNetFindings(abc)
filestream <- ReadFindings(list(A,B,C),filestream,"FIRST")
stopifnot(getCaseStreamLastId(filestream)==1001,
          abs(getCaseStreamLastFreq(filestream)-1.0) <.0001)

##Clean Up
CloseCaseStream(filestream)
DeleteNetwork(abc)
stopSession(sess)
```

FindingsProbability *Finds the probability of the findings entered into a Netica network.*

Description

This function assumes that the network has been compiled and that a number of findings have been entered. The function calculates the prior probability for the entered findings (that is, the normalization constant of the Bayesian network).

Usage

```
FindingsProbability(net)
```

Arguments

net	An active and compiled Bayesian Network (class NeticaBN).

Details

In the usual algorithms for propagating probabilities in a Bayesian network the probabilities are passed unnormalized. When reporting the probabilities, a normalization constant is calculated. This normalization constant is the probability of all of the findings that have been entered through NodeFinding(). (See Almond, 1995, for details on the use of normalization constants as probabilities of findings.)

It is not meaningful to call this function before the network has been compiled. Calling it before findings have been entered will result in a value of 1.0.

Value

A scalar real value representing the probability of the findings, or NA if the network was not found or not compiled.

Note

Netica gives a warning about the interpretation if likelihood findings have been set (through NodeLikelihood(). In this case, the value is perhaps better though of as a normalization constant.

Author(s)

Russell Almond

References

Almond, R. G. (1995) *Graphical Belief Modeling*. Chapman and Hall.

http://norsys.com/onLineAPIManual/index.html: FindingsProbability_bn()

See Also

NeticaNode,NeticaBN,NodeBeliefs(),EnterNegativeFinding(),RetractNodeFinding(),NodeLikelihood()

Examples

```
sess <- NeticaSession()
startSession(sess)

EMSMMotif <- ReadNetworks(file.path(library(help="RNetica")$path,
                          "sampleNets","EMSMMotif.dne"), session=sess)

CompileNetwork(EMSMMotif)
norm1 <- FindingsProbability(EMSMMotif)
stopifnot ( abs(norm1-1) <.0001)

## Find observable nodes
obs <- NetworkNodesInSet(EMSMMotif,"Observable")

NodeFinding(obs$Obs1a1) <- "Right"
NodeFinding(obs$Obs1a2) <- "Wrong"

prob1r2w <- FindingsProbability(EMSMMotif)
stopifnot (prob1r2w < 1, prob1r2w > 0)

## Clear it out and try again
RetractNetFindings(EMSMMotif)
NodeLikelihood(obs$Obs2a) <- c(.75,.75,.75)
prob75 <- FindingsProbability(EMSMMotif)
stopifnot( abs(prob75-.75) < .0001)

DeleteNetwork(EMSMMotif)
stopSession(sess)
```

GenerateRandomCase *Generates random cases for nodes in a Netica network*

Description

This function generates a random instantiation of the nodes in `nodelist` using the current (that is posterior to any findings entered into the net) joint probability distribution of those nodes in the network.

Usage

```
GenerateRandomCase(nodelist, method = "Default", timeout = 100, rng = NULL)
```

Arguments

nodelist	A list of active NeticaNode objects, all of which belong to the same network.
method	A character scalar used to describe the method used select the random numbers. This should have one of the values "Join_Tree_Sampling", "Forward_Sampling" or "Default_Sampling" (see details). Only the first letter is used and case is ignored, so "J", "F" and "D" are legal values.
timeout	This is a number describing how long to carry on computations under the forward sampling method. It is ignored under the join tree sampling method or when the default sampling method turns out to be join tree.
rng	This either be an existing NeticaRNG object or NULL in which case the default random number generator for the net is used.

Details

The function visits each node in nodelist and randomly sets a finding for that node based on the current beliefs about that node. This takes into account any findings previously entered into the graph (including the previously sampled nodes in the list). In particular, to generate multiple cases, the findings need to be retracted (using RetractNodeFinding(*node*) or RetractNetFindings(*net*) between each generation.

Netica supports three methods for doing the sampling:

Join_Tree_Sampling. For each node in turn, the beliefs are calculated and a random state is selected and entered as a finding (with beliefs propagating). The network must be compiled for this method to work.

Forward_Sampling. Random cases are generated directly using equations for continuous nodes if these are available. Random results not compatible with the current findings are rejected. This method is not guaranteed to converge, and may be quite slow if the current set of findings has a low probability. It will only run for a period of time indicated by timeout and returns a negative value if it does not complete successfully.

Default_Sampling. Netica figures out which method is better to use. It uses forward sampling if either rejections aren't a problem (presumably because there are no findings) or if the network is uncompiled. Otherwise it uses join tree sampling.

The rng argument can be used to associate a random number generator with the generation (see NeticaRNG). If the rng argument is NULL, then the default random number generator for the network is used. This is either a random number generator associated with the network using NetworkSetRNG, or else the default Netica random number generator.

Value

Invisibly returns 0 if the case was successfully generated or -1 if the case could not be generated (using the forward sampling method). In the latter case, a warning is issued as well.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GenerateRandomCase_bn()

See Also

NetworkSetRNG(), NeticaRNG(), NodeFinding, RetractNetFindings ReadFindings, CaseStream

Examples

```
sess <- NeticaSession()
startSession(sess)

irt5 <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","IRT5.dne"), session=sess)

irt5.theta <- NetworkFindNode(irt5,"Theta")
irt5.x <- NetworkFindNode(irt5,paste("Item",1:5,sep="_"))

CompileNetwork(irt5)

GenerateRandomCase(irt5.x)
sapply(irt5.x,NodeFinding)

RetractNetFindings(irt5)

GenerateRandomCase(irt5.x)
sapply(irt5.x,NodeFinding)

## This generates a fixed series of random cases and saves them to a
## file.
N <- 10L
rnodes <- c(list(irt5.theta),irt5.x)
casefile <- tempfile("irt5testcase",fileext=".cas")
filestream <- CaseFileStream(casefile, session=sess)
rng <- NewNeticaRNG(123456779, session=sess)
WithOpenCaseStream(filestream,
  WithRNG(rng,
    for (n in 1L:N) {
      GenerateRandomCase(rnodes,rng=rng)
      WriteFindings(rnodes,filestream,n)
      lapply(rnodes,RetractNodeFinding) # Only retract findings for
                                        # generated nodes
    }))

## With constructs force closure even on error exit.
stopifnot(!isNeticaRNGActive(rng),
          !isCaseStreamOpen(filestream))

DeleteNetwork(irt5)
stopSession(sess)
```

GetNamedNetworks *Finds a Netica network (if it exists) for the name.*

Description

This searches through the currently open Netica networks in the NeticaSession and returns a NeticaBN object pointing to the networks with the given names. If no network with the name is found NULL is returned instead, so this provides a way to check whether a network exists.

CheckNamedNetworks checks the internal Netica list of networks, not the networks cached in the NeticaSession object, so it can be used to check for inconsistencies.

Usage

```
GetNamedNetworks(namelist, session=getDefaultSession())
CheckNamedNetworks(namelist, session=getDefaultSession())
```

Arguments

namelist	A character vector giving the name or names of the networks to be found.
session	An object of type NeticaSession which defines the reference to the Netica workspace.

Details

GetNamedNetworks() searches the list of network names looking for a network with the appropriate name. If it is found, a handle to that network is returned as a NeticaBN object. If not, NULL is returned. Note that if a network of the specified name existed, it could return an inactive NeticaBN object corresponding to the deleted network, so it is probably good to check the result with is.active.

There are two ways that RNetica can check for a network of a given name. The first the network cache maintained by the NeticaSession object (session$nets). The function GetNamedNetworks just checks the cache, so it should be relatively fast. The function CheckNamedNetworks iterates through all of the networks in Netica's internal memory, so it should be slower, but should also spot problems with RNetica and Netica getting out of sync.

Value

If namelist is of length 1, then a single NeticaBN object or NULL will be returned.

If namelist is of length greater than 1, then a list of the same length as namelist is returned. Each element is a NeticaBN related to the corresponding name or NULL if the name does not refer to a network.

Note

Each NeticaBN is given a name when it is created. When the network is created, either through a call to CreateNetwork or ReadNetworks, the NeticaSession object updates its cache of the network names in its nets field. The nets field of the session object is an environment which associate the network's name with a NeticaBN object. Internally, functions that return a NeticaBN object (primarily NodeNet), search the network cache in the session object for the network with the corresponding name.

GetNamedNetworks uses the cache (which is hashed) and so should be fairly fast.

CheckNamedNetworks does a linear search through all networks, so it could be pretty slow if there are a large number of networks open. It should raise an error if the cache and the internal Netica specs are out of sync.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNthNet_bn()

See Also

CreateNetwork(), GetNthNetwork()

Examples

```
sess <- NeticaSession()
startSession(sess)

net1 <- CreateNetwork("myNet", session=sess)
## Fetch the network we just created by name.
net2 <- GetNamedNetworks("myNet", session=sess)
stopifnot(is(net2,"NeticaBN"))
stopifnot(NetworkName(net1)==NetworkName(net2))
stopifnot(net1==net2)

net3 <- CheckNamedNetworks("myNet", session=sess)
stopifnot(net1==net3)

## No network named "fish", this should return NULL
fish <- GetNamedNetworks("fish", session=sess)
stopifnot(all(sapply(fish,is.null)))
fish <- CheckNamedNetworks("fish", session=sess)
stopifnot(all(sapply(fish,is.null)))

DeleteNetwork(net1)
net1a <- GetNamedNetworks("myNet", session=sess)
stopifnot(NetworkName(net1a)=="myNet",!is.active(net1a))

net1b <- CheckNamedNetworks("myNet", session=sess)
```

```
stopifnot(is.null(net1b))

stopSession(sess)
```

GetNetworkAutoUpdate *Turns Netica automatic updating on or off for a network.*

Description

Netica networks can either propagate the effects of new findings immediately, or they can delay propagation until the user queries the network. These functions toggle the switch that controls the autoupdate mechanism

Usage

```
GetNetworkAutoUpdate(net)
SetNetworkAutoUpdate(net, newautoupdate)
WithoutAutoUpdate(net,expr)
```

Arguments

net	A NeticaBN object to be queried or changed.
newautoupdate	A logical values, TRUE to turn automatic updating on. A value NA produces an error.
expr	An R expression to be evaluated with automatic updating turned off.

Details

Automatic updating means that queries operate very quickly, however, if a large number of finding are to be entered before the next query, they can slow the network down. These functions provide a mechanism for controlling that.

GetNetworkAutoUpdate() returns the current status of the autoupdate flag. SetNetworkAutoUpdate() sets flag, but returns its current value (to make it easier to restore). The function WithoutAutoUpdate provides a mechanism for turning updating off while performing a series of operations.

Value

GetNetworkAutoUpdate() and SetNetworkAutoUpdate both returns the current autoupdate flag as a logical value.

WithoutAutoUpdate() returns the value of executing expr, unless executing expr results in an error in which case it returns a try-error.

Note

Automatic updating makes a lot of sense when Netica is running under the GUI, but not so much when it is running as an API. It is probably easiest to just set this to false all the time.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: SetNetAutoUpdate_bn(), GetNetAutoUpdate_bn()

See Also

NeticaBN, NodeBeliefs(), NodeFinding()

Examples

```
sess <- NeticaSession()
startSession(sess)

autoNet <- CreateNetwork("AutomaticTest", session=sess)

GetNetworkAutoUpdate(autoNet)

SetNetworkAutoUpdate(autoNet,FALSE)
stopifnot(!GetNetworkAutoUpdate(autoNet))
stopifnot(!SetNetworkAutoUpdate(autoNet,TRUE))
stopifnot(GetNetworkAutoUpdate(autoNet))

result <- TRUE
WithoutAutoUpdate(autoNet, result <<-GetNetworkAutoUpdate(autoNet))
stopifnot(!result)

DeleteNetwork(autoNet)
stopSession(sess)
```

GetNthNetwork *Fetch a Netica network by its position in the Netica list.*

Description

Fetches networks according to an internal sequence list of networks maintained inside of Netica. If the number passed is greater than the number of currently defined networks, this function will return NULL

Usage

```
GetNthNetwork(n, session = getDefaultSession())
```

Arguments

n	A vector of integers greater than 1.
session	An object of class NeticaSession which provides the link to the Netica environment. If not supplied, then the default value is the value of the function getDefaultSession() which is usually the value of DefaultNeticaSession in the global environment.

Details

The primary use for this function is probably to loop through all open networks. As this function will return NULL when there are no more networks, that can be used to terminate the loop.

Note that the sequence numbers can change, particularly after functions that open and close networks.

This is a wrapper for the Netica function GetNthNet_bn().

Starting with RNetica 0.5, the session object is a container which contains the open networks, so this function is no longer really needed.

Value

If n is of length 1, then a single NeticaBN object or NULL will be returned.

If n is of length greater than 1, then a list of the same length as n is returned. Each element is a NeticaBN related or NULL if the number is greater than the number of open networks.

Note

The Netica shared library uses a zero-based reference (i.e., the first net is 0), but this function subtracts 1 from the argument, so it uses a one-based reference system (the first net is 1).

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNthNet_bn()

See Also

NeticaSession, CreateNetwork(), GetNamedNetworks()

Examples

```
sess <- NeticaSession()
startSession(sess)

count <- 1
while (!is.null(net <- GetNthNetwork(count,sess))) {
  cat("Network number ",count," is ",net,".\n")
```

```
  count <- count +1
}
cat("Found ",count-1," networks.\n")

stopSession(sess)
```

`HasNodeTable` *Tests to see if a Netica node has a conditional probability table.*

Description

This function tests to see if a conditional probability table has been assigned to `node`. The function returns two values, the first tests for existence of the table, the second tests for a complete table (no NAs).

Usage

```
HasNodeTable(node)
```

Arguments

`node`	An active `NeticaNode` whose conditional probability table is to be tested.

Details

This function returns two values. The first is true or false according to whether the conditional probability table has been established, that is has `NodeProbs()` been set. The second value tests to see whether the conditional probability table is complete, that is, does it have any `NA`s associated with it.

In many cases, it is the second value that is of interest, so `all(HasNodeTable(`*node*`))` is often a useful idiom.

Value

A logical vector with two elements. The first states whether or not the node has any of its conditional probabilities set. The second tests whether or not the table has been completely specified.

Note

Generating incomplete tables is pretty hard to do in RNetica, a row must be deliberately set to `NA`. However, a network read in from a file might have incomplete tables.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: HasNodeTable_bn()

See Also

NeticaNode, NodeParents(), NodeInputNames(), DeleteNodeTable()

Examples

```
sess <- NeticaSession()
startSession(sess)

ab1 <- CreateNetwork("AB1", session=sess)
A <- NewDiscreteNode(ab1,"A",c("A1","A2","A3"))
B <- NewDiscreteNode(ab1,"B",c("B1","B2"))
AddLink(A,B)

##Nodes start undefined.
stopifnot(
  HasNodeTable(A)==c(FALSE,FALSE)
)

NodeProbs(A) <- c(0,1,0)
stopifnot(
  HasNodeTable(A)==c(TRUE,TRUE)
)

for (node in NetworkAllNodes(ab1)) {
  if (!all(HasNodeTable(node))) {
   cat("Node ", toString(node),
  " still needs a conditional probability table.\n")
  }
}

DeleteNetwork(ab1)
stopSession(sess)
```

IDname *Tests to see if a string is a valid as a Netica Identifier.*

Description

The function `is.IDname()` returns a logical vector indicating whether or not each element of `x` is a valid Netica identifier. The function `is.IDname()` attempts to massage the input value to conform to the IDname rules.

Usage

```
is.IDname(x)
as.IDname(x,prefix="y",maxlen=25)
```

Arguments

x	A character vector of possible identifier names.
prefix	A character scalar that provides an alphabetic prefix for names that start with an illegal character.
maxlen	The maximum number of characters to use in the converted name, which should be less than Netica's maximum of 30 characters.

Details

Netica identifiers (net names, node names, state names, and similar) are limited to 30 characters which must be a valid letter, number or the character '_'. The first character must be a letter. The function `is.IDname()` tests to see if a string conforms to these rules, and thus is a legal name.

The function `as.IDname()` attempts to coerce its argument into the IDname format by applying the following transformations.

1. The argument is coerced into a character vector.
2. If any value begins with a nonalphabetic character, the `prefix` argument is prepended to all values.
3. All non-alphanumeric characters are converted to '_'.
4. Each value is truncated to `maxlen` characters in length.

The truncation works by the following mechanism:

1. The string is truncated to length `maxlen-3`.
2. The UTF 8 values of the remaining characters is summed and the result is taken modulo 100 to provide a 2-digit hash code for the remaining characters.
3. The hash code is appended to the end of the truncated string separated with an _.

This should result in strings which are likely, but not guaranteed to be unique if the difference between two names is only after the last `maxlen-3` characters.

Note that although Netica allows variable names up to 30 characters in length, in some cases (particularly when stub variables are created after separating an edge from its parent) Netica creates new variable names by appending characters onto existing ones. That is why the recommended value for `maxlen` is set to 25.

Value

A logical vector of the same length of `x`.

Note

This is primarily a utility for doing argument checking inside of functions that require a Netica IDname.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html

See Also

CreateNetwork(), NewDiscreteNode(), NodeStates(), NodeName(), NodeInputNames(),

Examples

```
stopifnot(
is.IDname(c("aFish","Wanda1","feed me","fish_food","1more","US$",
                      "a12345678901234567890123456789O")) ==
          c(TRUE,TRUE,FALSE,TRUE,FALSE,FALSE,FALSE),
as.IDname(c("aFish","Wanda1","feed me","fish_food","1more","US$",
                      "a12345678901234567890123456789O")) ==
          c("aFish","Wanda1","feed_me","fish_food","y1more","US_",
                      "a1234567890123456789O1_25")
)
```

---

is.active *Check to see if a Netica network or node object is still valid.*

Description

NeticaSession, NeticaBN, NeticaNode, CaseStream, and NeticaRNG objects all contain embedded pointers into Netica's memory. The function is.active() checks to see that the corresponding Netica object still exists.

Usage

```
is.active(x)
```

Arguments

x	A NeticaBN or NeticaNode object to test, or a list of such objects.

Details

Internally, NeticaSession, NeticaBN and NeticaNode objects all contain pointers to the corresponding Netica objects. The DeleteNetwork() and DeleteNodes() functions deletes the Netica objects (and clears the pointers in the R objects). It is difficult to control when R objects are deleted, especially if they are protected in data structures that are saved in the workspace. The function is.active() is meant to check if the corresponding object is still valid. In most cases, RNetica will give an error (or at least a warning) if an inactive object is supplied as an argument.

For CaseStream objects (and its sub-classes FileCaseStream and MemoryCaseStream) active and open have the same meaning.

For NeticaRNG objects, they become inactive when they are freed.

Note that the function StopNetica() should make all NeticaBN and NeticaNode objects inactive. Thus, these objects cannot be saved from one R session to another, and should be recreated when needed. In particular, any Netica object restored from a saved workspace should be inactive.

Value

The function is.active() returns TRUE if the argument still points to a network or node loaded in Netica's memory, and FALSE if that network or node has been deleted. It returns NA if the argument is not a NeticaSession, NeticaBN, NeticaNode, CaseStream, or NeticaRNG object.

If x is a list, then a logical vector of the same length of x is returned with is.active() recursively applied to each one.

Note

The actual test done is to test the pointer to see if it is null or not. It should be the case that when an R object is disconnected from its Netica counterpart, the pointer is set to null.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html, http://lib.stat.cmu.edu/R/CRAN/doc/manuals/R-exts.html

See Also

StopNetica(), NeticaBN, DeleteNetwork(), NeticaNode, DeleteNodes(), NeticaSession, CaseStream, NeticaRNG

Examples

```
sess <- NeticaSession()
stopifnot(!is.active(sess))
startSession(sess)
stopifnot(is.active(sess))

anet <- CreateNetwork("ActiveNet", session=sess)
stopifnot(is.active(anet))

anodes <- NewContinuousNode(anet,paste("ActiveNode",1:2,sep=""))
stopifnot(all(is.active(anodes)))

inode <- DeleteNodes(anodes[[1]])
stopifnot(!is.active(anodes[[1]]))
stopifnot(!is.active(inode))
```

```
stopifnot(is.active(anodes[[2]]))

DeleteNetwork(anet)
stopifnot(!is.active(anet))
## Node gets deleted along with network
stopifnot(!any(is.active(anodes)))

rng <- NewNeticaRNG(1, session=sess)
stopifnot(is.active(rng))
FreeNeticaRNG(rng)
stopifnot(!is.active(rng))

casefile <- tempfile("testcase",fileext=".cas")
filestream <- CaseFileStream(casefile, session=sess)
stopifnot(is.active(filestream))
CloseCaseStream(filestream)
stopifnot(!is.active(filestream))

stopSession(sess)
stopifnot(!is.active(sess))
```

`is.discrete` *Determines whether a Netica node is discrete or continuous.*

Description

A `NeticaNode` object can take on either a discrete set of values or an arbitrary real value. These functions determine which type of node this is.

Usage

```
is.discrete(node)
is.continuous(node)
```

Arguments

node	A `NeticaNode` object to test.

Details

While in the Netica GUI, one first creates a node and then determines whether it will be discrete or continuous, in the API this is determined at the time of creation (by calling `NewContinuousNode()` or `NewDiscreteNode()`. These functions determine which type of node the given node is.

Note that setting `NodeLevels` can make a continuous node behave like a discrete one and vice versa. For continuous nodes, the levels are cut points for getting a discrete state from the node. For a discrete node, the levels are real values representing the midpoint of the states.

Value

`TRUE` or `FALSE` depending on whether a node is discrete or continuous.

Note

Currently, this function does not actually look at the internal Netica state, but rather looks at the field "discrete" which is set when the node is created.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeType_bn(), SetNodeLevels_bn()

See Also

NewDiscreteNode(), NewContinuousNode(), NeticaNode, NodeLevels(), NodeStates()

Examples

```
sess <- NeticaSession()
startSession(sess)

netx <- CreateNetwork("netx", session=sess)

bnode <- NewDiscreteNode(netx,"bool",c("True","False"))
stopifnot(is.discrete(bnode))
stopifnot(!is.continuous(bnode))

rnode <- NewContinuousNode(netx,"real")
stopifnot(!is.discrete(rnode))
stopifnot(is.continuous(rnode))

DeleteNetwork(netx)
stopSession(sess)
```

---

is.NodeRelated *Computes topological proprieties of a* Netica *network.*

Description

The function is.NodeRelated() tests to see if relation holds between node1 and node2. The function GetRelatedNodes creates a list of all nodes that satisfy the relation with any node in nodelist.

Usage

```
is.NodeRelated(node1, node2, relation = "connected")
GetRelatedNodes(nodelist, relation = "connected")
```

Arguments

node1	An active NeticaNode whose relationship will be tested.
node2	Another active NeticaNode whose relationship will be tested.
relation	A character scalar which should be one of the values: "parents", "children", "ancestors", "descendents" [sic], "connected", "markov_blanket", or "d_connected". Singular forms and modifiers are also allowed, see details.
nodelist	A list of active NeticaNode whose relationship will be tested.

Details

These functions are useful for testing the topology of a network. Each of the functions offers a measure related to the network. The is.NodeRelated() form tests the relationship between node1 and node2. The function GetRelatedNodes() returns a list of any nodes for which the relationship holds with any of the elements of nodelist. The plural and singular forms of the relationships can be used with both functions.

"parent", "parents". True if node1 is a parent of node2, or returns a list of parents of the nodes in nodelist.

"ancestor", "ancestors". True if there is a directed (parent to child) path from node1 to node2, or returns a list of ancestors of the nodes in nodelist.

"child", "children". True if node1 is a child of node2, or returns a list of children of the nodes in nodelist.

"descendent", "descedents" [This is the spelling used by Netica]. True if there is a directed (parent to child) path from node2 to node1, or returns a list of descedants of the nodes in nodelist.

"connected". True if there is a chain (unordered path) from node1 to node2, or returns a list of all nodes connected to any of the nodes in nodelist.

"markov_blanket". The Markov blanket of nodeset is the a set of nodes that renders the nodes in nodeset conditionally independent of the remaining nodes given the ones in the blanket. The simple form returns true if node2 is in the Markov blanket of node1.

"d_connected". The rules for d-connection are somewhat complex (see Pearl, 1988), but basically node1 and node2 are d-connected if they are not independent given the current findings. The function returns true if node1 and node2 are d-connected or a list of all nodes that are d-connected to the nodes in nodelist.

In addition, the relation can be modified in the GetRelatedNodes() form by adding one or more modifiers to the main relation separated by commas. The two that are useful in RNetica are:

"include_evidence_nodes". For the "markov_boundary" and "d_connected" relations indicates whether nodes with findings should be included in the result (they would normally not be included in the result).

"exclude_self". For the "ancestors", "descendents", "connected", and "d_connected" relations, the elements of nodelist are not initially added to the result.

Value

For is.NodeRelated() TRUE or FALSE, or NA if one of the input nodes was not active.

For GetNodeRelated() a list of NeticaNode objects which have the target relationship with one of the nodes in nodelist. There may be duplicates in this list.

Note

GetRelatedNodes() uses GetRelatedNodesMult_bn(), not GetRelatedNode_bn(), but that should not present any serious issues. Also, it always passes an empty list for the related_nodes arguments. Consequently, the "append", "union", "intersection", and "subtract" options don't make much sense. This is only a minor limitation as R provides similar functions.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: IsNodeRelated_bn(), GetRelatedNodes_bn(), GetRelatedNodesMult_bn()

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems.* Morgan–Kaufmann.

See Also

NeticaNode, NodeParents(), NodeChildren(), AddLink()

Examples

```
sess <- NeticaSession()
startSession(sess)

testnet <- CreateNetwork("ABCDEFG", session=sess)
###  A   D
###   \ / \
###    C   F - G
###   / \ /
###  B   E
A <- NewDiscreteNode(testnet,"A")
B <- NewDiscreteNode(testnet,"B")
C <- NewDiscreteNode(testnet,"C")
D <- NewDiscreteNode(testnet,"D")
E <- NewDiscreteNode(testnet,"E")
F <- NewDiscreteNode(testnet,"F")
G <- NewDiscreteNode(testnet,"G")

AddLink(A,C)
AddLink(B,C)

AddLink(C,D)
AddLink(C,E)

AddLink(D,F)
AddLink(E,F)

AddLink(F,G)

stopifnot(
```

```
  is.NodeRelated(A,C,"parent"),
  is.NodeRelated(D,C,"child"),
  is.NodeRelated(C,G,"ancestor"),
  is.NodeRelated(E,C,"descendent"),
  is.NodeRelated(A,B), ## Same as connected
  is.NodeRelated(D,E,"markov_blanket"),
  !is.NodeRelated(A,B,"d_connected"), ## No common ancestor
  is.NodeRelated(D,E,"d_connected") ## Common ancestor
)

stopifnot(
  setequal(GetRelatedNodes(F,"parents"),list(D,E)),
  setequal(GetRelatedNodes(C,"children"),list(D,E)),
  setequal(GetRelatedNodes(D,"descendents"),list(D,F,G)),
  setequal(GetRelatedNodes(E,"ancestors"),list(E,C,A,B)),
  setequal(GetRelatedNodes(E,"ancestors,exclude_self"),
           GetRelatedNodes(D,"ancestors,exclude_self")),
  setequal(GetRelatedNodes(A),list(A,B,C,D,E,F,G)), ##All nodes connected
  setequal(GetRelatedNodes(D,"markov_blanket"),list(C,E,F)),
  setequal(GetRelatedNodes(A,"d_connected"),list(A,C,D,E,F,G))
)

DeleteNetwork(testnet)
stopSession(sess)
```

`IsNodeDeterministic` *Determines if a node in a Netica Network is deterministic or not.*

Description

A node in a Bayesian network is deterministic if its value is determined by the states of its parents, that is if all conditional probabilities are 0 or 1.

Usage

```
IsNodeDeterministic(node)
```

Arguments

`node`	An active `NeticaNode` whose conditional probability table is to be tested.

Details

For discrete nodes, this returns `TRUE` if all the conditional probabilities are zero or one. It returns `FALSE` otherwise.

Value

`TRUE` if the conditional probability table for `node` is deterministic, `FALSE` otherwise. If the node is not active, or there is otherwise an error it returns `NA`.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: IsNodeDeterministic_bn()

See Also

NeticaNode, NodeParents(), NodeInputNames(), NodeStates()

Examples

```
sess <- NeticaSession()
startSession(sess)

ab <- CreateNetwork("AB", session=sess)
A <- NewDiscreteNode(ab,"A",c("A1","A2","A3"))
B <- NewDiscreteNode(ab,"B",c("B1","B2"))
AddLink(A,B)

##Undefined node is not deterministic.
stopifnot(!IsNodeDeterministic(A))

NodeProbs(A) <- c(0,1,0)
stopifnot(IsNodeDeterministic(A))

NodeProbs(A) <- c(1/3,1/3,1/3)
stopifnot(!IsNodeDeterministic(A))

NodeProbs(B) <- rbind(c(0,1), c(0,1), c(1,0))
stopifnot(IsNodeDeterministic(B))

DeleteNetwork(ab)
stopSession(sess)
```

JointProbability *Calculates the joint probability over several network nodes.*

Description

The Bayesian network, once compiled, gives the joint probability of all nodes in the network given the findings. This function calculates the joint probability over all of the nodes its argument and returns it as an array.

Usage

```
JointProbability(nodelist)
```

Arguments

nodelist	A list of active NeticaNode objects from the same network.

Details

This calculates the joint probability distribution over two, three or more variables in the same network. Calculating the joint probability is easy if all of the nodes are in the same clique, so one might want to use the function MakeCliqueNode(nodelist) before compiling the network to force the nodes in the same clique. The function can calculate the joint probability table for nodes not in the same clique, it just takes longer.

Value

A multidimensional array given the probabilities of the various configurations. The dimensions correspond to the variables in nodelist, and the dimnames of the result are the result of sapply(nodelist,NodeStates).

Note

One possible use for the joint probability function is to get a joint likelihood over the footprint nodes in an evidence model (see Almond et al, 1999; Almond & Mislevy, 1999). However, Netica currently does not support inserting a likelihood on a clique, just on a single node.

Author(s)

Russell Almond

References

Almond, R. G. & Mislevy, R. J. (1999) Graphical models and computerized adaptive testing. *Applied Psychological Measurement*, 23, 223–238.

Almond, R., Herskovits, E., Mislevy, R. J., & Steinberg, L. S. (1999). Transfer of information between system and evidence models. In Artificial Intelligence and Statistics 99, Proceedings (pp. 181–186). Morgan-Kaufmann

http://norsys.com/onLineAPIManual/index.html: JointProbability_bn()

See Also

NeticaNode,NodeBeliefs() MakeCliqueNode(), AddLink(), JunctionTreeReport(), MostProbableConfig()

Examples

```
sess <- NeticaSession()
startSession(sess)

EMSMMotif <- ReadNetworks(file.path(library(help="RNetica")$path,
                          "sampleNets","EMSMMotif.dne"), session=sess)

## Force Skills 1 and 2 into the same clique.
Skills12 <- NetworkFindNode(EMSMMotif,c("Skill1","Skill2"))
cn <- MakeCliqueNode(Skills12)
```

```
CompileNetwork(EMSMMotif)

## Prior Joint probability.
prior <- JointProbability(Skills12)
stopifnot (abs(sum(prior)-1) <.0001)

## Find observable nodes
obs <- NetworkNodesInSet(EMSMMotif,"Observable")

NodeFinding(obs$Obs1a1) <- "Right"
NodeFinding(obs$Obs1a2) <- "Wrong"

post <- JointProbability(GetClique(cn))
stopifnot (abs(sum(post)-1) <.0001)

DeleteNetwork(EMSMMotif)
stopSession(sess)
```

`JunctionTreeReport` *Produces a report about the junction tree from a compiled Netica network.*

Description

The process of compilation transforms the network into a junction tree – a tree of cliques in the original graph – that is more convenient computationally. The function `JunctionTreeReport(`*net*`)` produces a report on the junction tree. The function `NetworkCompiledSize(`*net*`)` reports on the size of the compiled network. The network must be compiled (`CompileNetwork(`*net*`)` must be called) before these functions are called.

Usage

```
JunctionTreeReport(net)
NetworkCompiledSize(net)
```

Arguments

`net`	An active and compiled `NeticaBN` object.

Value

For `JunctionTreeReport()` a character vector giving the report, one element per line.

For `NetworkCompiledSize()` a scalar value giving the size of the network.

Note

Currently, no attempt is made to parse the report, which has a fairly well structured format. Future versions may produce a report object instead.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: ReportJunctionTree_bn(), SizeCompiledNet_bn()

See Also

NeticaBN, CompileNetwork(), EliminationOrder(),

Examples

```
sess <- NeticaSession()
startSession(sess)

EMSMMotif <- ReadNetworks(file.path(library(help="RNetica")$path,
                          "sampleNets","EMSMMotif.dne"), session=sess)

CompileNetwork(EMSMMotif)

JunctionTreeReport(EMSMMotif)

NetworkCompiledSize(EMSMMotif)

DeleteNetwork(EMSMMotif)
stopSession(sess)
```

LearnCases *Learn Conditional Probability Tables from a Netica Case Stream*

Description

This function updates the conditional probabilities associated with the given list of nodes based on the findings associated with that node and its parents found in the `caseStream` argument, which should be a CaseStream object.

Usage

```
LearnCases(caseStream, nodelist, weight = 1)
```

Arguments

caseStream	This should be a CaseStream object, or else an object which can be made into a case stream: either a pathname for a case file, or a data frame of the format described in MemoryCaseStream. The case stream can be either opened or closed. If closed it is reopened before updating. In either case, it is closed at the end of the function. **Warning**, due to a bug in Netica, memory streams are not working and should not be used with Netica API 5.04 or earlier. See below.
nodelist	A list of active NeticaNode objects that reference the conditional probability tables to be updated.
weight	A multiplier for the weights of the cases in terms of number of observations. Negative weights unlearn previously learned cases.

Details

This is like calling the function LearnFindings repeatedly with the values of the nodes set to each of the case rows in turn. Thus, it updates the conditional probability tables for each nodes based on observed counts in the case files, taking the current probability and the NodeExperience as the prior distribution.

If the case stream has a column NumCases, then the weight assigned to Row j is weight*NumCases[j]. If the case stream does not have such a column, then it is treated as if each column has weight 1. (Among other purposes, this allows case data to be stored in a compact format where all of the possible cases are enumerated along with a count of repetitions.) Note that negative weights will unlearn cases.

Value

This function returns the CaseStream used in the analysis. This might have either been passed directly as the caseStream argument, or created from the value of the caseStream argument. In either case, the stream is closed at the end of the function.

Netica Bugs

In version 5.04 of the Netica API, there is a problem with using Memory Streams that seems to affect the functions LearnCases and LearnCPTs. Until this problem is fixed, most uses of Memory Streams will require file streams instead. Write the case file using write.CaseFile, and then create a file stream using CaseFileStream.

Note

To learn without using the current probabilities as priors, call DeleteNodeTable first.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: ReviseCPTsByCaseFile_bn()

See Also

NodeExperience, NodeProbs, NodeFinding, FadeCPT, LearnFindings, DeleteNodeTable, LearnCPTs

Examples

```
sess <- NeticaSession()
startSession(sess)

abb <- CreateNetwork("ABB", session=sess)
A <- NewDiscreteNode(abb,"A",c("A1","A2"))
B1 <- NewDiscreteNode(abb,"B1",c("B1","B2"))
B2 <- NewDiscreteNode(abb,"B2",c("B1","B2"))

AddLink(A,B1)
AddLink(A,B2)

A[] <- c(.5,.5)
NodeExperience(A) <- 10

B1["A1"] <- c(.8,.2)
B1["A2"] <- c(.2,.8)
B2["A1"] <- c(.8,.2)
B2["A2"] <- c(.2,.8)
NodeExperience(B1) <- c(10,10)
NodeExperience(B2) <- c(10,10)

casesabb <-
  data.frame(A=c("A1","A1","A1","A1","A1","A2","A2","A2","A2","A2"),
             B1=c("B1","B1","B1","B2","B2","B2","B2","B2","B1","B1"),
             B2=c("B1","B1","B1","B1","B2","B2","B2","B2","B2","B1"))
## LearnCases(casesabb,list(A,B1))
## There is currently a bug in Netica, so that this function does not
## work with memory streams.  As a work around, use proper file streams
## instead.

outfile <- tempfile("casesabb",fileext=".cas")
write.CaseFile(casesabb,outfile, session=sess)
LearnCases(outfile,list(A,B1))

## Probs for A & B1 modified, but B2 left alone
stopifnot(
  NodeExperience(A)==20,
  NodeExperience(B1)==c(15,15),
  NodeExperience(B2)==c(10,10),
  sum(abs(NodeProbs(A) - .5)) < .001,
  sum(abs(B1[["A1"]] - c(11,4)/15)) < .001,
  sum(abs(B1[["A2"]] - c(4,11)/15)) < .001,
  sum(abs(B2[["A1"]] - c(8,2)/10)) < .001,
  sum(abs(B2[["A2"]] - c(2,8)/10)) < .001
)

DeleteNetwork(abb)
```

```
stopSession(sess)
```

`LearnCPTs` *Learn Conditional Probability Tables with Missing Data.*

Description

This function updates the conditional probabilities associated with the given list of nodes based on the findings associated with that node and its parents found in the `caseStream` argument, which should be a `CaseStream` object. Unlike `LearnCases`, these algorithms can support cases with missing or latent variables.

Usage

```
LearnCPTs(caseStream, nodelist, method = "COUNTING", maxIters = 1000L, maxTol = 1e-06, weight = 1)
```

Arguments

`caseStream`	This should be a `CaseStream` object, or else an object which can be made into a case stream: either a pathname for a case file, or a data frame of the format described in `MemoryCaseStream`. The case stream can be either opened or closed. If closed it is reopened before updating. In either case, it is closed at the end of the function. **Warning**, due to a bug in Netica, memory streams are not working and should not be used with Netica API 5.04 or earlier. See below.
`nodelist`	A list of active `NeticaNode` objects that reference the conditional probability tables to be updated.
`method`	A character scalar giving the name of the method to be used. This should be one of "GRADIENT", "EM" or "COUNTING" (the default). See details.
`maxIters`	An integer scalar giving the maximum number of interactions for the EM and gradient decent algorithms.
`maxTol`	A real scalar giving the difference in log-likelihood required before the EM or gradient decent algorithms to be considered converged.
`weight`	A multiplier for the weights of the cases in terms of number of observations. Negative weights unlearn previously learned cases.

Details

This function attempts to update the conditional probability tables of the nodes named in `nodelist` using the data referenced in the first argument. Three different algorithms are available: *Counting*, *EM* and *Gradient Decent*. The *Counting* algorithm cannot handle cases with missing data or latent variables in the model. The `method` argument determines which method is used.

The *Counting* algorithm is the same as the one used in `LearnCases`. Cases where either the parent or the child variable is missing are ignored when updating the conditional probability table for the node, that is the neither affect the `NodeExperience` or the `NodeProbs`. As a consequence, models with latent variables cannot be fit with this algorithm.

The *EM* is similar to the *Counting* algorithms, but does more intelligent things with missing observations (particularly, missing parent variables). In particular, the complete data case of the *EM* algorithm is the same as the counting algorithm.

The *Gradient Decent* algorithm is an alternative iterative algorithm. According to the Netica documentation it is similar to back propagation in neural networks. Again according to Netica, it is faster than EM, but more likely to find a local maxima. It appears not to respect prior information about the conditional probability tables, and it sets the node experience to `-Inf`.

Both `EM` and `Gradient Decent` are an iterative algorithms. For these algorithms `maxIters` gives the maximum number of iterations, and `maxTol` gives the convergence criteria (required difference in log likelihood). These parameters are ignored for the *Counting* algorithm. Currently, Netica gives no indication of whether the algorithm terminated by achieving convergence (difference in log likelihood less than `maxTol`) or by exceeding `maxIters`. Norsys says they will fix this in an upcoming release.

If the case stream has a column `NumCases`, then the weight assigned to Row j is `weight*NumCases[j]`. If the case stream does not have such a column, then it is treated as if each column has weight 1. (Among other purposes, this allows case data to be stored in a compact format where all of the possible cases are enumerated along with a count of repetitions.) Note that negative weights will unlearn cases.

Value

Currently, `NULL` is returned. In the future, an object containing details about the convergence will be returned.

Netica Bugs

In version 5.04 of the Netica API, there is no indication of whether the call to LearnCPTs_bn has converged (terminated because the difference in log likelihood is less than `maxTol`) or not (terminated because the number of iterations exceeded `maxIters`). Norsys has indicated that they will add this functionality to a later release.

In version 5.04 of the Netica API, there is a problem with using Memory Streams that seems to affect the functions `LearnCases` and `LearnCPTs`. Until this problem is fixed, most uses of Memory Streams will require file streams instead. Write the case file using `write.CaseFile`, and then create a file stream using `CaseFileStream`.

Note

The `LearnCPTs` function will not update the conditional probability table of a node unless `NodeExperience` has been set for that node. Instead it will issue a warning and update the other nodes.

Author(s)

Russell G. Almond

References

`http://norsys.com/onLineAPIManual/index.html`: LearnCPTS_bn(), NewLearner_bn(), SetLearnerMaxTol_bn(), SetLearnerMaxTol_bn()

See Also

NodeExperience, NodeProbs, NodeFinding, FadeCPT, RetractNetFindings, LearnFindings
LearnCases

Examples

```
sess <- NeticaSession()
startSession(sess)

abb <- CreateNetwork("ABB", session=sess)
A <- NewDiscreteNode(abb,"A",c("A1","A2"))
B1 <- NewDiscreteNode(abb,"B1",c("B1","B2"))
B2 <- NewDiscreteNode(abb,"B2",c("B1","B2"))

AddLink(A,B1)
AddLink(A,B2)

A[] <- c(.5,.5)
NodeExperience(A) <- 10

B1["A1"] <- c(.8,.2)
B1["A2"] <- c(.2,.8)
B2["A1"] <- c(.8,.2)
B2["A2"] <- c(.2,.8)
NodeExperience(B1) <- c(10,10)
NodeExperience(B2) <- c(10,10)
casesabb <-
  data.frame(A=c("A1","A1","A1","A1","A1","A2","A2","A2","A2","A2"),
             B1=c("B1","B1","B1","B2","B2","B2","B2","B2","B1","B1"),
             B2=c("B1","B1","B1","B1","B2","B2","B2","B2","B2","B1"))
## LearnCPTs(casesabb,list(A,B1))
## There is currently a bug in Netica, so that this function does not
## work with memory streams.  As a work around, use proper file streams
## instead.

outfile <- tempfile("casesabb",fileext=".cas")
write.CaseFile(casesabb,outfile, session=sess)
LearnCPTs(outfile,list(A,B1))

## Probs for A & B1 modified, but B2 left alone
stopifnot(
  NodeExperience(A)==20,
  NodeExperience(B1)==c(15,15),
  NodeExperience(B2)==c(10,10),
  sum(abs(NodeProbs(A) - .5)) < .001,
  sum(abs(B1[["A1"]] - c(11,4)/15)) < .001,
  sum(abs(B1[["A2"]] - c(4,11)/15)) < .001,
  sum(abs(B2[["A1"]] - c(8,2)/10)) < .001,
  sum(abs(B2[["A2"]] - c(2,8)/10)) < .001
)
```

```
## Missing Data
## NAs in parents affect both parent and child.
casesabb1 <-
  data.frame(A=c("A1","A1","NA","A1","A1","A2","A2","A2","A2","A2"),
             B1=c("B1","B1","B1","B2","B2","B2","B2","NA","B1","B1"),
             B2=c("B1","B1","B1","NA","B2","B2","B2","B2","B2","B1"))

outfile1 <- tempfile("casesabb1",fileext=".cas")
write.CaseFile(casesabb1,outfile1, session=sess)
LearnCPTs(outfile1,list(A,B1,B2))

stopifnot(
  NodeExperience(A)==29,
  NodeExperience(B1)==c(19,19),
  NodeExperience(B2)==c(13,15),
  sum(abs(NodeProbs(A) - c(14,15)/29)) < .001,
  sum(abs(B1[["A1"]] - c(13,6)/19)) < .001,
  sum(abs(B1[["A2"]] - c(6,13)/19)) < .001,
  sum(abs(B2[["A1"]] - c(10,3)/13)) < .001,
  sum(abs(B2[["A2"]] - c(3,12)/15)) < .001
)

DeleteNetwork(abb)

####################################
## Start again with EM learning.

abb <- CreateNetwork("ABB", session=sess)
A <- NewDiscreteNode(abb,"A",c("A1","A2"))
B1 <- NewDiscreteNode(abb,"B1",c("B1","B2"))
B2 <- NewDiscreteNode(abb,"B2",c("B1","B2"))

AddLink(A,B1)
AddLink(A,B2)

A[] <- c(.5,.5)
NodeExperience(A) <- 10

B1["A1"] <- c(.8,.2)
B1["A2"] <- c(.2,.8)
B2["A1"] <- c(.8,.2)
B2["A2"] <- c(.2,.8)
NodeExperience(B1) <- c(10,10)
NodeExperience(B2) <- c(10,10)
casesabb <-
  data.frame(A=c("A1","A1","A1","A1","A1","A2","A2","A2","A2","A2"),
             B1=c("B1","B1","B1","B2","B2","B2","B2","B2","B1","B1"),
             B2=c("B1","B1","B1","B1","B2","B2","B2","B2","B2","B1"))
## LearnCPTs(casesabb,list(A,B1),method="EM")
## There is currently a bug in Netica, so that this function does not
## work with memory streams.  As a work around, use proper file streams
## instead.
```

```
outfile <- tempfile("casesabb",fileext=".cas")
write.CaseFile(casesabb,outfile, session=sess)
LearnCPTs(outfile,list(A,B1),method="EM")

## Complete data, this should look identical to the counting case.
## Note that NodeExperience is no longer an integer
stopifnot(
  abs(NodeExperience(A)-20) < .001,
  sum(abs(NodeExperience(B1)-c(15,15))) < .001,
  NodeExperience(B2)==c(10,10),
  sum(abs(NodeProbs(A) - .5)) < .001,
  sum(abs(B1[["A1"]] - c(11,4)/15)) < .001,
  sum(abs(B1[["A2"]] - c(4,11)/15)) < .001,
  sum(abs(B2[["A1"]] - c(8,2)/10)) < .001,
  sum(abs(B2[["A2"]] - c(2,8)/10)) < .001
)


## Missing Data
## EM deals more intelligently with missing data.
casesabb1 <-
  data.frame(A=c("A1","A1","NA","A1","A1","A2","A2","A2","A2","A2"),
             B1=c("B1","B1","B1","B2","B2","B2","B2","NA","B1","B1"),
             B2=c("B1","B1","B1","NA","B2","B2","B2","B2","B2","B1"))

outfile1 <- tempfile("casesabb1",fileext=".cas")
write.CaseFile(casesabb1,outfile1, session=sess)
LearnCPTs(outfile1,list(A,B1,B2),method="EM")

stopifnot(
  NodeExperience(A)>29,
  NodeExperience(B1)>c(19,19),
  NodeExperience(B2)>c(13,15)
)

## EM can handle complete latent variable case.
casesabb2 <-
  data.frame(B1=c("B1","B1","B1","B2","B2","B2","B2","NA","B1","B1"),
             B2=c("B1","B1","B1","NA","B2","B2","B2","B2","B2","B1"))

outfile2 <- tempfile("casesabb2",fileext=".cas")
write.CaseFile(casesabb1,outfile2, session=sess)
LearnCPTs(outfile1,list(A,B1,B2),method="EM")

stopifnot(
  NodeExperience(A)>39,
  NodeExperience(B1)>c(24,23),
  NodeExperience(B2)>c(14,20)
)


DeleteNetwork(abb)
```

```
####################################
## One more time with Gradient Decent learning.

abb <- CreateNetwork("ABB", session=sess)
A <- NewDiscreteNode(abb,"A",c("A1","A2"))
B1 <- NewDiscreteNode(abb,"B1",c("B1","B2"))
B2 <- NewDiscreteNode(abb,"B2",c("B1","B2"))

AddLink(A,B1)
AddLink(A,B2)

A[] <- c(.5,.5)
NodeExperience(A) <- 10

B1["A1"] <- c(.8,.2)
B1["A2"] <- c(.2,.8)
B2["A1"] <- c(.8,.2)
B2["A2"] <- c(.2,.8)
NodeExperience(B1) <- c(10,10)
NodeExperience(B2) <- c(10,10)
casesabb <-
  data.frame(A=c("A1","A1","A1","A1","A1","A2","A2","A2","A2","A2"),
             B1=c("B1","B1","B1","B2","B2","B2","B2","B2","B1","B1"),
             B2=c("B1","B1","B1","B1","B2","B2","B2","B2","B2","B1"))
## LearnCPTs(casesabb,list(A,B1),method="GRADIENT")
## There is currently a bug in Netica, so that this function does not
## work with memory streams.  As a work around, use proper file streams
## instead.

outfile <- tempfile("casesabb",fileext=".cas")
write.CaseFile(casesabb,outfile, session=sess)
LearnCPTs(outfile,list(A,B1),method="GRADIENT")

## Complete data, this should look identical to the counting case.
## Note that NodeExperience is no longer used, and the posterior
## distribution no longer reflects the prior.
stopifnot(
  NodeExperience(B2)==c(10,10),
  sum(abs(NodeProbs(A) - .5)) < .001,
  sum(abs(B1[["A1"]] - c(3,2)/5)) < .001,
  sum(abs(B1[["A2"]] - c(2,3)/5)) < .001,
  sum(abs(B2[["A1"]] - c(8,2)/10)) < .001,
  sum(abs(B2[["A2"]] - c(2,8)/10)) < .001
)

## Gradient algorithm sets experience to -infinity, so need to reset.
NodeExperience(A) <- 10
NodeExperience(B1) <- c(10,10)
NodeExperience(B2) <- c(10,10)


## Missing Data
## GRADIENT deals more intelligently with missing data.
```

```
casesabb1 <-
  data.frame(A=c("A1","A1","NA","A1","A1","A2","A2","A2","A2","A2"),
             B1=c("B1","B1","B1","B2","B2","B2","B2","NA","B1","B1"),
             B2=c("B1","B1","B1","NA","B2","B2","B2","B2","B2","B1"))

outfile1 <- tempfile("casesabb1",fileext=".cas")
write.CaseFile(casesabb1,outfile1, session=sess)
LearnCPTs(outfile1,list(A,B1,B2),method="GRADIENT")

## Gradient algorithm sets experience to -infinity, so need to reset.
NodeExperience(A) <- 10
NodeExperience(B1) <- c(10,10)
NodeExperience(B2) <- c(10,10)

## GRADIENT can handle complete latent variable case.
casesabb2 <-
  data.frame(B1=c("B1","B1","B1","B2","B2","B2","B2","NA","B1","B1"),
             B2=c("B1","B1","B1","NA","B2","B2","B2","B2","B2","B1"))

outfile2 <- tempfile("casesabb2",fileext=".cas")
write.CaseFile(casesabb1,outfile2, session=sess)
LearnCPTs(outfile1,list(A,B1,B2),method="GRADIENT")

DeleteNetwork(abb)
stopSession(sess)
```

`LearnFindings` *Learn Netica conditional probabilities from findings.*

Description

This function updates the conditional probabilities associated with the given list of nodes based on the findings associated with that node and its parents. Before calling this function the findings to be learned should be set using `NodeFinding`.

Usage

```
LearnFindings(nodes, weight = 1)
```

Arguments

`nodes`	A list of active `NeticaNode` objects that reference the conditional probability tables to be updated.
`weight`	The weight of the current observation in terms of number of observations. Negative weights unlearn previously learned cases.

Details

For the purposes of this function, Netica regards the probabilities in Row j of the CPT for each selected node as having an independent Dirichlet distribution with parameters $(a_{j1}, \ldots, a_{jK}) = n_j(p_{j1}, \ldots, p_{jK})$ where p_{jk} is the probability associated with State k in Row j and n_j is the experience associated with Row j.

If `LearnFindings` is called on a node which is currently instantiated to State k and whose parents are currently instantiated to the configuration which selects Row j of the table, then $n'_j = n_j + weight$ and $a'_{jk} = a_{jk} + weight$ with all other values remaining the same. The new conditional probabilities are $p'_{jk} = a'_{jk}/n'_j$.

The function `FadeCPT` is often used between calls to `LearnFindings` to down weight old cases when the conditional probabilities are thought to be changing slowly over time.

Value

This returns the list of nodes whose conditional probability tables have been modified.

Note

Do not confuse this function with `NodeFinding`. `NodeFinding` instantiates a node and updates all of the other beliefs associated with a node to reflect the new evidence. `LearnFindings` incorporates the current case (the currently instantiated set of findings) into the CPTs for the nodes.

The `LearnFindings` function will not update the conditional probability table of a node unless `NodeExperience` has been set for that node. Instead it will issue a warning and update the other nodes.

Author(s)

Russell G. Almond

References

http://norsys.com/onLineAPIManual/index.html: ReviseCPTsByFindings_bn()

See Also

`NodeExperience`, `NodeProbs`, `NodeFinding`, `FadeCPT`, `RetractNetFindings`, `LearnCases`, `LearnCPTs`

Examples

```
sess <- NeticaSession()
startSession(sess)

abb <- CreateNetwork("ABB", session=sess)
A <- NewDiscreteNode(abb,"A",c("A1","A2"))
B1 <- NewDiscreteNode(abb,"B1",c("B1","B2"))
B2 <- NewDiscreteNode(abb,"B2",c("B1","B2"))

AddLink(A,B1)
AddLink(A,B2)
```

```
A[] <- c(.5,.5)
NodeExperience(A) <- 10

B1["A1"] <- c(.8,.2)
B1["A2"] <- c(.2,.8)
B2["A1"] <- c(.8,.2)
B2["A2"] <- c(.2,.8)
NodeExperience(B1) <- c(10,10)
NodeExperience(B2) <- c(10,10)

## First Case
NodeFinding(A) <- "A1"
NodeFinding(B1) <- "B1"
NodeFinding(B2) <- "B2"

LearnFindings(list(A,B1))
## Probs for A & B1 modified, but B2 left alone
stopifnot(
  NodeExperience(A)==11,
  NodeExperience(B1)==c(11,10),
  NodeExperience(B2)==c(10,10),
  sum(abs(NodeProbs(A) - c(6,5)/11)) < .001,
  sum(abs(B1[["A1"]] - c(9,2)/11)) < .001,
  sum(abs(B1[["A2"]] - c(2,8)/10)) < .001,
  sum(abs(B2[["A1"]] - c(8,2)/10)) < .001,
  sum(abs(B2[["A2"]] - c(2,8)/10)) < .001
)

## Second Case
RetractNetFindings(abb)
NodeFinding(A) <- "A2"
NodeFinding(B1) <- "B1"
NodeFinding(B2) <- "B1"

LearnFindings(list(A,B1))
## Probs for A & B1 modified, but B2 left alone
stopifnot(
  NodeExperience(A)==12,
  NodeExperience(B1)==c(11,11),
  NodeExperience(B2)==c(10,10),
  sum(abs(NodeProbs(A) - c(6,6)/12)) < .001,
  sum(abs(B1[["A1"]] - c(9,2)/11)) < .001,
  sum(abs(B1[["A2"]] - c(3,8)/11)) < .001,
  sum(abs(B2[["A1"]] - c(8,2)/10)) < .001,
  sum(abs(B2[["A2"]] - c(2,8)/10)) < .001
)

## Retract Case 2
LearnFindings(list(A,B1),-1)
## Back to where we were before Case 1
stopifnot(
  NodeExperience(A)==11,
```

```
  NodeExperience(B1)==c(11,10),
  NodeExperience(B2)==c(10,10),
  sum(abs(NodeProbs(A) - c(6,5)/11)) < .001,
  sum(abs(B1[["A1"]] - c(9,2)/11)) < .001,
  sum(abs(B1[["A2"]] - c(2,8)/10)) < .001,
  sum(abs(B2[["A1"]] - c(8,2)/10)) < .001,
  sum(abs(B2[["A2"]] - c(2,8)/10)) < .001
)

DeleteNetwork(abb)
stopSession(sess)
```

MakeCliqueNode *Forces a collection of nodes in a Netica network to be in the same clique.*

Description

When a junction tree is compiled, if the nodes are in the same clique, it is easier to calculate their joint probability. The function `MakeCliqueNode(`*nodelist*`)` forces the nodes in `nodelist` by making a special one state clique node with all of the nodes in `nodelist` as a parent.

Usage

```
MakeCliqueNode(nodelist)
is.CliqueNode(x)
GetClique(cliquenode)
```

Arguments

`nodelist`	A list of active `NeticaNode` objects from the same network.
`x`	An object to be tested to see if it is a clique node.
`cliquenode`	A `CliqueNode` to be queried.

Details

It is substantially easier to calculate the joint probability of a number of nodes if they are all in the same clique (see `JointProbability(`*nodelist*`)`. If it is known that such a query will be common, the analyst can take steps to force the nodes into the same clique if required. The Student Model/Evidence Model algorithm of Almond and Mislevy (1999) also requires that the student model variables that are referenced in an evidence model all be in the same clique (although this algorithm is not currently supported by Netica).

A node and its parents is always a clique or a subset of a clique in the junction tree (see `CompileNetwork()` or `JunctionTreeReport()`). This function forces nodes into the same clique by creating a new `CliqueNode` and making all of the nodes in `nodelist` parents of the new node.

The CliqueNode is a subclass of NeticaNode. It has a number of special features. It's name is always "Clique" followed by a number. It only has one state, and it has a special "clique" field which records the nodelist used to create it. The function is.CliqueNode() tests a node to see if it is a clique node, and the function GetClique(*node*) retrieves the nodelist. (This should not be set manually).

The CliqueNode objects should, for the most part, behave like regular nodes. However, it is almost certainly a mistake to try and set findings on a CliqueNode.

Value

The function MakeCliqueNode(*nodelist*) returns a new CliqueNode object whose parents are the variables in nodelist. This behaves in most respects like an ordinary node, but it would almost certainly be a mistake to try and enter findings for this node. In particular, deleting the clique node will no longer constrain its parents to be in the same clique (although other connections in the network may cause the nodes to be placed in the same clique).

The function is.CliqueNode(*x*) returns a logical value which is true if x is a clique node.

The function GetClique(*node*) returns the nodelist used to create the clique node.

Note

Clique nodes only last for the R session that was used to create them. After that, they will appear like ordinary nodes. They will still be present in the network, but the special "clique" attribute will be lost.

Currently Netica only allows virtual evidence at the node level (NodeLikelihood()). I'm lobbying to get Netica to support it at the clique level as well. At which point, this function becomes extremely useful.

Author(s)

Russell Almond

References

Almond, R. G. & Mislevy, R. J. (1999) Graphical models and computerized adaptive testing. *Applied Psychological Measurement*, 23, 223-238.

Almond, R., Herskovits, E., Mislevy, R. J., & Steinberg, L. S. (1999). Transfer of information between system and evidence models. In Artificial Intelligence and Statistics 99, Proceedings (pp. 181–186). Morgan-Kaufmann

http://norsys.com/onLineAPIManual/index.html: See the NeticaEx function FormCliqueWith is the documentation for JointProbability_bn()

See Also

CliqueNode, NeticaNode, JointProbability(), AddLink(), JunctionTreeReport()

Examples

```
sess <- NeticaSession()
startSession(sess)

EMSMSystem <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","System.dne"), session=sess)

CompileNetwork(EMSMSystem)
## Note that Skill1 and Skill2 are in different cliques
JunctionTreeReport(EMSMSystem)

Skills12 <- NetworkFindNode(EMSMSystem,c("Skill1","Skill2"))
cn <- MakeCliqueNode(Skills12)
cnclique <- GetClique(cn)

stopifnot(
  is.CliqueNode(cn),
  setequal(sapply(cnclique,NodeName),sapply(Skills12,NodeName))
)

CompileNetwork(EMSMSystem)
## Note that Skill1 and Skill2 are in different cliques
JunctionTreeReport(EMSMSystem)

DeleteNodes(cn) ## This clears the clique.

DeleteNetwork(EMSMSystem)
stopSession(sess)
```

MemoryCaseStream-class

Class "MemoryCaseStream"

Description

This object is subclass of CaseStream so it is a wrapper around a Netica stream which is used to read/write cases. In this subclass, the case stream is associated with a data frame containing the dase file information. The function CaseMemoryStream is the constructor. the case stream is associated with a memory buffer that corresponds to an R data.frame object. The function MemoryStreamContents accesses the contents as a data frame.

Details

A Netica case file has a format that very much resembles the output of write.table. The first row is a header row, which contains the names of the variables, the second and subsequent rows contain a set of findings: an assignment of values to the nodes indicated in the columns. There are no row numbers, and the separator and missing value codes are the values of CaseFileDelimiter(), and CaseFileMissingCode() respectively.

In addition to columns representing variables, two special columns are allowed. The column named "IDnum", if present should contain integers which correspond to ID numbers for the cases (this correspond to the id argument of WriteFindings). The column named "NumCases" should contain number values and this allows rows to be differentially weighted (this correspond to the freq argument of WriteFindings).

A simple way to convert a data frame into a set of cases for use with various Netica functions that use cases would be to write the data frame to a file of the proper format, and then create a CaseFileStream on the just written file. The MemoryCaseStream shortcuts that process by writing the data frame to a memory buffer and then creating a stream around the memory buffer. Like the FileCaseStream, the MemoryCaseStream is a subclass of CaseStream and follows the same conventions.

The function CaseMemoryStream opens a new memory stream using data.frame as the source. If data.frame is NULL a new memory stream for writing is created. The function CloseCaseStream closes an open case stream (and is harmless if the stream is already closed. Although RNetica tries to close open case streams when they are garbage collected, users should not count on this behavior and should close them manually. Also be aware that all case streams are automatically closed when R is closes or RNetica is unloaded. The function isCaseStreamOpen tests to see if the stream is open or closed. The function OpenCaseStream if called on a closed MemoryCaseStream will reopen the stream in Netica using the current value of MemoryStreamContents as the source. (If called on an open stream it will do nothing but issue a warning).

The function getCaseStreamDataFrameName provides the value of label when the stream was created.

Extends

Class "CaseStream", directly.

All reference classes extend and inherit methods from "envRefClass". Note that because this is a reference class unlike traditional S3 and S4 classes it can be destructively modified. Also fields (slots) are accessed using the '$' operator.

Netica Bugs

In version 5.04 of the Netica API, there is a problem with using Memory Streams that seems to affect the functions LearnCases and LearnCPTs. Until this problem is fixed, most uses of Memory Streams will require file streams instead. Write the case file using write.CaseFile, and then create a file stream using CaseFileStream.

Fields

Note these should be regarded as read-only from user code.

Name: Object of class character identifier for stream. Default is the expression used to reference the data.

Session: Object of class NeticaSession

Session: Object of class NeticaSession a link to the session in which this case stream was created.

Netica_Case_Stream: Object of class externalptr a pointer to the case stream in Netica memory.

`Case_Stream_Position`: Object of class `integer` the number of the last read/writen record. This is `NA` if the end of the file has been reached.

`Case_Stream_Lastid`: Object of class `integer` the ID number of the last read/written record.

`Case_Stream_Lastfreq`: Object of class `numeric` giving the frequence of the last read/written record. This is used as a weight in learning applications.

`Case_Stream_DataFrameName`: Object of class `character` giving the expression used for the data frame.

`Case_Stream_DataFrame`: Object of class `data.frame` or `NULL` the data object that is the contents of the buffer, or `NULL` if the stream was created for writing.

`Case_Stream_Buffer`: Object of class `externalptr` used for an R-side string buffer, currently not used.

Methods

`open()`: Opens a connection too the file in Netica.

`show()`: Provides a description of the field

`initialize(Name, Session, Case_Stream_Path, ...)`: internal constructor; user code should use `CaseFileStream`.

The following methods are inherited (from `CaseStream`): close ("CaseStream"), isActive ("CaseStream"), isOpen ("CaseStream"), show ("CaseStream"), clearErrors ("CaseStream"), reportErrors ("CaseStream"), initialize ("CaseStream")

Note

In version 0.5 of RNetica, this class was renamed. It is now called `MemoryCaseStream` and the constructor is called `CaseMemoryStream` (while previously the class and the constructore had the same name). This matches the usage of `FileCaseStream` and its constructor `CaseFileStream`. This is not likely to be a problem as memory streams are not working well.

`MemoryCaseStreams` are most useful for small to medium size data frames. Larger data frames are probably better handled through case files.

Internally, a weak reference system is used to keep a list of Netica stream objects which need to be closed when RNetica is unloaded. Stream objects should also be forced closed when garbage collected. The weak reference system is somewhat experimental, so well designed code should manually close the streams when the program is through with it.

Stream objects are fragile, and will not survive saving and restoring an R session. However, the object retains information about itself, so that calling `OpenCaseStream` on the saved object, should reopen the stream. Note that any position information will be lost.

Author(s)

Russell Almond

References

`http://norsys.com/onLineAPIManual/index.html`: NewMemoryStream_ns(), `http://homepage.stat.uiowa.edu/~luke/R/references/weakfinex.html`

See Also

See CaseStream for the superclass and FileCaseStream for a sibling class. The function CaseMemoryStream is the constructor.

CaseFileDelimiter, CaseFileMissingCode, WriteFindings, ReadFindings, MemoryStreamContents

Examples

```
sess <- NeticaSession()
startSession(sess)

abc <- CreateNetwork("ABC", session=sess)
A <- NewDiscreteNode(abc,"A",c("A1","A2","A3","A4"))
B <- NewDiscreteNode(abc,"B",c("B1","B2","B3"))
C <- NewDiscreteNode(abc,"C",c("C1","C2"))

AddLink(A,B)
AddLink(A,C)
AddLink(B,C)

## This is the file written in CaseFileStream help.
casefile <- file.path(library(help="RNetica")$path,
                          "testData","abctestcases.cas")

CaseFileDelimiter("\t", session=sess)
CaseFileMissingCode("*", session=sess)
cases <- read.CaseFile(casefile, session=sess)

memstream <- CaseMemoryStream(cases, session=sess)


##Case 1
memstream <- ReadFindings(list(A,B,C),memstream,"FIRST")
stopifnot(NodeFinding(A) == "A1",
          NodeFinding(B) == "B1",
          NodeFinding(C) == "C1",
          getCaseStreamLastId(memstream)==1001,
          abs(getCaseStreamLastFreq(memstream)-1.0) <.0001)

##Case 2
memstream <- ReadFindings(list(A,B,C),memstream,"NEXT")
stopifnot(NodeFinding(A) == "A2",
          NodeFinding(B) == "B2",
          NodeFinding(C) == "C2",
          getCaseStreamLastId(memstream)==1002,
          abs(getCaseStreamLastFreq(memstream)-2.0) <.0001)

##Case 3
memstream <- ReadFindings(list(A,B,C),memstream,"NEXT")
stopifnot(NodeFinding(A) == "A3",
          NodeFinding(B) == "B3",
          NodeFinding(C) == "@NO FINDING",
          getCaseStreamLastId(memstream)==1003,
```

```
          abs(getCaseStreamLastFreq(memstream)-1.0) <.0001)

## EOF
memstream <- ReadFindings(list(A,B,C),memstream,"NEXT")
stopifnot (is.na(getCaseStreamPos(memstream)))


##Clean Up
CloseCaseStream(memstream)
DeleteNetwork(abc)

stopSession(sess)
```

`MemoryStreamContents` *Access the contents of a MemoryCaseStream*

Description

This function returns the contents of a `MemoryCaseStream`'s internal buffer as a `data.frame`. Alternatively, it sets the contents of the buffer to a given data frame.

Usage

```
MemoryStreamContents(stream)
MemoryStreamContents(stream) <- value
```

Arguments

`stream`	A `MemoryCaseStream` object whose contents is to be access.
`value`	Either a data frame giving the new value (see details), or else `NULL`.

Details

A set of cases for a Netica network corresponds to a `data.frame`. The columns represent nodes in the graph, and the values in that particular column correspond to findings for that node: a particular instantiation for that state, with a value of `NA` if the state of that node is unknown.

In addition to columns representing variables, two special columns are allowed. The column named "IDnum", if present should contain integers which correspond to ID numbers for the cases (this correspond to the `id` argument of `WriteFindings`). The column named "NumCases" should contain number values and this allows rows to be differentially weighted (this correspond to the `freq` argument of `WriteFindings`).

A `MemoryCaseStream` contains an R data frame object written out in string form. This function converts between the internal string object and the data frame representation. When called as `MemoryStreamContents(`*stream*`)` it reads the current value of the stream and converts it to a data frame. When called as setter function, it converts the value into a string and focuses the `MemoryCaseStream` object on this string.

Setting the contents to NULL creates a new empty stream buffer inside of the stream object. This is useful for creating a blank buffer for writing cases.

The code MemoryCaseStream object maintains a cached copy of the data frame associated with the memory stream. Calling the function in either the setter or getter form updates that cache. Calling this function when the stream is closed, will access the cached copy. In the case of the setter form, this will updated the cached value, and if the stream is reopened, it will focus on the new cached value. Note that if the stream is closed before MemoryStreamContents is called, then the value returned will be the cached value created when MemoryStreamContents was last called, or when the stream is opened.

Value

A data frame which corresponds to the contents of the stream buffer, or NULL if the stream buffer is empty.

Note

The cached value of the stream can be accessed with the expression *stream*$Case_Stream_DataFrame. While it is almost certainly a mistake to set this value directly, there may be situations (e.g., avoiding duplicating the data frame when the stream is essentially open for reading only) where it is useful. On the other hand, there may be situations where it is useful to read the cached value without forcing a reread of the memory buffer.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: SetStreamContents_ns(),

See Also

CaseFileDelimiter, CaseFileMissingCode, WriteFindings, ReadFindings, MemoryCaseStream, CaseStream

Examples

```
sess <- NeticaSession()
startSession(sess)

casefile <- file.path(library(help="RNetica")$path,
                          "testData","abctestcases.cas")

CaseFileDelimiter("\t", session=sess)
CaseFileMissingCode("*", session=sess)
cases <- read.CaseFile(casefile, session=sess)

memstream <- CaseMemoryStream(cases, session=sess)

## Should be the same as cases
```

```
stopifnot(all.equal(MemoryStreamContents(memstream),cases))


MemoryStreamContents(memstream) <- cases

CloseCaseStream(memstream) ## Don't forget to read off the value
                           ## first if needed before closing.

stopifnot(!isCaseStreamOpen(memstream))
## This should return the cached value.
MemoryStreamContents(memstream)

## Will clear stream when next open
MemoryStreamContents(memstream) <- NULL

OpenCaseStream(memstream)
stopifnot(is.null(MemoryStreamContents(memstream)))

CloseCaseStream(memstream)

## Second test, do this from scratch.

casesabb <-
  data.frame(IDnum=1001:1010,NumCases=rep(1,10),
             A=c("A1","A1","A1","A1","A1","A2","A2","A2","A2","A2"),
             B1=c("B1","B1","B1","B2","B2","B2","B2","B2","B1","B1"),
             B2=c("B1","B1","B1","B1","B2","B2","B2","B2","B2","B1"))

abbstream <- CaseMemoryStream(casesabb, session=sess)
MemoryStreamContents(abbstream)
CloseCaseStream(abbstream)

abb <- CreateNetwork("ABB", session=sess)
A <- NewDiscreteNode(abb,"A",c("A1","A2"))
B1 <- NewDiscreteNode(abb,"B1",c("B1","B2"))
B2 <- NewDiscreteNode(abb,"B2",c("B1","B2"))

AddLink(A,B1)
AddLink(A,B2)

A[] <- c(.5,.5)
NodeExperience(A) <- 10

B1["A1"] <- c(.8,.2)
B1["A2"] <- c(.2,.8)
B2["A1"] <- c(.8,.2)
B2["A2"] <- c(.2,.8)
NodeExperience(B1) <- c(10,10)
NodeExperience(B2) <- c(10,10)

abbstream <- CaseMemoryStream(casesabb, session=sess)
## This does not appear to work correctly
abbstream <- ReadFindings(list(A,B1,B2),abbstream,"FIRST")
```

```
NodeFinding(A)
NodeFinding(B1)
NodeFinding(B2)

CloseCaseStream(abbstream)
DeleteNetwork(abb)
stopSession(sess)
```

`MostProbableConfig` *Finds the configuration of the nodes most likely to have lead to observed findings.*

Description

Findings a set of values for each of the nodes in `nodelist` such that the probability of that value set is highest given the state of any findings entered into the network. This is sometimes called the "Most Probable Explanation" for the findings.

Usage

```
MostProbableConfig(net,nth = 0)
```

Arguments

`net`	An active and compiled `NeticaBN`.
`nth`	Leave this at its default value of zero, it is for future expansion.

Details

The most probable configuration of the nodes in the Bayesian network is the set of values for each of the nodes in the network which have the highest joint probability. This may or may not be thee same as setting the value of each node to the value that maximizes its `NodeBeliefs()`. Pearl (1988) describes a special max-propagation algorithm which can calculate the most likely configuration of nodes in a Bayesian network. This function runs that algorithm. The probability that is maximized is the posterior probability given the findings.

Note that this produces a configuration over all of the nodes in the network, not just the nodes in some particular set. The Netica documentation suggests running `AbsorbNodes()` over the unnecessary nodes first. Another possibility (if the set of interesting nodes is small) is to call `JointProbability()` on the affected nodes and then find the max of that.

Value

A character vector whose names are the names of the nodes in the network (see `NetworkAllNodes(net)`) and whose values are the names of the states that maximize the posterior probability given the findings.

Warning

The documentation for the Netica function MostProbableConfig_bn() states that likelihood findings (NodeLikelihood()) are not handled properly in MostProbableConfig(). Seems to indicate that this works properly, but some caution is still advised.

Note

The Bayesian network literature also discusses algorithms for the 2nd, 3rd, 4th, etc. most likely findings. These algorithms are slightly more difficult to implement, but are possible on future plans for the Netica API, as it offers the nth argument to the function MostProbableConfig_bn(). At this point in time, it is an error to set nth to anything but 0.

Author(s)

Russell Almond

References

Pearl, J. (1988) *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann.

http://norsys.com/onLineAPIManual/index.html: MostProbableConfig_bn()

See Also

NeticaBN, NodeBeliefs(), EnterNegativeFinding(), RetractNodeFinding(), NodeFinding() JointProbability(), FindingsProbability()

Examples

```
sess <- NeticaSession()
startSession(sess)

EMSMMotif <- ReadNetworks(file.path(library(help="RNetica")$path,
                          "sampleNets","EMSMMotif.dne"), session=sess)

CompileNetwork(EMSMMotif)
obs <- NetworkNodesInSet(EMSMMotif,"Observable")
prof <- NetworkNodesInSet(EMSMMotif,"Proficiency")

NodeFinding(obs$Obs1a1) <- "Right"
NodeFinding(obs$Obs1a2) <- "Wrong"
NodeFinding(obs$Obs1b1) <- "Right"
NodeFinding(obs$Obs1b2) <- "Wrong"

mpe <- MostProbableConfig(EMSMMotif)

## Observed values should be set at their findings level.
stopifnot (
  mpe$Obs1a1 == "Right",
  mpe$Obs1a2 == "Wrong",
```

```
  mpe$Obs1b1 == "Right",
  mpe$Obs1b2 == "Wrong"
)

## MPE for just proficiency variables.
mpe[names(prof)]

DeleteNetwork(EMSMMotif)
stopSession(sess)
```

`MutualInfo` *Calculates strength of relationship between two nodes in a network*

Description

The mutual information is a measure of how closely related one node is to another, i.e., how much information each node in `nodelist` provides about the `target` node. The expression `MutualInfo(`*target*`,` *nodelist*`)` calculates the mutual information of each node in `nodelist` with `target`.

The function `VarianceOfReal()` is similar, but instead it measures the reduction in variance of the target. The `target` node must be continuous or have numeric values assigned to all levels using `NodeLevels`.

Usage

```
MutualInfo(target, nodelist)
VarianceOfReal(target, nodelist)
```

Arguments

`target`	An active `NeticaNode` object that is the target of inference (i.e., we want to find the influence of other nodes on this node).
`nodelist`	A non-empty list of active `NeticaNode` objects whose effect on target is desired.

Details

The mutual information between two discrete variables is defined as:

$$MI(X,Y) = \sum_{x,y} \Pr(x,y) \log \frac{\Pr(x,y)}{\Pr(x)\Pr(y)}.$$

It is a measure of how much information `X` provides about `Y`. This measure is appropriate when both `X` and `Y` are discrete variables.

Mutual information is often used to select the next best variable to test (in the educational context, this would be the next item to select for an adaptive test). The highest value of the mutual information will provide the most information. (See Chapter 7 of Almond et al, 2015).

The function `VarianceOfReal(`*target*`,` *nodelist*`)` is related, but in this case `target` must either be continuous (`is.continuous(`*target*`)` is true) or have numeric values assigned to each level using

NodeLevels(*target*). For this function, the value returned is the amount by which the variance of target is expected to be reduced if the value of the observable node in the nodelist was learned. Again, higher values indicate better information.

Value

Returns a named numeric vector with the names corresponding to the nodes in nodelist and the value the mutual information or variance reduction.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: MutualInfo_bn(), VarianceOfReal_bn()

Almond, R. G., Mislevy, R. J., Steinberg, L. S., Yan, D. & Williamson, D. M. (2015) *Bayesian Networks in Educational Assessment*. Springer.

See Also

is.continuous(), NodeExpectedValue(), NodeLevels(),

Examples

```
sess <- NeticaSession()
startSession(sess)

irt5 <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","IRT5.dne"), session=sess)

irt5.theta <- NetworkFindNode(irt5,"Theta")
irt5.x <- NetworkFindNode(irt5,paste("Item",1:5,sep="_"))

CompileNetwork(irt5)

MutualInfo(irt5.theta,irt5.x)

VarianceOfReal(irt5.theta,irt5.x)

DeleteNetwork(irt5)
stopSession(sess)
```

NeticaBN *An object referencing a Bayesian network in Netica.*

Description

This file is now obsolete: See NeticaBN for the new class description.

This object is returned by various RNetica functions which create or find network objects, and contain handles to the Bayesian network. A NeticaBN object represents an active network. The function is.active() tests whether the network is still loaded into Netica's memory.

Usage

```
is.NeticaBN(x)
```

Arguments

x	The object to print or test

Details

This is an object of class NeticaBN. It consists of a name, and an invisible handle to a Netica network. The function is.active() tests the state of that handle and returns FALSE if the network is no longer in active memory (usually because of a call to DeleteNetwork()). The printed representation depends on whether or not it is active (inactive nodes print as "<Deleted Network: Name >").

For active networks, the equality test tests to see if both object point to the same object in Netica memory. Not that the name of the network is embedded in the object implementation and may get out of sync with the network, so the printed representations may be unequal even if it points to the same network. For inactive networks, the objects are compared using the cached names.

Value

For toString() a string. The function print() is usually called for its side effects.

The function is.NeticaBN() returns a logical scalar depending on whether or not its argument is a NeticaBN.

The function Ops.NeticaBN() returns a logical value depending on whether the objects are equal.

Note

[This reflects RNetica version 0.1–0.4. It is no longer current as of version 0.5.]

Internally, the NeticaBN objects are character strings with extra attributes. So as.character(net) will return the name of the network. Because of this, the default c() function will strip off the essential attributes returning them to strings. Use the cc() function instead to avoid this problem.

Note that if a NeticaBN object is stored in an R object, and the network is subsequently renamed (with a call to the set method of NetworkName), the old object may persist with the wrong name. This may result in a situation where the printed names of the objects are different but net1==net2 returns true. This can be fixed with the code NetworkName(net) <- NetworkName(net).

NeticaBN objects are all rendered inactive when StopNetica() is called, therefore they do not persist across R sessions. Generally speaking, the network should be saved, using WriteNetworks() and then reloaded in the new session using ReadNetworks(). When a network is saved or loaded the "Filename" attribute is set, to provide a mechanism for storing the filename across R sessions.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIurl/index.html: GetNetUserData_bn(), SetNetUserData_bn() (these are used to maintain the back pointers to the R object).

See Also

CreateNetwork(),DeleteNetwork(),GetNamedNetworks(),NetworkName(),is.active(),NetworkAllNodes(), WriteNetworks(),GetNetworkFileName(),cc()

Examples

```
## Not run:
net1 <- CreateNetwork("aNet")
stopifnot(is.NeticaBN(net1))
stopifnot(is.active(net1))
stopifnot(net1$Name=="aNet")

net2 <- GetNamedNetworks("aNet")
stopifnot(net2$Name=="aNet")
stopifnot(net1==net2)

NetworkName(net1) <- "Unused"
stopifnot(net1==net2)

netd <- DeleteNetwork(net1)
stopifnot(!is.active(net1))
stopifnot(!is.active(net2))
stopifnot(netd$Name=="Unused")
stopifnot(netd == net1)
## Warning:  The following expression used to not be true (RNetica <0.5)
## but now is.
net1 = net2


## End(Not run)
```

NeticaBN-class *Class* "NeticaBN"

Description

This is an R side container for a Netica netork. Note that it has a container for the nodes which have been advertised from the network.

Details

A NeticaBN is an R wrapper for the internal pointer to the Netica nework. A network is said to be 'active' if it references a network object in a current Netica session. A network become inactive when it is deleted (with a call to DeleteNetwork) or when the R session is saved an restored. In the latter case, if the network was saved with a call to WriteNetworks, then calling ReadNetworks on the inactive network will reload it from the save file.

Generally, NeticaBN objects are created with calls to either CreateNetwork or ReadNetworks. Both require a reference to an active NeticaSession object. NeticaBN objects are registered with the NeticaSession object, which contains a collection of all of the networks known about by the session.

The nodes field of the NeticaBN object (net$nodes) contains a cache of all code NeticaNode objects that are contained by the network and known about by R. Nodes are registered by their Netica name so the expression *net*$nodes$*nodename* or *net*$nodes[[*nodename*]] references a node with the name *nodename* in *net*.

Note that R node objects are created when a node is created in R, but not when a network is read in using ReadNetworks. This is useful for cases where the network is large and only a few nodes will be reference in the R code. The function NetworkFindNode() will find a node by name and create the R object corresponding to the node if needed. The function NetworkAllNodes() will create R objects for all nodes in the net.

Extends

All reference classes extend and inherit methods from "envRefClass". Note that because this is a reference class unlike traditional S3 and S4 classes it can be destructively modified. Also fields (slots) are accessed using the '$' operator.

Methods

Compare signature(e1 = "NeticaBN", e2 = "NeticaBN"): Tests for equality (mainly of pointers.

is.active signature(x = "NeticaBN"): Returns true if the NeticaBN objectcurrently references an active Netica object, and returns false if it references a deleted network or a network created in a previous sesion which has not be re-activated.

print signature(x = "NeticaBN"): Creates a printed representation.

toString signature(x = "NeticaBN"): Creates a character representation.

is.element signature(el = "NeticaBN", set = "list"): Checks to see if *el* is in list of nets.

Fields

Note these should be regarded as read-only from user code.

`Name`: Object of class `character` giving the Netica name of the network. Must follow the `IDname` rules. This should not be set by user code, use `NetworkName` instead.

`PathnameName`: Object of class `character` giving the path from which the network was last read or to which it was last saved.

`Netica_bn`: Object of class `externalptr` linking to the Netica object corresponding to this network.

`Session`: Object of class `NeticaSession`: a back pointer to the `NeticaSession` object in which the network was created.

`nodes`: Object of class `environment` which contains a cache of `NeticaNode` objects belonging to this network.

Class-Based Methods

`listNodes()`: Lists all of the cached nodes. (Contrast this to `NetworkAllNodes(net)` which lists all nodes in the network.

`searchNodes(pattern)`: Lists all cached nodes matching the regular expression given in pattern. (See `objects`.)

`show()`: Gives a detailed description of the object.

`isActive()`: Returns true if the object currently points to a Netica network, and false if it does not.

`findNode(nodename)`: Searches for a cached node with name `nodename`, returns it if found or `NULL` if not.

`clearErrors(severity)`: Calls `clearErrors` on the `Session` object.

`reportErrors(maxreport, clear)`: Calls `reportErrors` on the `Session` object.

`initialize(Name, Session, ...)`: Initialization function. Should not be called by user code.

`deactivate()`: Destroys the pointer to the Netica object. Should not be called by user code.

`deactivateNodes()`: Recursively deactives all nodes contained by this network. Should not be called by user code.

Note

The `NeticaBN` class was changed into a formal R6 reference class as of version 0.5 of RNetica. Prior to that, it was an S3 class created by adding attributes to a string. That proved to be less than robust, as several R functions (notably `c()`) would strip the attributes.

Another change is the method for finding the network object from the Netica pointer inside of the C code. Now the R objects are cached inside of a `NeticaSession` object by their netica name. The R object is found by searching the cache inside of the session object.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIurl/index.html: GetNetUserData_bn(), SetNetUserData_bn() (these are used to maintain the back pointers to the R object).

See Also

NeticaBN objects are contained by NeticaSession objects and contain NeticaNode objects.

CreateNetwork(),DeleteNetwork(),GetNamedNetworks(),NetworkName(),is.active(),NetworkAllNodes(), WriteNetworks(),GetNetworkFileName(),

Examples

```
sess <- NeticaSession()
startSession(sess)

net1 <- CreateNetwork("aNet",sess)
stopifnot(is.NeticaBN(net1))
stopifnot(is.active(net1))
stopifnot(net1$Name=="aNet")

net2 <- GetNamedNetworks("aNet",sess)
stopifnot(net2$Name=="aNet")
stopifnot(net1==net2)

NetworkName(net1) <- "Unused"
stopifnot(net1==net2)

netd <- DeleteNetwork(net1)
stopifnot(!is.active(net1))
stopifnot(!is.active(net2))
stopifnot(netd$Name=="Unused")
stopifnot(netd == net1)

stopSession(sess)
```

NeticaCaseStream *Functions for manipulating Netica case streams*

Description

The CaseStream object is a wrapper around a Netica stream which is used to read/write cases—sets of findings entered into a Netica network. There are two subclasses: FileCaseStream and MemoryCaseStream. The function ReadFindings reads the findings from the stream and the function WriteFindings writes them out.

Usage

```
OpenCaseStream(oldstream)
CloseCaseStream(stream)
is.NeticaCaseStream(x)
isCaseStreamOpen(stream)
getCaseStreamPos(stream)
getCaseStreamLastId(stream)
getCaseStreamLastFreq(stream)
```

Arguments

oldstream	A previously closed CaseStream object.
stream	A CaseStream object.
x	A object to be printed or whose type is to be determined.
...	Other arguments to toString. These are ignored.

Details

A CaseStream object is an R wrapper around a Netica stream object. There are two special cases: FileCaseStream objects are streams focused on a case file, and MemoryCaseStream objects are streams focused on a hunk of memory corresponding to an R data frame object.

Although the function WriteFindings always appends a new case to the end of a file (and hence does not need to keep the stream object open between calls), the function ReadFindings will read (by default) sequentially from the cases in the stream, and hence the stream needs to be kept open between calls.

The functions CaseFileStream and CaseMemoryStream create new streams and open them. The function OpenCaseStream will reopen a previously closed stream, and will issue a warning if the stream is already open. The function CloseCaseStream closes an open case stream (and is harmless if the stream is already closed). Although RNetica tries to close open case streams when they are garbage collected, users should not count on this behavior and should close them manually. Also be aware that all case streams are automatically closed when R is closes or RNetica is unloaded. The function isCaseStreamOpen tests to see if the stream is open or closed. The function WithOpenCaseStream executes an arbitrary R expression in a context where the stream is open, and then closed afterwards.

Netica internally keeps track of the current position of the stream when it is read or written. The functions getCaseStreamPos, getCaseStreamLastId and getCaseStreamLastFreq get information about the position in the file, the user generated id number and the frequency/weight assigned to the case at the time the stream was last read or written. In particular, the function ReadFindings returns a CaseStream object, which should be queried to find the ID and Frequencies read from the stream. When ReadFindings reaches the end of the stream, the value of getCaseStreamPos(*stream*) will be NA.

Value

The functions OpenCaseStream and CloseCaseStream both return their argument, which should be a CaseStream.

The function toString.CaseStream returns a string providing information about the source and status its argument.

The functions is.NeticaCaseStream and isCaseStreamOpen both return logical values indicating whether or not the condition holds. The latter function returns NA if its argument is not a CaseStream.

The function getCaseStreamPos returns a scalar integer values giving the position of the last record read from or written to the stream. The position is an integer corresponding to the number of characters that have been read in the stream. If an attempt has been made to read past the end of the stream, this value will be NA.

The function getCaseStreamLastId is a user specified integer associated with the case last read from or written to the stream. It's value is -1 if the user did not assign ID numbers.

The function getCaseStreamLastFreq returns a numeric scalar which is the weight associated with the last case read from or written to stream. If the user did not specify frequencies when the stream was written, the value returned is -1.

The functions LearnCPTs and LearnCases update the CPTs of a Bayesian network based on the cases in the case stream.

Note

The functions ReadNetworks and WriteNetworks also use Netica streams internally. However, as it is almost certainly a mistake to keep the stream open after the network has been read or written, no CaseStream object is created.

Internally, a weak reference system is used to keep a list of Netica stream objects which need to be closed when RNetica is unloaded. Stream objects should also be forced closed when garbage collected. The weak reference system is somewhat experimental, so well designed code should manually close the streams when the program is through with them.

Stream objects are fragile, and will not survive saving and restoring an R session. However, the object retains information about itself, so that calling OpenCaseStream on the saved object, should reopen the stream. Note that any position information will be lost.

The functions LearnCPTs and LearnCases don't seem to work with MemoryCaseStreams; for now, work around by writing the data out to a file and then writing using a FileCaseStream.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: NewFileStream_ns(),NewMemoryStream_ns(), DeleteStream_ns() http://homepage.stat.uiowa.edu/~luke/R/references/weakfinex.html

See Also

See CaseStream for details about the stream object. See FileCaseStream and MemoryCaseStream for specific details about these stream types.

CaseFileDelimiter, CaseFileMissingCode, WriteFindings, ReadFindings, CaseMemoryStream,CaseFileStream, WithOpenCaseStream LearnCPTs, LearnCases

Examples

```
sess <- NeticaSession()
startSession(sess)

abc <- CreateNetwork("ABC",sess)
A <- NewDiscreteNode(abc,"A",c("A1","A2","A3","A4"))
B <- NewDiscreteNode(abc,"B",c("B1","B2","B3"))
C <- NewDiscreteNode(abc,"C",c("C1","C2"))

AddLink(A,B)
AddLink(A,C)
AddLink(B,C)

## Outputfilename
casefile <- tempfile("testcase",fileext=".cas")

filestream <- CaseFileStream(casefile,sess)
stopifnot(is.NeticaCaseStream(filestream),
          isCaseStreamOpen(filestream))

## Case 1
NodeFinding(A) <- "A1"
NodeFinding(B) <- "B1"
NodeFinding(C) <- "C1"
filestream <- WriteFindings(list(A,B,C),filestream,1001,1.0)
stopifnot(getCaseStreamLastId(filestream)==1001,
          abs(getCaseStreamLastFreq(filestream)-1.0) <.0001)
pos1 <- getCaseStreamPos(filestream)
RetractNetFindings(abc)

## Case 2
NodeFinding(A) <- "A2"
NodeFinding(B) <- "B2"
NodeFinding(C) <- "C2"
## Double weight this case
filestream <- WriteFindings(list(A,B,C),filestream,1002,2.0)
pos2 <- getCaseStreamPos(filestream)
stopifnot(pos2>pos1,getCaseStreamLastId(filestream)==1002,
          abs(getCaseStreamLastFreq(filestream)-2.0) <.0001)
RetractNetFindings(abc)

## Case 3
NodeFinding(A) <- "A3"
NodeFinding(B) <- "B3"
## C will be missing
filestream <- WriteFindings(list(A,B,C),filestream,1003,1.0)
stopifnot(getCaseStreamLastId(filestream)==1003,
          abs(getCaseStreamLastFreq(filestream)-1.0) <.0001)
RetractNetFindings(abc)

## Close it
filestream <- CloseCaseStream(filestream)
```

```
stopifnot (is.NeticaCaseStream(filestream),
           !isCaseStreamOpen(filestream))

## Reopen it
filestream <- OpenCaseStream(filestream)
stopifnot (is.NeticaCaseStream(filestream),
           isCaseStreamOpen(filestream))

##Case 1
RetractNetFindings(abc)
filestream <- ReadFindings(list(A,B,C),filestream,"FIRST")
pos1a <- getCaseStreamPos(filestream)
stopifnot(pos1a==pos1,
          getCaseStreamLastId(filestream)==1001,
          abs(getCaseStreamLastFreq(filestream)-1.0) <.0001)

##Case 2
RetractNetFindings(abc)
filestream <- ReadFindings(list(A,B,C),filestream,"NEXT")
stopifnot(getCaseStreamPos(filestream)==pos2,
          getCaseStreamLastId(filestream)==1002,
          abs(getCaseStreamLastFreq(filestream)-2.0) <.0001)


##Clean Up
CloseCaseStream(filestream)
CloseCaseStream(filestream) ## This should issue a warning but be
## harmless.
DeleteNetwork(abc)

stopSession(sess)
```

`NeticaNode` *An object referencing a node in a Netica Bayesian network.*

Description

OBSOLETE: See `NeticaNode` (class) for current (RNetica 0.5 and beyond) implementation.

This object is returned by various RNetica functions which create or find nodes in a `NeticaBN` network. A `NeticaNode` object represents a node object inside of Netica's memory. The function `is.active()` tests whether the node is still a valid reference.

Usage

```
is.NeticaNode(x)
```

Arguments

`x`	The object to print or test

Details

This information is current only for Version 0.4 and earlier of RNetica.

This is an object of class NeticaNode. It consists of a name, and an invisible handle to a Netica node. The function is.active() tests the state of that handle and returns FALSE if the node is no longer in active memory (usually because of a call to DeleteNode() or DeleteNetwork().

NeticaNodes come in two types: discrete and continuous (see is.discrete()). The two types give slightly different meanings to the NodeStates() and NodeLevels() attributes of the node. The printed representation shows whether the node is discrete, continuous or inactive (deleted).

For active nodes, the equality test tests to see if both object point to the same object in Netica memory. Note that the name of the node is embedded in the R object implementation and may get out of sync with Netica memory, so the printed representations may be unequal even if it points to the same node. For inactive nodes, the objects are compared using the cached names.

Value

For toString() a string. The function print() is usually called for its side effects.

The function is.NeticaNode() returns a logical scalar depending on whether or not its argument is a NeticaBN.

Note

The first two paragraphs are obsolete as of Version 0.5.

Internally, the NeticaNode objects are character strings with extra attributes. So as.character(node) will return the name of the node. Because of this, the default c() function will strip off the essential attributes returning them to strings. Use the cc() function instead to avoid this problem.

Note that if a NeticaNode object is stored in an R object, and the Node is subsequently renamed (with a call to the set method of NodeName), the old object may persist with the wrong name. This may result in a situation where the printed names of the objects are different but node1==node2 returns true. This can be fixed with the code NodeName(net) <- NodeName(net).

NeticaNode objects are all rendered inactive when StopNetica() is called, therefore they do not persist across R sessions. Generally speaking, the network should be saved, using WriteNetworks() and then reloaded in the new session using ReadNetworks(). The node objects should then be recreated via a call to NetworkFindNode().

Note that RNetica is lazy about creating NeticaNode objects for nodes when a network is read from a file. Probably users should avoid creating or saving NetworkNode objects unless they are going to use them frequently.

Author(s)

Russell Almond

References

http://norsys.com/onLurl/Manual/index.html: AddNodeToNodeset_bn(), RemoveNodeFromNodeset_bn(), IsNodeInNodeset_bn() GetNodeUserData_bn(), SetNodeUserData_bn() (these are used to maintain the back pointers to the R object).

See Also

NeticaBN, NetworkFindNode(), is.active(), is.discrete(), NewContinuousNode(), NewDiscreteNode(), DeleteNodes(), NodeName(), NodeStates(), NodeLevels(), cc()

Examples

```
## Not run:
nety <- CreateNetwork("yNode")

node1 <- NewContinuousNode(nety,"aNode")
stopifnot(is.NeticaNode(node1))
stopifnot(is.active(node1))
stopifnot(as.character(node1)=="aNode")

node2 <- NetworkFindNode(nety,"aNode")
stopifnot(as.character(node2)=="aNode")
stopifnot(node1==node2)

NodeName(node1) <- "Unused"
stopifnot(node1==node2)
## Warning:  The following expression is true!
as.character(node1) != as.character(node2)

noded <- DeleteNodes(node1)
stopifnot(!is.active(node1))
stopifnot(!is.active(node2))
stopifnot(as.character(noded)=="Unused")
stopifnot(noded == node1)
## Warning:  The following expression is true!
node1 != node2

DeleteNetwork(nety)


## End(Not run)
```

NeticaNode-class *Class* "NeticaNode"

Description

This object is returned by various RNetica functions which create or find nodes in a NeticaBN network. A NeticaNode object represents a node object inside of Netica's memory. The function is.active() tests whether the node is still a valid reference.

Details

This is an object of class NeticaNode. It consists of a name, and an pointer to a Netica node in the workspace. The function is.active() tests the state of that handle and returns FALSE if the node is no longer in active memory (usually because of a call to DeleteNode() or DeleteNetwork().

NeticaNodes come in two types: discrete and continuous (see is.discrete()). The two types give slightly different meanings to the NodeStates() and NodeLevels() attributes of the node. The printed representation shows whether the node is discrete, continuous or inactive (deleted).

NeticaNode objects are created at two different times. First, when the user creates a node in a network using the NewContinuousNode() or NewDiscreteNode() functions. The second is when a user first reads the network in from a file using ReadNetworks and then subsequently searches for the node using NetworkFindNode. Note that this latter means that there may be nodes in the Netica network for which no R object has yet been created. When NeticaNode objects are created, they are cached in the NeticaBN object. Cached objects can be referenced by the nodes field of the NeticaBN object (which is an R environment). Thus, the expressions *net*$nodes$*nodename* and *net*$nodes[[*nodename*]] both reference a node with the Netica name *nodename* in the network *net*. Note that both of these expressions will yeild NULL if no R object has yet been created for the node. The function NetworkAllNodes(*net*) will as a side effect create node objects for all of the nodes in *net*.

The function match (and consequently %in% does not like it when the first argument is a node. To get around this problem, wrap the node in a list. I've added a method for the function is.element which does this automatically.

Extends

All reference classes extend and inherit methods from "envRefClass". Note that because this is a reference class unlike traditional S3 and S4 classes it can be destructively modified. Also fields (slots) are accessed using the '$' operator.

Methods

[<- signature(x = "NeticaNode"): Sets conditional probabliity table for node, see Extract.NeticaNode.

[signature(x = "NeticaNode"): Gets conditional probabliity table for node, see Extract.NeticaNode.

[[signature(x = "NeticaNode"): Gets conditional probabliity table for node, see Extract.NeticaNode.

Compare signature(e1 = "NeticaNode", e2 = "ANY"): Tests two nodes for equality

is.element signature(el = "NeticaNode", set = "list"): Checks to see if *el* is in list of nodes.

print signature(x = "NeticaNode"): Makes printed representation.

toString signature(x = "NeticaNode"): Makes character representation.

Fields

Note these should be regarded as read-only from user code.

Name**:** Object of class character giving the Netica name of the node. Must follow the IDname rules. This should not be modified by user code, use NodeName instead.

Netica_Node**:** Object of class externalptr giving the address of the node in Netica's memory space.

Net**:** Object of class NeticaBN, a back reference to the network in which this node resides.

discrete**:** Object of class logical true if the node is discrete and false otherwise.

Class-Based Methods

`show()`: Prints a description of the node.

`isActive()`: Returns true if the object currently points to a Netica node, and false if it does not.

`clearErrors(severity)`: Calls `clearErrors` on the `Net$Session` object.

`reportErrors(maxreport, clear)`: Calls `reportErrors` on the `Net$Session` object.

`initialize(Name, Net, discrete, ...)`: Initialziation function. Should not be called directly by user code. Use `NewDiscreteNode` or `NewContinuousNode` instead.

`deactivate()`: Recursively deactives all nodes contained by this network. Should not be called by user code.

Note

`NeticaNode` objects are all rendered inactive when `StopNetica()` is called, therefore they do not persist across R sessions. Generally speaking, the network should be saved, using `WriteNetworks()` and then reloaded in the new session using `ReadNetworks()`. The node objects should then be recreated via a call to `NetworkFindNode()` or `NetworkAllNodes()`.

Author(s)

Russell Almond

References

`http://norsys.com/onLurl/Manual/index.html`: AddNodeToNodeset_bn(), RemoveNodeFromNodeset_bn(), IsNodeInNodeset_bn() GetNodeUserData_bn(), SetNodeUserData_bn() (these are used to maintain the back pointers to the R object).

See Also

Its container class can be found in `NeticaBN`.

The help file `Extract.NeticaNode` explains the principle methods of referencing the conditional probability table.

`NetworkFindNode()`, `is.active()`, `is.discrete()`, `NewContinuousNode()`, `NewDiscreteNode()`, `DeleteNodes()`, `NodeName()`, `NodeStates()`, `NodeLevels()`,

Examples

```
sess <- NeticaSession()
startSession(sess)

nety <- CreateNetwork("yNode",sess)

node1 <- NewContinuousNode(nety,"aNode")
stopifnot(is.NeticaNode(node1))
stopifnot(is.active(node1))
stopifnot(node1$Name=="aNode")
```

```
node2 <- NetworkFindNode(nety,"aNode")
stopifnot(node2$Name=="aNode")
stopifnot(node1==node2)

NodeName(node1) <- "Unused"
stopifnot(node1==node2)
node1$Name == node2$Name

noded <- DeleteNodes(node1)
stopifnot(!is.active(node1))
stopifnot(!is.active(node2))
stopifnot(noded$Name=="Unused")
stopifnot(noded == node1)
node1 == node2

DeleteNetwork(nety)
stopSession(sess)
```

NeticaRNG *Creates a Netica Random Number Generator*

Description

These functions create and manipulate Netica Random Number Generators (NeticaRNG. Note that the storage for NeticaRNG objects should be freed when you are done with them by calling FreeNeticaRNG(rng).

Usage

```
NewNeticaRNG(seed = sample.int(.Machine$integer.max,1L), session=getDefaultSession())
FreeNeticaRNG(rng)
WithRNG(rng,expr)
is.NeticaRNG(x)
isNeticaRNGActive(rng)
```

Arguments

seed	An unsigned integer to use as a seed for the random number generator.
session	An object of type NeticaSession which defines the reference to the Netica workspace.
rng	A NeticaRNG object
x	An arbitrary object
expr	An expression to be executed.

Details

Netica supports random number generator objects which serve can be used to generate random cases (`GenerateRandomCase()`). In either case explicitly creating a random number generator is optional. If this is not done, the default random number generator is uses, which is slightly slower because it needs to be threadsafe. As RNetica probably adds more overhead than the non-threadsafe RNG, the primary use for creating a NeticaRNG is to produce a reproducable sequence of random cases.

Creating a random number generator with *rng* `<-` `NewNeticaRNG(`*seed*`)` generates an object in Netica space. The memory for that object should be freed when that is complete. The expression `FreeNeticaRNG(`*rng*`)` frees this object. The function `WithRNG` can be used to execute code in a context where the RNG will be freed after after completion or in the case of early termination due to an error.

When the random number generator is freed, or if the R session or Netica session is terminated, the NeticaRNG object will become inactive. The function `isNeticaRNGActive(`*rng*`)` tests to see if the random number generator is active (the Netica version still exists).

Value

The value of `NewNetiaRNG(seed)` is an active `NeticaRNG` object.

The value of `FreeRNG(rng)` is its argument which is now inactive.

The value of `WithRNG(rng,expr)` is the result of evaluating `expr`.

The values of `is.NeticaRNG(x)` and `isNeticaRNGActive(rng)` are logical scalars.

Note

There are two other uses of newly created Netica RNG objects in the Netica Manual which are not currently supported by RNetica. One is to simply generate uniform random numbers which seems superfluous given R's richer random number generation facilities. The second it to associate the RNG with a network. According to the manual, such RNGs no longer need to be deleted when you are done with them. This seems like it could lead to a situation where a single RNG was associated with two networks and then the RNG was deleted when the first network was deleted. Therefore the function `NetworkSetRNG()` always creates a new RNG.

Internally, a weak reference system is used to keep a list of Netica RNG objects which need to be closed when RNetica is unloaded. RNG objects should also be forced closed when garbage collected. The weak reference system is somewhat experimental, so well designed code should manually close the RNG when the program is through with it.

Author(s)

Russell Almond

References

`http://norsys.com/onLineAPIManual/index.html`: NewRandomGenerator_ns(), DeleteRandomGen_ns() `http://homepage.stat.uiowa.edu/~luke/R/references/weakfinex.html`

See Also

An object of NeticaRNG which is what is produced by a call to NewNeticaRNG.

NetworkSetRNG(), GenerateRandomCase()

Examples

```
sess <- NeticaSession()
startSession(sess)

rng <- NewNeticaRNG(123456789,sess)

stopifnot(is.NeticaRNG(rng),
          isNeticaRNGActive(rng))

FreeNeticaRNG(rng)
stopifnot(!isNeticaRNGActive(rng))

stopSession(sess)
```

NeticaRNG-class *Class* "NeticaRNG"

Description

This is an object containing a reference to a Netica Random Number Generator. Note that the storage for NeticaRNG objects should be freed when you are done with them by calling FreeNeticaRNG(*rng*).

Details

Netica supports random number generator objects which serve can be used to generate random cases (GenerateRandomCase()). In either case explicitly creating a random number generator is optional. If this is not done, the default random number generator is uses, which is slightly slower because it needs to be threadsafe. As RNetica probably adds more overhead than the non-threadsafe RNG, the primary use for creating a NeticaRNG is to produce a reproducable sequence of random cases.

Creating a random number generator with *rng* <- NewNeticaRNG(*seed*) generates an object in Netica space. The memory for that object should be freed when that is complete. The expression FreeNeticaRNG(*rng*) frees this object. The function WithRNG can be used to execute code in a context where the RNG will be freed after after completion or in the case of early termination due to an error.

When the random number generator is freed, or if the R session or Netica session is terminated, the NeticaRNG object will become inactive. The function isNeticaRNGActive(*rng*) tests to see if the random number generator is active (the Netica version still exists).

Extends

All reference classes extend and inherit methods from "envRefClass". Note that because this is a reference class unlike traditional S3 and S4 classes it can be destructively modified. Also fields (slots) are accessed using the '$' operator.

Methods

print `signature(x = "NeticaRNG")`: Produces a printer representation of the RNG object.

toString `signature(x = "NeticaRNG")`: Produces a string representation of the RNG object.

Fields

Note these should be regarded as read-only from user code.

`Name`: Object of class `character` giving a name to the RNG. These are autogenerated with a number.

`Session`: Object of class `NeticaSession` : a back pointer to the `NeticaSession` object in which the network was created.

`Netica_RNG`: Object of class `externalptr` which hold the pointer to the RNG object in the Netica workspace.

`Seed`: Object of class `integer` giving the seed used to initialize the RNG.

Class-Based Methods

`show()`: Provides a printed description

`free()`: Frees the RNG in Netica Memory. Equivalent to `FreeNeticaRNG`.

`isActive()`: Test to see if the RNG is active (been created in Netica) or inactive (already freed).

`clearErrors(severity)`: Calls `clearErrors` on the `Session` object.

`reportErrors(maxreport, clear)`: Calls `reportErrors` on the `Session` object.

`initialize(Name, Session, Seed, ...)`: Initialization function. Should not be called by user code.

Note

There are two other uses of newly created Netica RNG objects in the Netica Manual which are not currently supported by RNetica. One is to simply generate uniform random numbers which seems superfluous given R's richer random number generation facilities. The second it to associate the RNG with a network. According to the manual, such RNGs no longer need to be deleted when you are done with them. This seems like it could lead to a situation where a single RNG was associated with two networks and then the RNG was deleted when the first network was deleted. Therefore the function `NetworkSetRNG()` always creates a new RNG.

Internally, a weak reference system is used to keep a list of Netica RNG objects which need to be closed when RNetica is unloaded. RNG objects should also be forced closed when garbage collected. The weak reference system is somewhat experimental, so well designed code should manually close the RNG when the program is through with it.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: NewRandomGenerator_ns(), DeleteRandomGen_ns() http://homepage.stat.uiowa.edu/~luke/R/references/weakfinex.html

See Also

An object of NeticaRNG which is produced by a call to NewNeticaRNG.

NetworkSetRNG(), GenerateRandomCase()

Examples

```
sess <- NeticaSession()
startSession(sess)

rng <- NewNeticaRNG(123456789,sess)

stopifnot(is.NeticaRNG(rng),
          isNeticaRNGActive(rng))

FreeNeticaRNG(rng)
stopifnot(!isNeticaRNGActive(rng))

stopSession(sess)
```

NeticaSession *Creates a connection between R and Netica*

Description

This function creates a NeticaSession object which encapsulates the link between R and Netica. It also contains a collection of the networks associated with this Netica session.

Usage

```
NeticaSession(...,LicenseKey=character(),
              SessionName=paste("RNetica Session", date()),
              Checking=character(), maxmem=integer())
getDefaultSession()
```

Arguments

...	Possible to pass other fields initializers for subclasses. Base class uses fields below.
LicenseKey	If supplied, this should be a character scalar providing the license key purchased from Norsys http://www.norsys.com/. If left as default, RNetica will run in a limited mode.
SessionName	A character vector giving an identifier for the session. Only used in printing.
Checking	Object of class character one of the keywords: "NO_CHECK", "QUICK_CHECK", "REGULAR_CHECK", "COMPLETE_CHECK", or "QUERY_CHECK", which controls how rigorous Netica is about checking errors. A value of character() uses the Netica default which is "REGULAR_CHECK".
maxmem	Object of class numeric containing an integer indicating the maximum amount of memory to be used by the Netica shared library in bytes. If supplied, this should be at least 200,000.

Details

Starting with verison 0.5 of RNetica, in order to start Netica, you must first create an object of class NeticaSession and then call startSession on that object. This object then contains a pointer to the Netica environment, and networks are created within the NeticaSession object.

Netica is commercial software. The RNetica package downloads and installs the demonstration version of Netica which is limited in its functionality (particularly in the size of the networks it handles). Unlocking the full version of Netica requires a license key which can be purchased from Norsys (http://www.Norsys.com/). They will send a license key which unlocks the full capabilities of the shared library. This should be given as the LicenseKey argument to the constructor. If no license key is applied, then Netica will run in a limited mode which limits the number of networks and nodes in the networks. This is sufficient to run the test cases, and explore the capabilities, but for serious model building you will need to purchase a license.

The checking argument, if supplied, is used to call the Netica function ArgumentChecking_ns(). See the documentation of that function for the meaning of the codes. The default value, "REGULAR_CHECK" is appropriate for most development situations.

The maxmem argument, if supplied, is used to limit the amount of memory used by Netica. This is passed in a call to the Netica function LimitMemoryUsage_ns(). Netica will complain if this value is less than 200,000. Leaving this as NULL will not place limits on the size of Netica's memory for tables and things.

Prior to version 0.5, the Netica session pointer was managed inside of the c layer of RNetica. Thus, the session was an implicit argument to several functions. In particular, the functions CreateNetwork, GetNthNetwork, GetNamedNetworks, and ReadNetworks all now have a session argument. A session argument is also needed by some lower level functions which create Netica objects: CaseFileDelimiter, CaseFileMissingCode, CaseFileStream, CaseMemoryStream, OpenCaseStream and NewNeticaRNG. For backwards compatability, the default for the session argument is now the value of getDefaultSession().

In the previous version the session was created an the StartNetica() function was called when the RNetica namespace was attached. Thus the user did not need to worry about starting the Netica session. To be backwards compatable, the function getDefaultSession() searches for a default NeticaSession object and if necessary creates one and starts it.

The function getDefaultSession() does the following steps:

1. It first looks for a variable `DefaultNeticaSession` in the global environment. If this exists, it will be used as the session. If it does not exist, and R is running in interactive mode, then the user will be prompted to create one. If running in batch mode, or if the user does not want to create the default environment, then `getDefaultSession()` will raise an error.
2. If creating a new session, it will look for a variable called `NeticaLicenseKey` in the global environment. If that exists, it will be used as the license key when creating a new session. If not, then a new limited session will be created.
3. If a session object was found in step 1, or created in step 2, then, if necessary is it activated with a call to `startSession`.

Value

An object of class `NeticaSession`.

License

The Netica API is not free-as-in-speech software, the use of the Netica shared library makes you subject to the Netica License agreement (which can be found in the RNetica folder in your R library. If you do not agree to the terms of that license, please uninstall RNetica.

The Netica API is also not free-as-in-beer software. The demonstration version of the Netica API, however, is. In order for you to make full use of the RNetica API, you must purchase a Netica API license from Norsys (`http://norsys.com/`).

RNetica itself (the glue layers between R and Netica) is free (in both the speech and beer senses) software. Suggestions for improvements and bug fixes are welcome.

Author(s)

Russell Almond

References

`http://norsys.com/onLineAPIManual/index.html`: NewNeticaEnviron_ns(), InitNetica2_bn(), CloseNetica_bn(), LimitMemoryUsage_ns(), ArgumentChecking_ns()

See Also

See `NeticaSession` for a discription of methods that use the Netica Session object.

`startSession`, `stopSession`, `CreateNetwork`, `GetNthNetwork`, `GetNamedNetworks`, and `ReadNetworks`. It is

Examples

```
## Not run:
## Create a fully licensed session, and save it as the default
DefaultNeticaSession <- NeticaSession(LicenseKey="License Key from Norsys")

## Create a limited mode session
sess <- NeticaSession()
```

```
startSession(sess)
NeticaVersion(sess)
getDefaultSession()
stopSession(sess)

## End(Not run)
```

NeticaSession-class *Class* "NeticaSession"

Description

An R object which provides a link to the Netica session. One of these must be present and active to allow access to Netica from R. This object is also how one enables the Netica license.

Details

A Netica session is an R wrapper for the internal pointer to the Netica workspace. It is used by a number of high-level Netica functions to provide access to the workspace, notably: CreateNetwork, GetNthNetwork, GetNamedNetworks, and ReadNetworks. It is also needed by some lower level functions which create Netica objects: CaseFileDelimiter, CaseFileMissingCode, CaseFileStream, CaseMemoryStream, OpenCaseStream and NewNeticaRNG.

When initially created, the NeticaSession object is not active; that is, the connection to the internal Netica environment is not yet established. Calling the method `startSession` will start the session, making it active. The method `stopSession` will deallocate the Netica workspace. Note that it will also make any NeticaBN or NeticaNode objects it contains inactive. The function `is.active()` tests to see whether the session is currently active.

Starting with the introduction of the NeticaSession class, a license key purchased from Norsys (http://www.norsys.com/) is a field of the NeticaSession class. This field should either be a character scalar giving the complete string supplied by Norsys or a character scalar vector of length zero (the default). The licence key is passed to Netica with the call to `startSession()`; if it is a valid license key, then Netica starts in unlimited mode. If it is not a valid license key, the Netica will start in a limited mode which restricts the number of objects that can be created. All of the examples in the documentation should work in the limited mode; but people wanting to do serious work will need to purchase a license key.

As the LicenseKey is stored in the NeticaSession object, do not share dumps of the NeticaSession object with people who are not eligible to use your license.

The `nets` field of the NeticaSession object contains a collection (actually an environment) of all of the networks. They are referenced using their Netica names. In particular, the construct `session$nets$netname` will return the network named netname if it exists, or `NULL`. Similarly, the construct *session*`$nets[[`*netname*`]]` will attempt to return the network whose name is the value of `netname`. The method `session$listNets()` lists the names of all networks registered with the session. This collection is maintained by the CreateNetwork, DeleteNetwork, and ReadNetworks, so it is almost certainly an error to try and manually change it. Network objects should be renamed using NetworkName which will update the collection.

Note that if an R workspace containing a NeticaSession object is saved and restored, then the nets field will be a collection of inactive NeticaBN objects corresponding to the networks that were open when the session was saved. As these contain the pathnames where the networks were last saved, it can be used to reload the networks for a project.

It is unknown what would happen if more than one NeticaSession is active at the same time. Many possibilities are less than ideal.

Extends

All reference classes extend and inherit methods from "envRefClass". Note that because this is a reference class unlike traditional S3 and S4 classes it can be destructively modified. Also fields (slots) are accessed using the '$' operator.

Methods

is.active signature(x = "NeticaSession"): Returns true if the link to Netica is currently active and available and false if not.

startSession signature(session = "NeticaSession"): Starts the Netica session and makes it available.

stopSession signature(session = "NeticaSession"): Stops the Netica session and makes all NeticaBN and NeticaNode objects inactive.

restartSession signature(session = "NeticaSession"): Stops and restarts the session.

Fields

Note these should be regarded as read-only from user code.

LicenseKey: Object of class character giving the license key obtained from Norsys (http://www.norsys.com/. Leaving this field as character(0) will result in the limited version of Netica being used.

SessionName: Object of class character giving an identifier for the session. This is just used for printing.

NeticaHandle: Object of class externalptr giving the C memory location of the Netica API workspace. This should not be manipulated by users. If the session is inactive, this pointer will be nil.

Checking: Object of class character one of the keywords: "NO_CHECK", "QUICK_CHECK", "REGULAR_CHECK", "COMPLETE_CHECK", or "QUERY_CHECK", which controls how rigorous Netica is about checking errors. A value of character() uses the Netica default which is "REGULAR_CHECK".

maxmem: Object of class numeric containing an integer indicating the maximum amount of memory to be used by the Netica shared library in bytes. If supplied, this should be at least 200,000.

nets: Object of class environment used to store NeticaBN objects opened by this session. This should be regarded as read-only by user code.

Class-Based Methods

`neticaVersion()`: Returns the netica version associated with this session.

`show()`: Provides information about the session.

`isActive()`: Returns a logical value indicating whether Netica is currently started.

`listNets(pattern="")`: Lists the names of all of the networks registered with this session. If pattern is supplied it should be a regular expression, only nets whose name contains the regular expression will be listed.

`initialize(..., SessionName, autostart)`: Creates a new session. If `autostart` is true, then the session will be started after it is created.

`findNet(netname)`: Returns a `NeticaBN` object associated with a network, or `NULL` if no network with that name exists.

`clearErrors(severity)`: Clears errors of a given severity level or lower. Severity levels in order are: `"NOTHING_ERR"`, `"REPORT_ERR"`, `"NOTICE_ERR"`, `"WARNING_ERR"`, `"ERROR_ERR"`, `"XXX_ERR"`. The default is to report all errors.

`reportErrors(maxreport, clear)`: Reports errors. The `maxreport` value gives the maximum number of errors to report. The `clear` value (default true) asks if errors should be cleared after reporting. Note: errors are reported to standard output, not standard error.

Note

The session object is part of a fundamental redesign of the guts of the way RNetica works starting with version 0.5. There are three features of this redesign:

1. The old `NeticaBN` and `NeticaNode` classes, which were implemented as S3 classes formed by adding attributes to strings, have been replaced with R6 reference classes, which should be more robust. In particular, the `c()` command strips attributes, which tended to destroy node and net objects.
2. The session pointer used to be handled with a global variable in the RNetica source code. It is now a field in the new session object. This should allow more flexibility as well as not relying on a hidden mechanism.
3. Instead of using back-pointers from the Netica objects, Sessions contain an environment where nets are registered and nets contain an environment where nodes are registered. This should fix a problem with the backpointers pointing to the wrong location.

In the earlier design, the session object was hidden. For that reason, the function `getDefaultSession()` has been added. This looks for an object called `DefaultNeticaSession` in the gobal environment. If this object does not exist, the user will be prompted to make one the first time `getDefaultSession()` is called. This is the default for most high level functions which take a section argument; hopefully, this will provide backwards compatability.

The error reporting mechanism now also works through the session object. The `session$reportErrors()` and *session*`$clearErrors()` methods are used to maintain this system. The `NeticaBN` object contains a back pointer to its session, and delegates error reporting to the session. In particular, *net*`$Session` returns the session object. Similarly, a `NeticaNode` contains a back pointer to the `NeticaBN`, so *node*`$Net$Session` accesses the session.

Author(s)

Russell Almond

References

https://pluto.coe.fsu.edu/RNetica/

See Also

CreateNetwork, GetNthNetwork, GetNamedNetworks, and ReadNetworks. It is also needed by some lower level functions which create Netica objects: CaseFileDelimiter, CaseFileMissingCode, CaseFileStream, CaseMemoryStream, OpenCaseStream and NewNeticaRNG.

NeticaBN

Examples

```
## Create a limited mode session
## Not run:
## Create a fully licensed session, and save it as the default
DefaultNeticaSession <- NeticaSession(LicenseKey="License Key from Norsys")

## End(Not run)
sess <- NeticaSession()

startSession(sess)
NeticaVersion(sess)

myNet <- CreateNetwork("myNet",sess)

stopifnot(myNet==sess$nets$myNet)
stopifnot(myNet==sess$nets[["myNet"]])
stopifnot(myNet==sess$findNet("myNet"))
stopifnot(identical(sess$listNets(),c("myNet")))

sess$reportErrors()
sess$clearErrors()  ## Not necessary as the previous statement clears too.

stopSession(sess)

## Not run:
## Shows how to restore networks from a default session
## Existing in the workspace
for (netname in DefaultNeticaSession$listNets()) {
  net <- DefaultNeticaSession$findNet(netname)
  ReadNetworks(GetNetworkFileName(net),DefaultNeticaSession)
}

## End(Not run)
```

`NeticaVersion` *Fetches the version number of Netica.*

Description

The version number of Netica is returned as both an integer and a string.

Usage

```
NeticaVersion(session=getDefaultSession())
```

Arguments

`session`	An object of type `NeticaSession` which defines the reference to the Netica workspace.

Details

This is a synnonym for *session*`$neticaVersion()` (see `NeticaSession`).

This must be called after the call to `StartNetica()`.

Value

A list with two elements:

`number`	Netica version number times 100 (to make it an integer).
`message`	String defining Netica version.

Note

RNetica was developed with Netica API 5.04

Author(s)

Russell Almond

References

`http://norsys.com/onLineAPIManual/index.html`: GetNeticaVersion_bn()

See Also

`NeticaSession` `StartNetica()`

Examples

```
sess <- NeticaSession()
startSession(sess)
print(sess$neticaVersion()$message)
stopifnot(NeticaVersion(sess)$number > 409) ## Version 4.09 is a popular one.
stopSession(sess)
```

NetworkFindNode *Finds nodes in a Netica network.*

Description

The function NetworkFindNode finds a node in a NeticaBN with the given name. If no node with the specified name found, it will return NULL. The function NetworkAllNodes() returns a list of all nodes in the network.

Usage

```
NetworkFindNode(net, name)
NetworkAllNodes(net)
```

Arguments

net	The NeticaBN to search.
name	A character vector giving the name or names of the desired nodes. Names must follow the IDname protocol.

Details

Although each NeticaNode belongs to a single network, a network contains many nodes. Within a network, a node is uniquely identified by its name. However, nodes can be renamed (see NodeName()).

The function NetworkAllNodes() returns all the nodes in the network, however, the order of the nodes in the network could be different in different calls to this function.

Starting with RNetica version 0.5, NeticaBN objects keep a cache of node objects in the environment *net*$nodes. In particular, the methods *net*$findNode() will search the cache, and *net*$listNodes() will list the names of the nodes in the cache. Also, *net*$nodes$*nodename* or *net*$nodes[["*nodename*"]] will fetch the cached node (if it exists) or return NULL if it does not.

Nodes that are created in RNetica, using NewDiscreteNode or NewContinuousNode are automatically added to the cache. This is also true of other functions which return NeticaNode objects. For example, NodeParents(*node*) will add the parents of *node* to the cache if they are not there already.

A potential problem arises when the network is read from a file using ReadNetworks. This function does not automatically cache the nodes. Calling NetworkFindNode will add the nodes to the cache. Calling NetworkAllNodes will add all nodes to the cache. Calling NetworkNodesInSet can be used to pull just a subsetof nodes into the cache.

Value

The NeticaNode object or list of NeticaNode objects corresponding to names, or a list of all node objects for NetworkAllNodes(). In the latter case, the 'names' attribute of the returned list will be set to the node names.

Note

NeticaNode objects do not survive the life of a Netica session (or by implication an R session). So the safest way to "save" a NeticaNode object is to recreate it using NetworkFindNode() after the network is reloaded.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html, GetNodeNamed_bn(), GetNetNodes_bn()

See Also

NeticaBN talks more about the node cache and has other functions for manipulating it.

NetworkNodesInSet can be used to find a labeled subset of nodes.

NodeNet() retrieves the network from the node.

Examples

```
sess <- NeticaSession()
startSession(sess)

tnet <- CreateNetwork("TestNet", session=sess)
nodes <- NewDiscreteNode(tnet,c("A","B","C"))

nodeA <- NetworkFindNode(tnet,"A")
stopifnot (nodeA==nodes[[1]])

nodeBC <- NetworkFindNode(tnet,c("B","C"))
stopifnot(nodeBC[[1]]==nodes[[2]])
stopifnot(nodeBC[[2]]==nodes[[3]])

allnodes <- NetworkAllNodes(tnet)
stopifnot(length(allnodes)==3)
stopifnot(is.element(nodeA,allnodes)) ## NodeA in there somewhere.

## Not run:
## Safe way to preserve node and network objects across R sessions.
tnet <- WriteNetworks(tnet,"Tnet.neta")
q(save="yes")
# R
library(RNetica)
```

```
sess <- NeticaSession()
startSession(sess)
tnet <- ReadNetworks(tnet, session=sess)
nodes <- NetworkFindNodes(tnet,tnet$listNodes())

## End(Not run)
DeleteNetwork(tnet)
stopSession(sess)
```

`NetworkFootprint` *Returns a list of names of unconnected edges.*

Description

When a link is detached through setting a `NodeParents()` to `NULL`, or through copying a node but not its parent to a new network, this leaves a *stub node*, an unsatisfied connection. This function runs through the set of nodes in a network and lists the names of all unsatisfied connections.

Usage

```
NetworkFootprint(net)
```

Arguments

`net`	An active `NeticaBN` to be examined.

Details

Stub nodes – unsatisfied links or connections – can happen in two ways. Either by setting one of the values of `NodeParents(`*node*`)` to `NULL`, or by copying a node (using `CopyNodes()`) without copying its parents. (This can also be done in the Netica GUI by detaching the link from the parent end). This this case Netica names the `NodeInputNames()` according to the name of the old node.

The function `NetworkFootprint(`*net*`)` search all of the nodes in `net` to find stub nodes, and reports the `NodeInputNames()` of the stub nodes. This function provides a test for unsatisfied connections, and should be of assistance when joining two networks together. The function `AdjoinNetwork(`*sm* , *em*`)` joins two networks together and attempts to resolve the unsatisfied connections in `em`.

One particular application of the footprint is in the EM–SM algorithm (Almond et al, 1999; Almond and Mislevy, 1999). Here it is assumed that nodes in the footprint of an evidence model will be joined. Making a clique node `MakeCliqueNode()` ensures that joint information from the evidence model will find a good home in the system model network.

Value

A character vector giving the input names of the stub nodes in `net`. Duplicate values are removed.

Author(s)

Russell Almond

References

Almond, R. G. & Mislevy, R. J. (1999) Graphical models and computerized adaptive testing. *Applied Psychological Measurement*, 23, 223-238.

Almond, R., Herskovits, E., Mislevy, R. J., & Steinberg, L. S. (1999). Transfer of information between system and evidence models. In Artificial Intelligence and Statistics 99, Proceedings (pp. 181–186). Morgan-Kaufman

See Also

NeticaNode, NodeParents(), MakeCliqueNode(), NodeInputNames(), CopyNodes(),AdjoinNetwork()

Examples

```
sess <- NeticaSession()
startSession(sess)

## System/Student model
EMSMSystem <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","System.dne"), session=sess)

CompileNetwork(EMSMSystem)
JunctionTreeReport(EMSMSystem)

## Evidence model for Task 1a
EMTask1a <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","EMTask1a.dne"), session=sess)

NetworkFootprint(EMTask1a)
## The corresponding clique is not in system model, so force it in.
MakeCliqueNode(NetworkFindNode(EMSMSystem, NetworkFootprint(EMTask1a)))
CompileNetwork(EMSMSystem)
JunctionTreeReport(EMSMSystem)


## Evidence model for Task 2a
EMTask2a <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","EMTask2a.dne"), session=sess)
NetworkFootprint(EMTask2a)
## This is already a clique, so nothing to do.

DeleteNetwork(list(EMSMSystem,EMTask1a,EMTask2a))
stopSession(sess)
```

NetworkName *Gets or Sets the name of a Netica network.*

Description

Gets or sets the name of the network. Names must conform to the IDname rules.

Usage

```
NetworkName(net, internal=FALSE)
NetworkName(net) <- value
```

Arguments

net	A NeticaBN object which links to the network.
internal	A logical scalar. If true, the actual Netica object will be consulted, if false, a cached value in the R object will be used.
value	A character scalar containing the new name.

Details

Network names must conform to the IDname rules for Netica identifiers. Trying to set the network to a name that does not conform to the rules will produce an error, as will trying to set the network name to a name that corresponds to another different network.

The NetworkTitle() function provides another way to name a network which is not subject to the IDname restrictions.

Note that the name of the network is stored in two places: in the Name field of the NeticaBN object (*net*$Name), and internally in the Netica object. These should be the same; however, may not be. The internal field is used to force a check of the internal Netica object rather than the field in the R object.

Value

The name of the network as a character vector of length 1.

The setter method returns the modified object.

Note

This paragraph is obsolete as of RNetica version 0.5, it describes the previous versions only.

NeticaBN objects are internally implemented as character vectors giving the name of the network. If a network is renamed, then it is possible that R will hold onto an old reference that still using the old name. In this case, NetworkName(net) will give the correct name, and GetNamedNets(NetworkName(net)) will return a reference to a corrected object.

Starting with RNetica 0.5, NeticaBN objects are cached in the NeticaSession object. The setter method for NetworkName updates the cache as well.

In versions of RNetica less than 0.5, trying to set the name of a node to a name that was already used would generate a warning instead of an error. It now generates an error.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNetName_bn(), SetNetName_bn()

See Also

CreateNetwork(), NeticaBN, GetNamedNetworks(), NetworkTitle()

Examples

```
sess <- NeticaSession()
startSession(sess)

net <- CreateNetwork("funNet", session=sess)
netcached <- net
stopifnot(!is.null(sess$findNet("funNet")))

stopifnot(NetworkName(net)=="funNet")
stopifnot(NetworkName(net,internal=TRUE)=="funNet")

NetworkName(net)<-"SomethingElse"
stopifnot(net$Name=="SomethingElse")
stopifnot(is.null(sess$findNet("funNet")))
stopifnot(!is.null(sess$findNet("SomethingElse")))

stopifnot(NetworkName(net)==NetworkName(netcached))
stopifnot(NetworkName(net)==NetworkName(netcached,internal=TRUE))

net1 <- CreateNetwork("funnyNet", session=sess)
cat("Next statement should generate an error message.\n")
nn <- try(NetworkName(net1) <- "SomethingElse")
stopifnot(is(nn,"try-error"))

DeleteNetwork(net)
DeleteNetwork(net1)
stopSession(sess)
```

NetworkNodeSetColor *Returns or sets a display colour to use with a Netica node.b*

Description

Returns the display colour associated with a node set or sets the node set colour to a specified value. The colour of the node in the Netica GUI will be the colour of the highest priority node set associated with the node (see NetworkSetPriority().

Usage

```
NetworkNodeSetColor(net, setname, newcolor)
```

Arguments

net	An active NeticaBN object representing the network.
setname	A character scalar giving the name of the node set to be coloured.
newcolor	An optional scalar of any of the three kind of R colours, i.e., either a colour name (an element of colors()), a hexadecimal string of the form "#rrggbb" or "#rrggbbaa" (see rgb()), or an integer i meaning palette()[i]. Non-string values are coerced to integer. There are two special values: NA is used to indicate that the set should not have a colour associated with it. If newcolor is missing, then the existing colour is returned and not changed.

Details

Netica determines the visual style of a node by stepping through the node sets to which the node belongs in priority order (see NetworkSetPriority()) . Each node set can either have a colour set, or a flag set to indicate that the next node in order or priority should be used to determine the appearance of the node. The expression NetworkNodeSetColor(net,setname,NA) sets the flag so that membership in setname does not affect the display of the node.

The function NetworkNodeSetColor(*net*, *setname*, *colour*) sets the colour associated with the visual display of these nodes (this is only visible when the network is open in the Netica GUI). The colour can be specified in any of the usual ways that colours are specified in R (see col2rgb()). The special value NA is used to indicate that the set should be 'transparent', that is the colour of the next set in priority should be used to colour the node.

The function NetworkNodeSetColor(*net*, *setname*), with the third argument missing, returns the current node set colour instead of setting it.

Value

The old value of the node colour as hexadecimal string value of the form "#rrggbb".

Note

The colors of the built-in Netica node sets serve as the ultimate default for the display of nodes. These cannot be set or queried through this function. (This is a limitation of the Netica API).

Author(s)

Russell Almond

References

http://norsys.com/onLurl/Manual/index.html: ReorderNodesets_bn(), SetNodesetColor_bn()

See Also

NeticaNode, NodeSets(), NetworkNodeSets(), col2rgb(), rgb(), NetworkNodesInSet(), NetworkSetPriority()

Examples

```
sess <- NeticaSession()
startSession(sess)

nsnet <- CreateNetwork("NodeSetExample", session=sess)

Ability <- NewContinuousNode(nsnet,"Ability")

X1 <- NewDiscreteNode(nsnet,"Item1",c("Right","Wrong"))
EssayScore <- NewDiscreteNode(nsnet,"EssayScore",paste("level",5:0,sep="_"))

Value <- NewContinuousNode(nsnet,"Value")
NodeKind(Value) <- "Utility"
Placement <- NewDiscreteNode(nsnet,"Placement",
     c("Advanced","Regular","Remedial"))
NodeKind(Placement) <- "Decision"

NodeSets(Ability) <- "ReportingVariable"
NodeSets(X1) <- "Observable"
NodeSets(EssayScore) <- c("ReportingVariable","Observable")

## Default colour is NA (transparent)
stopifnot(
  is.na(NetworkNodeSetColor(nsnet,"Observable"))
)

## Make Reporting variables a pale blue
NetworkNodeSetColor(nsnet,"ReportingVariable",rgb(1,.4,.4))
stopifnot(
  NetworkNodeSetColor(nsnet,"ReportingVariable") == "#ff6666"
)
## Using R (nee X11) color list.
NetworkNodeSetColor(nsnet,"Observable","wheat2")
stopifnot(
  NetworkNodeSetColor(nsnet,"ReportingVariable") == "#ff6666"
)

DeleteNetwork(nsnet)
stopSession(sess)
```

`NetworkNodeSets` *Returns a list of node sets associated with a Netica network.*

Description

A node set is a character label associated with a node which provides information about its role in the models. This function returns the complete list of node sets associated with any node in the network.

Usage

```
NetworkNodeSets(net, incSystem = FALSE)
```

Arguments

net	An active NeticaBN object representing the network.
incSystem	A logical flag. If TRUE then built-in Netica node sets are returned as well as the user defined ones.

Details

Netica node sets are a collection of string labels that can be associated with various nodes in a network using the function NodeSets(). Node sets do not have any meaning to Netica: node set membership only affect the way the node is displayed (see NetworkNodeSetColor()). One purpose of node sets is to label a set of nodes that play a similar role in the model. For example, "ReportingVariable" or "Observable".

The expression NetworkNodeSets(*net*) returns the node sets that are currently associated with any node in *net*. If incSystem=TRUE, then the internal Netica system node sets will be included as well. These begin with a colon (':'). This value cannot be set directly, only indirectly through the use of NodeSets.

Value

A character vector giving the node sets used by the network.

Note

Node sets cannot be destroyed, only created. An empty node set has no effect.

Author(s)

Russell Almond

References

http://norsys.com/onLurl/Manual/index.html: GetAllNodesets_bn()

See Also

NeticaNode, NodeSets(), NetworkSetPriority(), NetworkNodesInSet(), NetworkNodeSetColor()

Examples

```
sess <- NeticaSession()
startSession(sess)
nsnet <- CreateNetwork("NodeSetExample", session=sess)

Ability <- NewContinuousNode(nsnet,"Ability")

EssayScore <- NewDiscreteNode(nsnet,"EssayScore",paste("level",5:0,sep="_"))
```

```
Value <- NewContinuousNode(nsnet,"Value")
NodeKind(Value) <- "Utility"
Placement <- NewDiscreteNode(nsnet,"Placement",
     c("Advanced","Regular","Remedial"))
NodeKind(Placement) <- "Decision"

stopifnot(
  length(NetworkNodeSets(nsnet)) == 0, ## Nothing set yet
  length(NetworkNodeSets(nsnet,TRUE)) == 22 ## Number of system states
)

NodeSets(Ability) <- "ReportingVariable"
stopifnot(
  NetworkNodeSets(nsnet) == "ReportingVariable"
)
NodeSets(EssayScore) <- "Observable"
stopifnot(
  setequal(NetworkNodeSets(nsnet),c("Observable","ReportingVariable"))
)
## Changing spelling of name adds new set, doesn't delete the old one.
NodeSets(EssayScore) <- "Observables"
stopifnot(
  setequal(NetworkNodeSets(nsnet),
           c("Observables", "Observable","ReportingVariable"))
)
## Nor does deletion
NodeSets(Ability) <- character()
stopifnot(
  setequal(NetworkNodeSets(nsnet),
           c("Observables", "Observable","ReportingVariable"))
)

DeleteNetwork(nsnet)
stopSession(sess)
```

`NetworkNodesInSet` *Returns a list of node labeled with the given node set in a Netica Network.*

Description

A node set is a character label associated with a node which provides information about its role in the models. This function returns a list of all nodes labeled with a particular node set.

Usage

```
NetworkNodesInSet(net, setname)
NetworkNodesInSet(net, setname) <- value
```

Arguments

net	An active NeticaBN object representing the network.
setname	A character scalar giving the node set to look for.
value	A list of active NeticaNode objects which should be in the node set.

Details

Netica node sets are a collection of string labels that can be associated with various nodes in a network using the function NodeSets(). Node sets do not have any meaning to Netica: node set membership only affect the way the node is displayed (see NetworkNodeSetColor()). One purpose of node sets is to label a set of nodes that play a similar role in the model. For example, "ReportingVariable" or "Observable".

The expression NetworkNodesInSet(*net*,*setname*) searches through the network for all nodes labeled with the given setname. It returns a list of such nodes.

The expression NetworkNodesInSet(*net*,*setname*) <-*value* make sure that *setname* is in the node sets of all nodes that are in *value* and that it is not in the node sets of any node that is not in *value*.

Note that it is acceptable to use the system built-ins in the getter method (but not the setter). For example searching for ":TableIncomplete" will return a collection of nodes for which the conditional probability table has not yet been set.

Value

A list of nodes which are associated with the named node set.

Author(s)

Russell Almond

References

http://norsys.com/onLurl/Manual/index.html: GetAllNodesets_bn(), IsNodeInNodeset_bn()

See Also

NeticaBN, NodeSets(), NetworkSetPriority(), NetworkNodesInSet(), NetworkNodeSetColor()

Examples

```
sess <- NeticaSession()
startSession(sess)

nsnet <- CreateNetwork("NodeSetExample", session=sess)

Ability <- NewContinuousNode(nsnet,"Ability")

XX <- NewDiscreteNode(nsnet,paste("Item",1:5,sep=""),c("Right","Wrong"))
X1 <- XX[[1]]
```

```
EssayScore <- NewDiscreteNode(nsnet,"EssayScore",paste("level",5:0,sep="_"))

Value <- NewContinuousNode(nsnet,"Value")
NodeKind(Value) <- "Utility"
Placement <- NewDiscreteNode(nsnet,"Placement",
     c("Advanced","Regular","Remedial"))
NodeKind(Placement) <- "Decision"

NodeSets(Ability) <- "ReportingVariable"
NodeSets(X1) <- "Observable"
NodeSets(EssayScore) <- c("ReportingVariable","Observable")

## setequal doesn't deal well with arbitrary objects, so
## just use the names.
nodeseteq <- function(x,y) {
  setequal(sapply(x,NodeName),sapply(y,NodeName))
}

stopifnot(
  nodeseteq(NetworkNodesInSet(nsnet,"ReportingVariable"),
            list(Ability,EssayScore)),
  nodeseteq(NetworkNodesInSet(nsnet,"Observable"),
            list(X1,EssayScore)),
  nodeseteq(NetworkNodesInSet(nsnet,"Observables"),
            list()),
  nodeseteq(NetworkNodesInSet(nsnet,":Nature"),
            c(list(Ability,EssayScore),XX)),
  nodeseteq(NetworkNodesInSet(nsnet,":Decision"),
            list(Placement)),
  nodeseteq(NetworkNodesInSet(nsnet,":Utility"),
            list(Value))
)

NetworkNodesInSet(nsnet,"TestSet") <- XX[1:3]
stopifnot(
  is.element("TestSet",NodeSets(XX[[1]])),
  is.element("TestSet",NodeSets(XX[[2]])),
  is.element("TestSet",NodeSets(XX[[3]])),
  !is.element("TestSet",NodeSets(XX[[4]])),
  !is.element("TestSet",NodeSets(XX[[5]]))
)
NetworkNodesInSet(nsnet,"TestSet") <- XX[2:4]
stopifnot(
  !is.element("TestSet",NodeSets(XX[[1]])),
  is.element("TestSet",NodeSets(XX[[2]])),
  is.element("TestSet",NodeSets(XX[[3]])),
  is.element("TestSet",NodeSets(XX[[4]])),
  !is.element("TestSet",NodeSets(XX[[5]]))
)
NetworkNodesInSet(nsnet,"TestSet") <-
  c(NetworkNodesInSet(nsnet,"TestSet"),XX[[5]])
stopifnot(
```

```
  !is.element("TestSet",NodeSets(XX[[1]])),
  is.element("TestSet",NodeSets(XX[[2]])),
  is.element("TestSet",NodeSets(XX[[3]])),
  is.element("TestSet",NodeSets(XX[[4]])),
  is.element("TestSet",NodeSets(XX[[5]]))
)

DeleteNetwork(nsnet)
stopSession(sess)
```

`NetworkSetPriority` *Changes the priority order of the node sets.*

Description

Netica sets the visual appearance (i.e., colour, see `NetworkNodeSetColor()`) of a node according to highest priority set to which the node belongs. This function changes the order of priority.

Usage

```
NetworkSetPriority(net, setlist)
```

Arguments

`net`	An active `NeticaBN` object representing the network.
`setlist`	A character vector containing a subset of the node set names. The first ones in the sequence will have the highest priority.

Details

Netica determines the visual style of a node by stepping through the node sets to which the node belongs in priority order. Each node set can either have a colour set, or a flag set to indicate that the next node in order or priority should be used to determine the appearance of the node (see `NetworkNodeSetColor()`).

This function switches the priority of the node sets names in the second argument. The node sets note mentioned in `setlist` are not affected.

Value

Returns the `net` argument invisibly.

Note

The priority of the Netica internal node sets (the ones beginning with ':') are set by Netica and cannot be changed. They all have lower priority than the user-defined node sets.

Author(s)

Russell Almond

References

http://norsys.com/onLurl/Manual/index.html: ReorderNodesets_bn(), SetNodesetColor_bn()

See Also

NeticaNode, NodeSets(), NetworkNodeSets(), NetworkNodesInSet(), NetworkNodeSetColor()

Examples

```
sess <- NeticaSession()
startSession(sess)
nsnet <- CreateNetwork("NodeSetExample", session=sess)

Ability <- NewContinuousNode(nsnet,"Ability")

EssayScore <- NewDiscreteNode(nsnet,"EssayScore",paste("level",5:0,sep="_"))

Value <- NewContinuousNode(nsnet,"Value")
NodeKind(Value) <- "Utility"
Placement <- NewDiscreteNode(nsnet,"Placement",
     c("Advanced","Regular","Remedial"))
NodeKind(Placement) <- "Decision"

NodeSets(EssayScore) <- c("ReportingVariable","Observable")

NetworkSetPriority(nsnet,c("Observable","ReportingVariable"))
## Now EssayScore should be coloured like an observable.
stopifnot( NodeSets(EssayScore) == c("Observable","ReportingVariable"))

NetworkSetPriority(nsnet,c("ReportingVariable","Observable"))
## Now EssayScore should be coloured like a Reporting Variable
stopifnot( NodeSets(EssayScore) == c("ReportingVariable","Observable"))

DeleteNetwork(nsnet)
stopSession(sess)
```

NetworkSetRNG *Sets a random number generator associates with the network.*

Description

This function creates a new random number generator using the given seed and associates it with the network.

Usage

```
NetworkSetRNG(net, seed=sample.int(.Machine$integer.max,1L))
```

Arguments

net	An active NeticaBN whose random number generator is to be set.
seed	An unsigned integer to be uses as the seed.

Details

Associating a random number generator with a Netica network has two effects. First, if the seed is constant, then subsequent calls to GenerateRandomCase will create a reproducible sequence of cases. Second, as the default random number generator Netica uses is threadsafe, the random number generation will be slightly faster.

Value

Returns the net argument.

Note

This function both creates the random number generator (see NeticaRNG) and associates it with the network argument. Following the Netica API, it should be possible to separate the two operations, but it unclear what would happen if the RNG object was then freed (either manually or by associating it with another network and then deleting that other network). It therefore seemed safer to encapsulate the RNG creation process in the C code.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: NewRandomGenerator_ns(), SetNetRandomGen_bn()

See Also

NeticaRNG, NewNeticaRNG(), GenerateRandomCase()

Examples

```
sess <- NeticaSession()
startSession(sess)

rnet <- CreateNetwork("Random", session=sess)
NetworkSetRNG(rnet, 1234469767)

DeleteNetwork(rnet)
stopSession(sess)
```

NetworkTitle *Gets the title or comments associated with a Netica network.*

Description

The title is a longer name for a network which is not subject to the Netica IDname restrictions. The comment is a free form text associated with a network.

Usage

```
NetworkTitle(net)
NetworkTitle(net) <- value
NetworkComment(net)
NetworkComment(net) <- value
```

Arguments

net	A NeticaBN object.
value	A character object giving the new title or comment.

Details

The title is meant to be a human readable alternative to the name, which is not limited to the IDname restrictions. The title also affects how the network is displayed in the Netica GUI.

The comment is any text the user chooses to attach to the network. If *value* has length greater than 1, the vector is collapsed into a long string with newlines separating the components.

Value

A character vector of length 1 providing the title or comment.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNetTitle_bn(), SetNetTitle_bn(), GetNetComments_bn(), SetNetComments_bn()

See Also

NeticaBN, NetworkName()

Examples

```
sess <- NeticaSession()
startSession(sess)
firstNet <- CreateNetwork("firstNet", session=sess)

NetworkTitle(firstNet) <- "My First Bayesian Network"
stopifnot(NetworkTitle(firstNet)=="My First Bayesian Network")

now <- date()
NetworkComment(firstNet)<-c("Network created on",now)
## Print here escapes the newline, so is harder to read
cat(NetworkComment(firstNet),"\n")
stopifnot(NetworkComment(firstNet) ==
  paste(c("Network created on",now),collapse="\n"))


DeleteNetwork(firstNet)
stopSession(sess)
```

`NetworkUndo` *Undoes (redoes) a Netica operation on a network.*

Description

Netica maintains an internal queue of reversible operations on a network. The `NetworkUndo()` rolls them back off the stack. The `NetworkRedo()`.

Usage

```
NetworkUndo(net)
NetworkRedo(net)
```

Arguments

`net`	A `NeticaBN` object on which an action took place.

Details

The details of which operations are undoable is not clearly documented in Netica. Some obvious things, like adding nodes, do not appear to work.

Value

Returns an invisible integer which is the return code from the underlying network function. Its value is not documented, other than it will be negative if the undo/redo stack is empty.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: UndoNetLastOper_bn(), RedoNetOper_bn()

See Also

NeticaBN, CreateNetwork

Examples

```
## Not run:
sess <- NeticaSession()
startSession(sess)

activeNet <- CreateNetwork("undoRedoTest", session=sess)

NewContinuousNode(activeNet,"Node1")
NewContinuousNode(activeNet,"Node2")
NewContinuousNode(activeNet,"Node3")

## These tests don't actually work, I'm not sure
## what constitutes an undoable action in Netica.
print(NetworkUndo(activeNet))
stopifnot(length(NetworkAllNodes(activeNet))==2)

print(NetworkUndo(activeNet))
stopifnot(length(NetworkAllNodes(activeNet))==1)

print(NetworkRedo(activeNet))
stopifnot(length(NetworkAllNodes(activeNet))==2)

DeleteNetwork(activeNet)
stopSession(sess)

## End(Not run)
```

NetworkUserField *Gets user definable fields associated with a Netica network.*

Description

Netica provides a mechanism for associating user defined values with a network as a series of key/value pairs. The key must be a IDname and the value can be an arbitrary string (NetworkUserField) or arbitrary object (NetworkUserObj).

Usage

```
NetworkUserField(net, fieldname)
NetworkUserField(net, fieldname) <- value
NetworkUserObj(net, fieldname)
```

```
NetworkUserObj(net, fieldname) <- value
NetworkAllUserFields(net)
```

Arguments

net	A NeticaBN object indicating the network.
fieldname	A character scalar conforming to the IDname rules.
value	For NetworkUserField, an arbitrary character vector containing the new value. Only the first element is used. For NetworkUserObj, an arbitrary object which is serialized with dputToString and then saved.

Details

Netica contains a mechanism for associating user data with networks. In the Netica documentation, they note that only strings are really supported as only strings are portable across implementations. The function NetworkUserField provides direct access for storing strings.

The function NetworkUserObj wraps the call to NetworkUserField with a call to dputToString or dgetFromString to allow the serialization of arbitrary objects.

Value

The function NetworkUserField returns a character scalar with the value stored in the field fieldname, or NA if no such field exists.

The function NetworkUserObj returns an arbitrary object created by calling dgetFromString on the value stored in the field fieldname, or NULL if no such field exists. If the string cannot be interpreted as an R object, it generates an error.

The function NetworkAllUserFields returns a character vector containing all user data stored with the network (this will be the serialized versions of objects, not the objects themselves). The names of the result are the names of the fields.

Note

In his book *Extending R* John Chambers suggest serializing R objects through XML or JSON mechanisms rather than the older dump protocol. I may move to that later, although it will likely cause backwards compatability issues.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html GetNetUserField_bn(), SetNetUserField_bn(), GetNetNthUserField_bn()

See Also

NeticaBN, NetworkComment() NodeUserField, dputToString()

Examples

```
sess <- NeticaSession()
startSession(sess)

userNet <- CreateNetwork("UserNet", session=sess)
NetworkUserField(userNet,"Author") <- "Russell Almond"
NetworkUserField(userNet,"Status") <- "In Progress"

stopifnot(NetworkUserField(userNet,"Author")=="Russell Almond")
stopifnot(NetworkUserField(userNet,"Status")=="In Progress")

fields <- NetworkAllUserFields(userNet)
stopifnot(length(fields)==2)
stopifnot(all(!is.na(match(c("Russell Almond","In Progress"),fields))))
stopifnot(all(!is.na(match(c("Author","Status"),names(fields)))))

stopifnot(is.na(NetworkUserField(userNet,"gender")))
stopifnot(is.null(NetworkUserObj(userNet,"gender")))

x <- sample(1L:10L)
NetworkUserObj(userNet,"x") <- x
x1 <- NetworkUserObj(userNet,"x")
stopifnot(all(x==x1))

DeleteNetwork(userNet)
stopSession(sess)
```

`NewDiscreteNode` *Creates (or destroys) a node in a Netica Bayesian network.*

Description

Creates a new node in the `NeticaBN` `net`. Netica Nodes can be either discrete, in which case a list of states must be given, or continuous, where states are not given. The function `DeleteNodes()` deletes a single node or a list of nodes.

Usage

```
NewDiscreteNode(net, names, states = c("Yes","No"))
NewContinuousNode(net, names)
DeleteNodes(nodes)
```

Arguments

`net`	A `NeticaBN` object point to the network where the nodes will be created.
`names`	A character vector containing the name or names of the new nodes to be created. The names must follow the `IDname` rules.

states	Either or character vector, or a list of character vectors. If it is a list, its length should be the same as the length of names. The character vectors give the names of the states for the corresponding node. The entries should all correspond to the IDname rules.
nodes	A NeticaNode or list of NeticaNode objects to be deleted. If a list of nodes, all must be from the same network.

Details

Both NewDiscreteNode() and NewContinuousNode() create new nodes in the network *net*. If *names* has length greater than 1, multiple nodes are created.

Netica currently supports two types of nodes. Discrete nodes represent nominal variables. Continuous nodes represent real variables. Continuous nodes cannot be changed to discrete nodes (or vise versa) using calls to the API [this is a different from the GUI]. However, a continuous node can be made to behave like a discrete node (or vise versa) by setting the NodeLevels() attribute.

NewDiscreteNode() additionally requires the *states* argument to set the initial set of states. (These can be changed later through calls to NodeStates()). If states is a character vector, it is used for the state names. If names has length greater than one, all nodes are created with the same set of states. The default values create a collection of binary variables. If states is a list, then each entry should be a character vector providing the list of states for the corresponding new node.

The function NewContinuousNode() creates a new continuous node. It appears as if Netica expects continuous nodes to be used in one of three ways: (1) they can be discretized using NodeLevels(), (2) they can be used as utilities, (3) they can be used as constants. The function NodeKind() can change a nature node (the default) to a constant or utility node. It appears as if Netica will not compile the network unless this is done for all nodes.

The function DeleteNode() deletes a single node or a group of nodes. If multiple nodes are to be deleted in a single call, they must all belong to the same network. Node that any NeticaNode objects that referenced the just deleted nodes will become inactive (see is.active()).

These functions will affect the cache of nodes maintained by the NeticaBN class (*net*$nodes). The creation functions will add the new nodes to the cache, and the deletion function will remove the nodes from the cache.

Value

For NewDiscreteNode() or NewContinuousNode(), this returns either a single object of class NeticaNode or a list of such objects (depending on the length of names).

For DeleteNodes() a list of inactive NeticaNode objects corresponding to the recently deleted nodes. If a node was not found, a value of NULL will be returned instead. These will be inactive.

Note

Netica nodes internally contain a pointer back to the net they are associated with (see NodeNet()), so most functions involving nodes don't require the net to be named. The node creation functions are an exception.

Most functions involving lists of nodes assume that all nodes come from the same network. Netica will generate an error if this is not the case.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: NewNode_bn(), DeleteNode_bn(), GetNodeType_bn(), SetNodeLevels_bn()

See Also

CreateNetwork(), NeticaNode, NodeName(), is.discrete(), is.active(), NodeStates(), NodeLevels(), NodeKind()

Examples

```
sess <- NeticaSession()
startSession(sess)
safetyNet <- CreateNetwork("safetyNet", session=sess)

noded1 <- NewDiscreteNode(safetyNet, "frayed") ## Yes/No
stopifnot(
 NodeName(noded1) == "frayed",
 NodeStates(noded1) == c("Yes", "No"),
 is.discrete(noded1)
 )

## Both variables should have the same set of states
noded23 <- NewDiscreteNode(safetyNet,c("TensionNS","TensionEW"),
            c("High","Med","Low"))
stopifnot(
  all(sapply(noded23,is.active)),
  all(sapply(noded23,is.discrete)),
  NodeNumStates(noded23[[1]]) == 3,
  NodeStates(noded23[[1]])==NodeStates(noded23[[2]])
)

noded45 <- NewDiscreteNode(safetyNet,c("MeshSize","RopeThickness"),
            list(c("Coarse","Fine"),c("Thick","Medium","Thin")))
stopifnot(
  all(sapply(noded45,is.active)),
  all(sapply(noded45,is.discrete)),
  NodeNumStates(noded45[[1]]) == 2,
  NodeNumStates(noded45[[2]]) == 3,
  NodeStates(noded45[[1]])!=NodeStates(noded45[[2]])
)

nodec <- NewContinuousNode(safetyNet, "Area")
stopifnot(
  is.active(nodec),
  is.continuous(nodec),
  NodeName(nodec) == "Area"
)
```

```
stopifnot(length(NetworkAllNodes(safetyNet))==6)

DeleteNodes(nodec)
stopifnot(length(NetworkAllNodes(safetyNet))==5)

DeleteNodes(noded45)
stopifnot(length(NetworkAllNodes(safetyNet))==3)

DeleteNetwork(safetyNet)
stopSession(sess)
```

`NodeBeliefs` *Returns the current marginal probability distribution associated with a node in a Netica network.*

Description

After a network is compiled, marginal probabilities are available at each of the nodes. Entering findings changes these to probabilities associated with the conditions represented by the findings. This function returns the marginal probabilities for the variable *node* conditioned on the findings.

The function `IsBeliefUpdated(`*node*`)` checks to see whether the value of findings have been propagated to *node* yet.

Usage

```
NodeBeliefs(node)
IsBeliefUpdated(node)
```

Arguments

`node`	An active `NeticaNode` representing the variable whose marginal distribution is to be determined.

Details

The function `NodeBeliefs()` is not available until the network has been compiled (`CompileNetwork()`). Asking for the marginal values before the network is compiled will throw an error.

When findings are entered, the marginal probabilities (or beliefs) associated with *node* will change. The process of propagating the findings from an evidence node to a query node is known as updating. Depending on the size and topology of the network, the updating process might take some time. To speed up operations, the *AutoUpdate* flag on the network can be cleared using `SetNetworkAutoUpdate()`.

If the *AutoUpdate* flag is not set for the network, then calling `NodeBeliefs(node)` could trigger an update cycle and hence take some time. The function `IsBeliefUpdated(code)` tests to see whether the marginal probability for *node* currently incorporates all of the findings. It returns true if it does and false if not.

Value

The function NodeBeliefs(*node*) returns a vector of probabilities of length NodeNumStates(*node*). The names of the result are the state names.

The function IsBeliefUpdated(*node*) returns TRUE if calling NodeBeliefs(*node*) will not result in probabilities being updated.

Note

I tend to avoid the term "belief" because I've spent so much time writing about Dempster–Shafer models (belief functions). Netica uses it to mean the marginal probability for a node given all of the entered evidence and conditional probability tables of all of the nodes.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeBeliefs_bn(), IsBeliefUpdated_bn()

See Also

NeticaNode, NeticaBN, NodeProbs(), NodeFinding(), JointProbability(), MostProbableConfig(), FindingsProbability() NodeExpectedValue(), NodeValue(), CalcNodeValue(),

Examples

```
sess <- NeticaSession()
startSession(sess)

irt5 <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","IRT5.dne"), session=sess)

irt5.theta <- NetworkFindNode(irt5,"Theta")
irt5.x <- NetworkFindNode(irt5,paste("Item",1:5,sep="_"))

## Not run:
NodeBeliefs(irt5.theta) ## This call will produce an errors because irt5
                        ## is not compiled

## End(Not run)
stopifnot(
  !IsBeliefUpdated(irt5.theta)
)
CompileNetwork(irt5) ## Ready to enter findings

stopifnot (
  ## irt5 is parent node, so marginal beliefs and conditional
  ## probability table should be the same.
  sum(abs(NodeBeliefs(irt5.theta) - NodeProbs(irt5.theta))) < 1e-6
)
```

```
## Marginal probability for Node 5
irt5.x5.init <- NodeBeliefs(irt5.x[[5]])

SetNetworkAutoUpdate(irt5,TRUE) ## Automatic updating
NodeFinding(irt5.x[[1]]) <- "Right"
stopifnot(
  IsBeliefUpdated(irt5.x[[5]])
)
irt5.x5.time1 <- NodeBeliefs(irt5.x[[5]])
stopifnot (
  sum(abs(irt5.x5.init-irt5.x5.time1)) > 1e-6
)

SetNetworkAutoUpdate(irt5,FALSE) ## Automatic updating
NodeFinding(irt5.x[[2]]) <- "Right"
stopifnot(
  !IsBeliefUpdated(irt5.x[[5]])
)
irt5.x5.time2 <- NodeBeliefs(irt5.x[[5]])
stopifnot (
  sum(abs(irt5.x5.time2-irt5.x5.time1)) > 1e-6,
  IsBeliefUpdated(irt5.x[[5]]) ## Now we have updated it.
)

DeleteNetwork(irt5)
stopSession(sess)
```

NodeChildren *Returns a list of the children of a node in a Netica network.*

Description

The children of a node *parent* are the nodes which are directly connected to *parent* with an edge oriented from *parent*. The function NodeChildren(*parent*) returns a list of the children of *parent*

Usage

```
NodeChildren(parent)
```

Arguments

parent	A NeticaNode whose children are to be found.

Details

The function NodeChildren(*parent*) only returns the immediate descendants of parent. A list of all descendants can be found using the function GetRelatedNodes(*parent*,"decendents").

The function link{NodeParents}() returns the opposite end of the link, however, unlike NodeParents(), NodeChildren() cannot be directly set.

Value

A list (possibly empty) of NeticaNode objects which are the children of *parent*.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeChildren_bn()

See Also

NeticaNode, AddLink(), NodeParents(), GetRelatedNodes()

Examples

```
sess <- NeticaSession()
startSession(sess)

chnet <- CreateNetwork("ChildcareCenter", session=sess)
mom <- NewContinuousNode(chnet,"Mother")
stopifnot(
  length(NodeChildren(mom))==0
)

daughters <- NewDiscreteNode(chnet,paste("Daughter",1:3,sep=""))
sapply(daughters, function(d) AddLink(mom,d))

stopifnot(
  length(NodeChildren(mom))==3,
  all(match(daughters,NodeChildren(mom),nomatch=0))>0
)

DeleteNetwork(chnet)
stopSession(sess)
```

NodeEquation *Gets or sets the equation Netica uses to calculate the CPT for a node*

Description

Netica contains a facility to calculate the conditional probability table for a node from an equation. NodeEquation() gets or sets the equation. EquationToTable() recalculates the conditional probability table associated with the node.

Usage

```
NodeEquation(node)
NodeEquation(node,autoconvert=TRUE) <- value
EquationToTable(node, numSamples = 25, sampUnc = TRUE, addExist = TRUE)
```

Arguments

node	An active NeticaNode object that references the node whose equation is to manipulated.
autoconvert	A logical value that indicates whether or not the CPT should be recalculated after the equation is set.
value	A character value giving the equation. If it has length greater than one, it is collapsed with newlines between.
numSamples	In some cases Netica uses sampling to calculate the CPT. If it does, then this is the number of sample.
sampUnc	A logical flag indicating whether or not sampling uncertainty should be added to the values. Note that setting this to FALSE could cause zero probabilities for configurations not realized in the sampling, which may or may not be a good thing.
addExist	A logical flag indicating whether or not the sampled values should be added to (TRUE) or replace (FALSE) the existing CPT. Can be used to create blended CPTs.

Details

This is a fairly minimilistic support for Netica's equation feature. Netica equations are strings, but have a very specific syntax (see the Netica manual for details). The RNetica code does no checking before passing the value to Netica.

The function `EquationToTable()` builds a conditional probability table from the equation and must be called before Netica will update the table used in calculations. The documentation for this function is somewhat unclear. In particular, it is not clear when Netica uses sampling to calculating the CPT (this should not be needed in most of the examples I've worked with).

There are two differences between the RNetica implementation and the default Netica behavior. First, equations can be fairly complex. If *value* is a character vector, RNetica will concatenate it into a single string before passing it to Netica. Second, by default RNetica automatically recalculates the table when the equation is set. This is usually the desired behavior, but can be suppressed by setting `autoconvert=FALSE`.

Constants play a special role in Netica formulas. A formula can reference the value of a constant node even if it is not a marked parent of the node whose equation is being defined. It appears as if the value of the constant must be set before the table is created.

Value

The function `NodeEquation` returns the equation as a character scalar. The function `EquationToTable` returns the `node` argument invisibly.

Note

I personally find the Netica equation syntax to be verbose and unwieldy. I have found it easier to calculate the CPTs directly in R (using functions from the CPTtools package, CPTtools-package) and then entering those CPTs into Netica. The functions are provided here mainly for completeness.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeEquation_bn(), SetNodeEquation_bn(), EquationToTable_bn()

The reference document for Netica equations: http://www.norsys.com/WebHelp/NETICA/X_Equations.htm

See Also

NodeValue(),NodeKind(), NodeProbs(), Extract.NeticaNode

Examples

```
sess <- NeticaSession()
startSession(sess)

grn <- CreateNetwork("GradedResponseTest", session=sess)

## Set up the variables in our network
skill <- NewDiscreteNode(grn,"Skill",c("High","Medium","Low"))
NodeLevels(skill) <- c(1,0,-1)

score1 <- NewDiscreteNode(grn,"Score1",
  c("FullCredit","PartialCredit","NoCredit"))

## Set up a couple of constants for use in formulae
a1 <- NewContinuousNode(grn,"A1")
NodeKind(a1) <- "Constant"
b1_1 <- NewContinuousNode(grn,"B1_1")
NodeKind(b1_1) <- "Constant"
b1_2 <- NewContinuousNode(grn,"B1_2")
NodeKind(b1_2) <- "Constant"
diffB1 <- NewContinuousNode(grn,"DiffB1")

NodeLevels(diffB1) <- seq(-4,4,.5)

NodeValue(a1) <- 1
NodeValue(b1_1) <- -1.5
NodeValue(b1_2) <- 0

## Note, this will generate an error if the values of the constants are
## not set first.
```

```
NodeEquation(diffB1) <- "DiffB1() = B1_2 - B1_1"

## I think this should return 1.5, but it return NA.  I'm not sure what
## is happening here?
CalcNodeValue(diffB1)

## This is the rather clunky format for Netica formulae.  This
## implements a graded response model.
dsformula <- c(
"p(Score1 | Skill) =",
   " (Score1==FullCredit)? 1/(1+exp(-1.7*(A1/sqrt(1)*Skill-B1_2)))  :",
   " (Score1==PartialCredit) ? 1/(1+exp(-1.7*(A1/sqrt(1)*Skill-B1_1))) -",
                             " 1/(1+exp(-1.7*(A1/sqrt(1)*Skill-B1_2))) :",
   "1 - 1/(1+exp(-1.7*(A1/sqrt(1)*Skill-B1_1)))"
)

AddLink(skill,score1)
NodeEquation(score1) <- dsformula

score1[]
## Expected value:
#   Skill Score1.FullCredit Score1.PartialCredit Score1.NoCredit
#1   High         0.8455347            0.1404016      0.01406363
#2 Medium         0.5000000            0.4275735      0.07242648
#3    Low         0.1544653            0.5461019      0.29943281

NodeValue(b1_1) <- -2
score1[]    ## Change not propagated yet

EquationToTable(score1)
score1[]  ## Now it changes


DeleteNetwork(grn)
stopSession(sess)
```

NodeExpectedUtils *Calculates expected utility for each value of a decision node*

Description

Calculates the expected utility for a decision node. That is for each state of the decision node it calculates the expected utility if that state is chosen.

Usage

```
NodeExpectedUtils(node)
```

Arguments

node	An active NeticaNode object that references the node. This should be a decision node, that is NodeKind(*node*) should equal "Decision".

Details

This solves a decision problem. In an influence diagram (decision net), one decision node is considered a predecessor if its value is known at the time when a decision is made. The compilation process for a decision net will fill in predecessor relationships when they are implied by paths through nature nodes. Decision networks are typically "solved" by working backwards in time from the last decision to the first.

The expression NodeExpectedUtils(*node*) will only return a meaningful result if either, *node* represents the first sequential decision, or all prior decisions have been made (and their values are known).

Value

This should return a named numeric vector of length NodeNumStates(*node*) with each element corresponding to one of the states of node.

Warning

This function is currently returning an internal Netica error. Do not use until I get clarification from Norsys.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeExpectedUtils_bn()

For more about decision nets: http://www.norsys.com/WebHelp/NETICA/X_Decision_Problems.htm

See Also

NodeKind(), NodeValue(), NodeExpectedValue()

Examples

```
sess <- NeticaSession()
startSession(sess)

## Read the RTI network from the library.
rti <- ReadNetworks(file.path(library(help="RNetica")$path,
                              "sampleNets","CostOfTesting.dne"), session=sess)
## The two decision nodes
Test <- NetworkFindNode(rti,"Test")
Instruction <- NetworkFindNode(rti,"Instruction")
```

```
## Network must be compiled before analysis:
CompileNetwork(rti)


## Not run:
## NETICA BUG, these currently give errors.
NodeExpectedUtils(Test)
NodeExpectedUtils(Instruction)

NodeFinding(Test) <- "Yes"

NodeExpectedUtils(Instruction)

## End(Not run)

DeleteNetwork(rti)
stopSession(sess)
```

NodeExpectedValue *Calculates expected value for a numeric node*

Description

Calculates the expected value for *node* based on the current beliefs about the nodes states. The *node* should either be continuous or a discrete node with levels assigned to the values. The standard deviation is supplied as an attribute.

Usage

```
NodeExpectedValue(node)
```

Arguments

node	An active NeticaNode object that references the node. The node should be continuous or have a numeric value associated with each level (see NodeLevels).

Value

Returns a scalar real giving the expected value for *node*. It has an attribute called "std_dev" which contains the standard deviation.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeExpectedValue_bn()

See Also

NodeBeliefs(),NodeLevels(),NodeLevels(),is.continuous() NodeValue(),CalcNodeValue(),

Examples

```
sess <- NeticaSession()
startSession(sess)

irt5 <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","IRT5.dne"), session=sess)

irt5.theta <- NetworkFindNode(irt5,"Theta")
irt5.x <- NetworkFindNode(irt5,paste("Item",1:5,sep="_"))

CompileNetwork(irt5) ## Ready to enter findings

## Prior should have mean 0, Std 1.095
stopifnot(abs(NodeExpectedValue(irt5.theta)) <.000001)
stopifnot(abs(attr(NodeExpectedValue(irt5.theta),"std_dev")-1.095445)<.00001)

NodeFinding(irt5.x[[1]]) <- "Right"
## Expected value should go up
stopifnot(NodeExpectedValue(irt5.theta)>0)

DeleteNetwork(irt5)
stopSession(sess)
```

NodeExperience *Gets or sets the amount of experience associated with a node.*

Description

In learning, if the row of the conditional probability table has a Dirichlet distribution, this sets the sum of the parameters for the row. This is the number of pseudo observations for that row of the CPT.

Usage

```
NodeExperience(node)
NodeExperience(node) <- value
```

Arguments

node	An active NeticaNode.
value	An array of pseudo counts, these should be positive values. The shape of the array should match the ParentStates(*node*).

Details

When learning the conditional probabilities associated with a conditional probability table, the most general model considers each row of the conditional probability table as an independent Dirichlet distribution. If there are k states, then the parameters of the Dirichlet distribution are $a_1, \ldots, a_k$ and the expected value is $p_1 = a_1/n, \ldots, p_k = a_k/n$, where $n = a_1 + \ldots + a_k$ is the normalization constant. An alternative way to represent the Dirichlet parameters is with the probability vector and the normalization. The *experience* is the normalization constant. Note that after observing m additional observations, the normalization constant will become $n + m$, so the experience can be thought of as a pseudo-observation count. Finally, the variance of the Dirichlet distribution decreases, as n increases, so it can also be thought of as a measure of precision.

An unconditional distribution has exactly one normalization constant. A conditional distribution has on for each row of the conditional probability, that is associated with each possible configuration of the parent variables. The value of `NodeExperience(`*node*`)` is an array with dimnames matching `ParentStates(`*node*`)`. In particular, this means that specific values of experience can be accessed by using the names of the parent states.

Value

An array whose dimnames are `ParentStates(`*node*`)`. If the node has no parents, the value is a scalar.

Note

I tend to refer to this distribution as a "hyper-Dirichlet" distribution, although Spiegelhalter and Lauritzen (1990) used that term to refer to a network in which all of the nodes were parameterized in that way.

Author(s)

Russell Almond

References

`http://norsys.com/onLineAPIManual/index.html`: SetNodeExperience_bn(), GetNodeExperience_bn()

See Also

`NeticaNode`, `NodeParents()`, `NodeProbs()`, `CPA`

Examples

```
sess <- NeticaSession()
startSession(sess)
abc <- CreateNetwork("ABC", session=sess)
A <- NewDiscreteNode(abc,"A",c("A1","A2","A3","A4"))
B <- NewDiscreteNode(abc,"B",c("B1","B2","B3"))
C <- NewDiscreteNode(abc,"C",c("C1","C2"))

AddLink(A,B)
```

```
AddLink(A,C)
AddLink(B,C)

## Parentless node, only need one value
NodeExperience(A) <- 10
stopifnot(
  abs(NodeExperience(A)-10)<.00001
)

NodeExperience(B) <- c(1,2,3,4)
stopifnot(
  length(NodeExperience(B))==4,
  all(names(NodeExperience(B))==NodeStates(A)),
  abs(NodeExperience(B)[2]-2)<.00001
)

## Set them all to the same value.
NodeExperience(C) <- 10
stopifnot(
  all(dim(NodeExperience(C))==sapply(ParentStates(C),length)),
  all(dimnames(NodeExperience(C))[[1]]==ParentStates(C)[[1]]),
  all(dimnames(NodeExperience(C))[[2]]==ParentStates(C)[[2]]),
  all(names(dimnames(NodeExperience(C)))==ParentNames(C)),
  abs(NodeExperience(C)[3,2]-10)<.00001
)
NodeExperience(C)["A3","B2"] <- 11
stopifnot(
  abs(NodeExperience(C)[3,2]-11)<.00001
)


DeleteNetwork(abc)
stopSession(sess)
```

`NodeFinding` *Returns of sets the observed value associated with a Netica node.*

Description

A finding is an observed variable in a Bayesian network. The expression `NodeFinding(`*node*`) <-` *value* indicates that the observed value of *node* should be set to *value*. The function `NodeFinding(`*node*`)` returns the current value.

Usage

```
NodeFinding(node)
NodeFinding(node) <- value
```

Arguments

node	An active NeticaNode whose value was observed or hypothesized.
value	A character or integer scalar indicating the value which was observed or hypothesized. If a character, it should be one of the values in NodeStates(*node*). If an integer it should be a value between 1 and NodeNumStates(*node*) inclusive.

Details

Setting NodeFinding(*node*) <- *value* essentially asserts that $Pr(node = value) = 1$. The value may be either expressed as a character name of one of the states, or an integer giving the index into the state table.

Note that setting NodeFinding(*node*) <- *value* clears any previous findings (including virtual findings set through NodeLikelihood() or EnterNegativeFinding()), that may have been set. The function RetractNodeFinding(*node*) will clear the current finding without setting it to a new value.

The function NodeFinding(*node*) returns the currently set finding, if there is one. It can also return one of the three special values:

1. "@NEGATIVE FINDINGS" — Negative findings have been entered using EnterNegativeFinding().
2. "@LIKELIHOOD" — Uncertain evidence which provides a likelihood of various states of the node were entered using NodeLikelihood(node)
3. "@NO FINDING" — No findings, including negative findings or likelihood findings were entered.

Value

The expression NodeFinding(*node*)<-*value* returns the modified node invisibly.

The function NodeFinding(*node*) returns a string which is either the currently set finding or one of the special values "@NO FINDING", "@LIKELIHOOD", or "@NEGATIVE FINDINGS".

Note

If SetNetworkAutoUpdate() has been set to TRUE, then this function could take some time as each finding is individually propagated. Consider wrapping multiple calls setting NodeFinding() in WithoutAutoUpdate(*net*, ...).

Unlike the Netica function EnterFinding_bn() the function "NodFinding<-" internally calls RetractFindings. So there is no need to do this manually.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeFinding_bn(), EnterFinding_bn()

See Also

NeticaBN, NodeBeliefs(), EnterNegativeFinding(), EnterFindings(), RetractNodeFinding(), NodeLikelihood(), EnterGaussianFinding(), EnterIntervalFinding(), JointProbability(), NodeValue(), MostProbableConfig(), FindingsProbability()

Examples

```
sess <- NeticaSession()
startSession(sess)
irt5 <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","IRT5.dne"), session=sess)

irt5.theta <- NetworkFindNode(irt5,"Theta")
irt5.x <- NetworkFindNode(irt5,paste("Item",1:5,sep="_"))

CompileNetwork(irt5) ## Ready to enter findings

stopifnot (
  ## irt5 is parent node, so marginal beliefs and conditional
  ## probability table should be the same.
  sum(abs(NodeBeliefs(irt5.theta) - NodeProbs(irt5.theta))) < 1e-6
)
## Marginal probability for Node 5
irt5.x5.init <- NodeBeliefs(irt5.x[[5]])

SetNetworkAutoUpdate(irt5,TRUE) ## Automatic updating
NodeFinding(irt5.x[[1]]) <- "Right"
stopifnot(
  IsBeliefUpdated(irt5.x[[5]])
)
irt5.x5.time1 <- NodeBeliefs(irt5.x[[5]])
stopifnot (
  sum(abs(irt5.x5.init-irt5.x5.time1)) > 1e-6
)

SetNetworkAutoUpdate(irt5,FALSE) ## Automatic updating
NodeFinding(irt5.x[[2]]) <- 2 ## Wrong
stopifnot(
  !IsBeliefUpdated(irt5.x[[5]]),
  NodeFinding(irt5.x[[2]]) == "Wrong"
)
irt5.x5.time2 <- NodeBeliefs(irt5.x[[5]])
stopifnot (
  sum(abs(irt5.x5.time2-irt5.x5.time1)) > 1e-6,
  IsBeliefUpdated(irt5.x[[5]]) ## Now we have updated it.
)

## Negative finding
EnterNegativeFinding(irt5.theta,c("neg1","neg2")) ## Rule out negatives.
stopifnot(
  NodeFinding(irt5.theta) == "@NEGATIVE FINDINGS"
)
```

```
## Clearing Findings
RetractNodeFinding(irt5.theta)
stopifnot(
  NodeFinding(irt5.theta) == "@NO FINDING"
)

##Virtual findings for X3.  Assume judge has said right, but judge has
##  80% accuracy rate.
NodeLikelihood(irt5.x[[3]]) <- c(.8,.2)
stopifnot(
  NodeFinding(irt5.x[[3]]) == "@LIKELIHOOD"
)

DeleteNetwork(irt5)
stopSession(sess)
```

NodeInputNames *Associates names with incoming edges on a Netica node.*

Description

The function NodeInputNames() can be used to set or retrieve names for each of the parents of *node*. This facilitates operations such as copying and reconnecting the nodes.

Usage

```
NodeInputNames(node)
NodeInputNames(node) <- value
```

Arguments

node	A NeticaNode object whose parent link names will be retrieved or set.
value	A character vector of length length(NodeParents(*node*)) giving the new names. Names must conform to the IDname convention.

Details

When a parent node is detached from a child, Netica names the link with the name of the old node. For example, suppose that the following commands were executed AddLink(A,C); AddLink(B,C). Then if the node B is detached, via NodeParents(C)[2]<-list(NULL), Netica will replace B with a stub node, and name the link "B". The command NodeParents(C)$B <- D would then attach the node D where the old node was attached.

Rather than relying on the automatic naming scheme, the node names can be directly set using NodeInputNames(*node*)<-*newvals*. Netica will not rename a detached link if there already exists a name for that link. Explicitly naming the links rather than relying on Netica's naming scheme is

probably good practice. If node input names are set, then they will be used names for the return value of NodeParents()

The getter form NodeInputNames() returns the currently set names of the input links. If an input link whose name has not been set either directly or via inserting a NULL in NodeParents() has a name of "".

Value

The function NodeInputNames() returns a character vector of the same length as GetNodeParents() giving the current names of the links. If a link has not yet been named, the corresponding entry of the vector will be the empty string.

The setter function returns the node object invisibly.

Note

To detach a parent, you must use list(NULL) on the left hand side of NodeParents(node)[i] <- list(NULL) and not NULL.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeInputNames_bn(), SetNodeInputNames_bn(), SwitchNodeParent_bn()

See Also

NeticaNode, AddLink(), NodeParents()

Examples

```
sess <- NeticaSession()
startSession(sess)
abnet <- CreateNetwork("AB", session=sess)

anodes <- NewDiscreteNode(abnet, paste("A",1:3,sep=""))
B <- NewDiscreteNode(abnet,"B")

NodeParents(B) <- anodes
stopifnot(
  all(NodeInputNames(B)=="")
)

NodeParents(B)[2] <- list(NULL)
stopifnot(
  NodeInputNames(B)==c("","A2","")
)

## Now can use A2 as name
```

```
D <- NewDiscreteNode(abnet,"D")
NodeParents(B)$A2 <- D
## But name doesn't change
stopifnot(
  NodeInputNames(B)==c("","A2","")
)

##Name the inputs
NodeInputNames(B) <- paste("Input",1:3,sep="")
stopifnot(
  names(NodeParents(B))[2]=="Input2"
)

## Now detaching nodes doesn't change input names.
NodeParents(B)[1] <- list(NULL)
stopifnot(
  NodeKind(NodeParents(B)[[1]])=="Stub",
  NodeInputNames(B)[1]=="Input1"
)

DeleteNetwork(abnet)
stopSession(sess)
```

NodeKind *Gets or changes the kind of a node in a Netica network.*

Description

Netica supports nodes of four different kinds: "Nature", "Decision", "Utility", and "Constant". A fifth kind, "Stub" is used for a reference to a node when an edge has been detached from a node. The function NodeKind() returns the current kind.

Usage

```
NodeKind(node)
NodeKind(node) <- value
```

Arguments

node	A NeticaNode object whose kind is to be determined or manipulated.
value	A character string with one of the values: "Nature", "Decision", "Utility", or "Constant". Actually, only the first letter is matched, so this could be one of N, D, U or C.

Details

A "Nature" node (the default when the node is created) is a random variable whose value can be predicted using the network. Pure Bayesian networks use only "Nature" nodes.

A "Decision" node is one whose value will be chosen by some decision maker. A "Utility" node is one whose value the decision maker is trying to optimize. A influence diagram contains decision nodes and utilities in addition to nature nodes. The goal is implicitly to find a setting of the decision nodes that maximizes the expected utility.

A "Constant" node is a parameter used for building a conditional probability table. Its value is nominally fixed, but it can be changed to perform sensitivity analysis.

A "Stub" is a reference to a node created by removing a parent node from another node without changing the table. It is assumed that a real node will later be attached in that location. This kind can only be set internally to Netica; the expression NodeKind(node) <- "Stub" will generate an error.

Value

A character vector of length one containing one of the values: "Nature", "Decision", "Utility", "Constant", or "Stub".

Note

Internal to Netica, "Stub"s are called DISCONNECTED_NODEs. I changed the name to make them start with a unique letter.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeKind_bn(), SetNodeKind_bn()

See Also

NeticaNode, is.discrete(), NodeParents()

Examples

```
sess <- NeticaSession()
startSession(sess)
knet <- CreateNetwork("kNet", session=sess)

skills <- NewContinuousNode(knet,paste("SkillAtTime",1:2,sep=""))

reward <- NewContinuousNode(knet,"RewardForSkill")
NodeKind(reward) <- "Utility"

placement <-
  NewDiscreteNode(knet,"Placement",c("Tier1","Tier2","Tier3"))
NodeKind(placement) <- "Decision"

instructionCost <- NewContinuousNode(knet,"CostOfInstruction")
NodeKind(instructionCost) <- "U"
```

```
pretest <- NewDiscreteNode(knet,"PretestDecision",c("yes","no"))
NodeKind(pretest) <- "D"

pretestScore <- NewContinuousNode(knet,"PretestScore")
NodeKind(pretestScore) <- "Nature"

pretestCost <- NewContinuousNode(knet,"PretestCost")
NodeKind(pretestCost) <- "u"

pretestR <- NewContinuousNode(knet,"PretestReliability")
NodeKind(pretestR) <- "Constant"

stopifnot(
 NodeKind(skills[[1]]) == "Nature",
 NodeKind(skills[[2]]) == "Nature",
 NodeKind(reward) == "Utility",
 NodeKind(placement) == "Decision",
 NodeKind(instructionCost) == "Utility",
 NodeKind(pretest) == "Decision",
 NodeKind(pretestScore) == "Nature",
 NodeKind(pretestCost) == "Utility",
 NodeKind(pretestR) == "Constant"
 )

## To make stub node, need links
AddLink(skills[[1]],pretestScore)
NodeInputNames(pretestScore) <- "SkillTested"
## Detach node
NodeParents(pretestScore)$SkillTested <- list(NULL)
stopifnot(
  NodeKind(NodeParents(pretestScore)$SkillTested) == "Stub"
)

DeleteNetwork(knet)
stopSession(sess)
```

`NodeLevels` *Accesses the levels associated with a Netica node.*

Description

The levels associate a numeric value with the levels of a discrete `NeticaNode`, or cut a discrete node into a number ordered categories. This function fetches or retrieves the levels for *node*. See description for more details.

Usage

```
NodeLevels(node)
NodeLevels(node) <- value
```

Arguments

node	A NeticaNode whose levels are to be accessed.
value	A numeric vector of values. For discrete nodes, *values* should have length NodeNumStates(*node*). For continuous nodes, it can be of any length (except 1) should be in either increasing or decreasing order.

Details

The behavior of the levels depends on whether the node is discrete (is.discrete(*node*)==TRUE) or continuous (is.discrete(*node*)==TRUE).

Discrete. For discrete nodes, the levels are associated with the states and provide a numeric summary of the states. In particular, if NodeLevels are set, then it is meaningful to calculate an expected value for the node. The vector returned by NodeLevels() is named with the names of the states, making the association clear. When setting the NodeLevels, it should have length equal to the number of states (NodeNumStates(*node*)).

Note that the first time the NodeLevels() are set, the entire vector must be set. After that point individual values may be changed.

Continuous. For a continuous node, the levels are used to split the continuous range into intervals (similar in spirit to the function cut()). The levels represent the endpoints of the intervals and should be in either increasing or decreasing order. The values Inf and -Inf are acceptable for the endpoints of the interval. There should be one more level than the desired number of states.

The states of a continuous node are defined by the node levels, and it is not meaningful to try to set NodeStates(), NodeStateTitles() or NodeStateComments().

Setting NodeLevels(*node*)<-NULL for a continuous node will clear the levels and the states.

Value

For discrete nodes, a numeric vector of length NodeNumStates(), with names equal to the state names. If levels have not be set, NAs will be returned.

For continuous nodes, a numeric vector of length NodeNumStates()+1 with no names, or character(0).

Note

The overloading of node levels is a "feature" of the Netica API. It is not great design, but it probably will be maintained for backwards compatibility.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: SetNodeLevels_bn()(), GetNodeLevels_bn(), GetNodeNumberStates_bn(), GetNodeStateName_bn(), SetNodeStateNames_bn()

See Also

NewDiscreteNode(), NeticaNode, NodeName(), is.discrete(), is.active(), NodeStateTitles(), NodeStates(), NodeStateComments(),

Examples

```
sess <- NeticaSession()
startSession(sess)
lnet <- CreateNetwork("LeveledNet", session=sess)

## Discrete Node
vnode <- NewDiscreteNode(lnet,"volt_switch",c("Off","Reverse","Forwards"))
stopifnot(
  length(NodeLevels(vnode))==3,
  names(NodeLevels(vnode)) == NodeStates(vnode),
  all(is.na(NodeLevels(vnode)))
)

## Not run:
## Don't run this until the levels for vnode have been set,
## it will generate an error.
NodeLevels(vnode)[2] <- 0

## End(Not run)

NodeLevels(vnode) <- 1:3
stopifnot(
  length(NodeLevels(vnode))==3,
  names(NodeLevels(vnode)) == NodeStates(vnode),
  NodeLevels(vnode)[2]==2
)

NodeLevels(vnode)["Reverse"] <- -2

## Continuous Node
wnode <- NewContinuousNode(lnet,"Weight")
stopifnot(
 length(NodeLevels(wnode))==0,
 NodeNumStates(wnode)==0
)

NodeLevels(wnode) <- c(0, 0.1, 10, Inf)
stopifnot(
 length(NodeStates(wnode))==3,
 NodeNumStates(wnode)==3
)
NodeStates(wnode) <- c("Low","Medium","High")
stopifnot(
 NodeStates(wnode)[3] == "High",
 is.null(names(NodeLevels(wnode)))
)
## Change number of states
```

```
NodeLevels(wnode) <- c(0, 0.1, 10, 100, Inf)
stopifnot(
 length(NodeStates(wnode))==4,
 NodeNumStates(wnode)==4,
 all(nchar(NodeStates(wnode))==0)
)
## Clear levels
NodeLevels(wnode) <- c()
stopifnot(
 NodeNumStates(wnode)==0,
 length(NodeStates(wnode))==0
)

DeleteNetwork(lnet)
stopSession(sess)
```

NodeLikelihood *Returns or sets the virtual evidence associated with a Netica node.*

Description

The findings associated with a node can be expressed as the probability of the evidence occurring in each of the states of the node. This is the *likelihood* associated with the node. This function retrieves or sets the likelihood.

Usage

```
NodeLikelihood(node)
NodeLikelihood(node) <- value
```

Arguments

node	An active NeticaNode whose evidence is to be queried or set.
value	A numeric vector of length NodeNumStates(*node*) representing the new likelihood for the node. All values must be between zero and one and there must be at least one positive value, but the sum does not need to equal 1.

Details

This function retrieves or sets virtual evidence associated with each node. Suppose that some set of evidence e is observed. The each of the values in the likelihood represents the conditional probability $Pr(e|node == state)$. Note that the likelihood can be thought of as the message that a new node *child* which was a child of *node* with no other parents would pass to *node* if its value was set.

As the likelihood values are conditional probabilities, they do not need to add to 1, although they are still restricted to the range [0,1]. Also, at least one value must be non-zero (this represents an impossible case) or Netica will genrepate an error.

Entering findings through NodeFinding(*node*) <- *state* sets a special likelihood. In this case, the likelihood value corresponding to *state* will be one, and all others will be zero. Similarly, the expression EnterNegativeFinding(*node*, *statelist*) sets a special likelihood with 0's corresponding to the states in *statelist* and 1's elsewhere.

Setting the likelihood calls RetractNodeFinding(), clearing any previous finding, negative finding or likelihood.

Value

The function NodeLikelihood(*node*) returns a vector of likelihoods of length NodeNumStates(*node*). The names of the result are the state names.

The expression NodeLikelihood(*node*)<-*value* returns the modified node invisibly.

Warning

The documentation for the Netica function MostProbableConfig_bn() states that likelihood findings are not properly taken into account in MostProbableConfig(). Some quick tests indicate that it is doing something sensible, but more extensive testing and/or clarification is needed.

The documentation for the Netica function FindingsProbability_bn() also provides a warning about likelihood evidence. The function FindingsProbability(*net*) still gives a result, but it is the normalization constant for the network, and not necessarily a probability.

Note

If SetNetworkAutoUpdate() has been set to TRUE, then setting the likelihood could take some time as each finding is individually propagated. Consider wrapping multiple calls setting NodeLikelihood() in WithoutAutoUpdate(net, ...).

Unlike the Netica function EnterNodeLikelihood_bn() the function "NodeLikelihood<-" internally calls RetractFindings. So there is no need to do this manually.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeLikelihood_bn(), EnterNodeLikelihood_bn()

See Also

NeticaBN, NodeBeliefs(), EnterNegativeFinding(), RetractNodeFinding(), NodeFinding() JointProbability(), MostProbableConfig(), FindingsProbability()

Examples

```
sess <- NeticaSession()
startSession(sess)
irt5 <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","IRT5.dne"), session=sess)

irt5.theta <- NetworkFindNode(irt5,"Theta")
irt5.x <- NetworkFindNode(irt5,paste("Item",1:5,sep="_"))

CompileNetwork(irt5) ## Ready to enter findings

## Simple finding
NodeFinding(irt5.x[[1]])<-"Wrong"
stopifnot(
  NodeLikelihood(irt5.x[[1]]) == c(0,1)
)

## Negative finding
EnterNegativeFinding(irt5.theta,c("neg1","neg2")) ## Rule out negatives.
stopifnot(
  NodeLikelihood(irt5.x[[1]]) == c(0,1),
  NodeLikelihood(irt5.theta) == c(1,1,1,0,0),
  NodeFinding(irt5.theta) == "@NEGATIVE FINDINGS"
)

## Clearing Findings
RetractNodeFinding(irt5.theta)
stopifnot(
  NodeLikelihood(irt5.theta) == c(1,1,1,1,1)
)

##Virtual findings for X3.  Assume judge has said right, but judge has
##  80% accuracy rate.
NodeLikelihood(irt5.x[[3]]) <- c(.8,.2)
stopifnot(
  sum(abs(NodeLikelihood(irt5.x[[3]]) - c(.8,.2))) < 1e-6,
  NodeFinding(irt5.x[[3]]) == "@LIKELIHOOD"
)

## Add in virtual likelihood from a second judge
NodeLikelihood(irt5.x[[3]]) <- NodeLikelihood(irt5.x[[3]]) * c(.75,.25)
stopifnot(
  sum(abs(NodeLikelihood(irt5.x[[3]]) - c(.6,.05))) < 1e-6
)


DeleteNetwork(irt5)
stopSession(sess)
```

NodeName *Gets or set of a Netica node.*

Description

Gets or sets the name of the node. Names must conform to the IDname rules.

Usage

```
NodeName(node, internal=FALSE)
NodeName(node)<- value
```

Arguments

node	An active NeticaNode object that references the node.
internal	A logical scalar. If true, the actual Netica object will be consulted, if false, a cached value in the R object will be used.
value	An character vector of length 1 giving the new name.

Details

Node names must conform to the IDname rules for Netica identifiers. Trying to set the node to a name that does not conform to the rules will produce an error, as will trying to set the node name to a name that corresponds to a different node in the network.

On a call to the setting method, if a node of the given name already exists, a warning will be issued and the node argument will be returned unchanged.

The NodeTitle() function provides another way to name a node which is not subject to the IDname restrictions.

Note that the name of the node is stored in two places: in the Name field of the NeticaNode object (*node*$Name), and internally in the Netica object. These should be the same; however, may not be. The internal field is used to force a check of the internal Netica object rather than the field in the R object.

Value

The name of the node as a character vector of length 1.

The setter method returns the NeticaNode object.

Note

This paragraph is obsolete as of RNetica version 0.5, it describes the previous versions only.

NeticaNode objects are internally implemented as character vectors giving the name of the network. If a node is renamed, then it is possible that R will hold onto an old reference that still using the old name. In this case, NodeName(node) will give the correct name, and NetworkFindNode(net,NodeName(node)) will return a reference to a corrected object.

Starting with RNetica 0.5, NeticaNode objects are cached in the NeticaBN object. The setter method for NodeName updates the cache as well.

In versions of RNetica less than 0.5, trying to set the name of a node to a name that was already used would generate a warning instead of an error. It now generates an error.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeName_bn(), SetNodeName_bn()

See Also

NewDiscreteNode(), NeticaNode, NetworkFindNode(), NodeTitle(), NeticaBN

Examples

```
sess <- NeticaSession()
startSession(sess)
net <- CreateNetwork("funNet", session=sess)

pnode <- NewDiscreteNode(net,"play")
nodecached <- pnode

stopifnot(NodeName(pnode)=="play")
stopifnot(NodeName(pnode,internal=TRUE)=="play")
stopifnot(net$findNode("play")==pnode)
stopifnot(net$nodes$play==pnode)

NodeName(pnode)<-"work"
stopifnot(pnode$Name=="work")
stopifnot(is.null(net$findNode("play")))
stopifnot(net$nodes$work==pnode)

stopifnot(NodeName(pnode) == NodeName(nodecached))
stopifnot(NodeName(pnode) == NodeName(nodecached,internal=TRUE))

snode <- NewContinuousNode(net,"sleep")
cat("Next statement should generate an error message.\n")
nn <- try(NodeName(snode)<- "work") ## This should raise an error
stopifnot(is(nn,"try-error"))

allNodes <- NetworkAllNodes(net)
NodeName(allNodes$work) <- "effort"
stopifnot(net$nodes$effort == pnode)

DeleteNetwork(net)
stopSession(sess)
```

NodeNet *Finds which Netica network a node comes from.*

Description

Each active NeticaNode object lives inside of a NeticaBN object. This function finds the network corresponding to a node.

Usage

```
NodeNet(node, internal=FALSE)
```

Arguments

node	A NeticaNode object.
internal	A logical scalar. If true, the actual Netica object will be consulted, if false, a cached value in the R object will be used.

Details

Two nodes with the same details in different networks are not identical inside of Netica. Nodes are always constructed inside of nets, and the Net field of a node cannot be changed. (See CopyNodes for copying a node to a new network.)

Starting with RNetica version 0.5, a NeticaNode object can figure out its network in two different ways. First the field *node*$Net has the NeticaBN object associated with this node. The second is by going into the Netica node object, finding the corresponding network and then looking it up by name in the NeticaSession object. With the option internal=TRUE this is what is done to check the node.

The node must be active. If is.active(*node*) returns false, this function will return NULL. Note that the expression *node*$Net will return the (possible inactive) NeticaBN object that the node used to belong to.

The functions NetworkAllNodes() and NetworkFindNode() provide pseudo-inverses for this function.

Value

A NeticaBN object which contains node, or NULL if node is not active and the internal method was selected.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeNet_bn()

See Also

NeticaBN, NeticaNode, is.active(), NetworkAllNodes(), NetworkFindNode()

Examples

```
sess <- NeticaSession()
startSession(sess)
neta <- CreateNetwork("Net_A", session=sess)
netb <- CreateNetwork("Net_B", session=sess)

nodea <- NewContinuousNode(neta,"Node")
nodeb <- NewContinuousNode(netb,"Node")

stopifnot(NodeNet(nodea)==neta)
stopifnot(NodeNet(nodeb)==netb)

stopifnot(NodeNet(nodea)==NodeNet(nodea,internal=TRUE))

## Note
stopifnot(nodea != nodeb)
## But:
stopifnot(nodea$Name == nodeb$Name)

DeleteNodes(nodeb)
stopifnot(is.null(NodeNet(nodeb)))
stopifnot(nodeb$Net==netb)

DeleteNodes(nodea)

DeleteNetwork(list(neta,netb))
stopSession(sess)
```

NodeParents *Gets or sets the parents of a node in a Netica network.*

Description

A parent of a NeticaNode is another node which has a link (created through AddLink() from that node to *child*. This function returns the list of parents. It also allows the list of parents for the node to be set, altering the topology of the network (see details).

Usage

```
NodeParents(child)
NodeParents(child) <- value
```

Arguments

child	An active NeticaNode object whose parents are of interest.
value	A list of NeticaNode objects (or NULLs) which will become the new parents. Order of the nodes is important. See details.

Details

At its most basic level, NodeParents() reports on the topology of a network. Suppose we add the links A1 --> B, A2 --> B, and A3 --> B to the network. Then NodeParents(B) should return list(A1, A2, A3). The order of the inputs is important, because that this determines the order of the dimensions in the conditional probability table (NodeProbs()).

The parent list can be set. This can accomplishes a number of different goals: it can replace a parent variable, it can add additional parents, it can remove extra parents, and it can reorder parents. Changing the parents alters the topology of the network. Note that Netica networks must always be acyclic directed graphs. In particular, if is.NodeRelated(*child*,"decendent",*parent*) returns true for any prospective parent, Netica will generate an error (new parents must node be descendants of the child as that would produce a cycle).

Setting an element of the parent list to list(NULL) has special semantics. In this case, the parent node becomes a special *stub node* (or DISCONNECTED_TYPE, see NodeKind()). This creates a Bayesian network fragment which can later be connected to another Bayesian network (using SetParents() with the new parent.

The function NodeInputNames(*child*), returns a list of names for the parent variables. Naming the parent variables facilitates disconnecting the node and reconnecting it. Whenever a node is disconnected, the corresponding input is named after the disconnected node, unless it already has an input name.

Value

A list of NeticaNode objects representing the parents in the order that they will be used to establish dimensions for the conditional probability table. If NodeInputNames(*child*) has been set, the names of the result will be the input names.

The setting variant returns the modified *child* object.

Note

Much of the checking for this function is done internally in the Netica API, and not in the RNetica interface layer. In particular, creating directed cycles will produce errors in Netica and not in RNetica.

This is actually an attempt to make the RNetica interface more R-like, covering the common cases of NodeParents(*child*) <- *value*. Under the hood it is using the Netica function SwitchNodeParent_bn() to produce the expected behavior.

The fact that if x is a list x[[2]]<-NULL deletes the second element rather than replacing it with NULL is a serious design flaw in R. However, it is documented in the FAQ and it is unlikely to change, so we need to workaround it. We do this by setting the element we want to delete to list(NULL). Nominally, we would do this through x[2]<-list(NULL), which is the official workaround for the design flaw. NodeParents<- will accept list(NULL) in place of NULL because nobody who isn't part of the R Core Development Team will ever remember which form they are suppose to use here.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeParents_bn(), SwitchNodeParent_bn()

See Also

NeticaNode, AddLink(), NodeChildren(), NodeKind(), NodeInputNames(), is.NodeRelated()

Examples

```
sess <- NeticaSession()
startSession(sess)
abnet <- CreateNetwork("AB", session=sess)

anodes <- NewDiscreteNode(abnet, paste("A",1:3,sep=""))
B <- NewDiscreteNode(abnet,"B")

## Should be empty list
stopifnot(length(NodeParents(B))==0)

NodeParents(B) <- anodes
stopifnot(
  length(NodeParents(B))==3,
  NodeParents(B)[[2]] == anodes[[2]]
)

## Reorder nodes
NodeParents(B) <- anodes[c(2:3,1)]
stopifnot(
  length(NodeParents(B))==3,
  NodeName(NodeParents(B)[[2]])=="A3",
  all(nchar(names(NodeParents(B)))==0)
)

## Remove a node.
NodeParents(B) <- anodes[2:1]
stopifnot(
  length(NodeParents(B))==2,
  NodeName(NodeParents(B)[[2]])=="A1",
  all(nchar(names(NodeParents(B)))==0)
)

## Add a node
NodeParents(B) <- anodes[3:1]
stopifnot(
  length(NodeParents(B))==3,
  NodeName(NodeParents(B)[[3]])=="A1",
  all(nchar(names(NodeParents(B)))==0)
)
```

```
##Name the inputs
NodeInputNames(B) <- paste("Input",1:3,sep="")
stopifnot(
  names(NodeParents(B))[2]=="Input2"
)

## Detach the parent
NodeParents(B)$Input2 <- list(NULL)
stopifnot(
  length(NodeParents(B))==3,
  NodeKind(NodeParents(B)$Input2) == "Stub"
)

## Remove all parents
NodeParents(B) <- list()
stopifnot(
  length(NodeParents(B))==0
)

DeleteNetwork(abnet)
stopSession(sess)
```

NodeProbs — *Gets or sets the conditional probability table associated with a Netica node.*

Description

A complete Bayesian networks defines a conditional probability distribution for a node given its parents. If all the nodes are discrete, this comes in the form of a conditional probability table a multidimensional array whose first several dimensions follow the parent variable and whose last dimension follows the child variable.

Usage

```
NodeProbs(node)
NodeProbs(node) <- value
```

Arguments

node	An active, discrete NeticaNode whose conditional probability table is to be accessed.
value	The new conditional probability table. See details for the expected dimensions.

Details

Let node be the node of interest and parent*1*, parent*2*, ..., parent*p*, where p is the number of parents. Let *pdim* = sapply(NodeParents(*node*), NodeNumStates) be a vector with the number of states for each parent. A parent configuration is defined by assigning each of the parent values to one of its possible states. Each parent configuration defines a (conditional) probability distribution over the possible states of *node*.

The result of NodeProbs(*node*) will be an array with dimensions c(*pdim*, NodeNumStates(*node*)). The first p dimensions will be named according to the NodeInputNames(*node*) or the NodeName(*parent*) if the input names are not set. The last dimension will be named according to the node itself. The dimnames for the resulting array will correspond to the state names.

The setter form expects an array of the same dimensions as an argument, although it does not need to have the dimnames set.

Value

A conditional probability array of class c("CPA","array"). See details.

Note

Note that the expression node[...] also accesses the partial or complete node conditional probability table. See Extract.NeticaNode.

All of this assumes that these are discrete nodes, that is is.discrete(node) will return true for both node and all of the parents. It is unknown what Netica does is this is not right.

This doc file is still pretty lame. Probably need to redo output as a CPT class.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeProbs_bn(), SetNodeProbs_bn()

See Also

Extract.NeticaNode, NeticaNode, NodeParents(), NodeInputNames(), NodeStates(), CPA, CPF, normalize()

Examples

```
sess <- NeticaSession()
startSession(sess)
abc <- CreateNetwork("ABC", session=sess)
A <- NewDiscreteNode(abc,"A",c("A1","A2","A3","A4"))
B <- NewDiscreteNode(abc,"B",c("B1","B2","B3"))
C <- NewDiscreteNode(abc,"C",c("C1","C2"))

AddLink(A,B)
AddLink(A,C)
```

```
AddLink(B,C)

NodeProbs(A)<-c(.1,.2,.3,.4)
NodeProbs(B) <- normalize(matrix(1:12,4,3))
NodeProbs(C) <- normalize(array(1:24,c(4,3,2)))

Aprobs <- NodeProbs(A)
Bprobs <- NodeProbs(B)
Cprobs <- NodeProbs(C)
stopifnot(
  is.CPA(Aprobs),
  is.CPA(Bprobs),
  is.CPA(Cprobs)
)

DeleteNetwork(abc)
stopSession(sess)
```

NodeSets *Lists or changes the node sets associated with a Netica node.*

Description

A node set is a character label associated with a node which provides information about its role in the models. This function returns or sets the labels associated with a node.

Usage

```
NodeSets(node, incSystem = FALSE)
NodeSets(node) <- value
AddNodeToSets(node,sets)
RemoveNodeFromSets(node,sets)
```

Arguments

node	An active NeticaNode object.
incSystem	A logical flag. If TRUE then built-in Netica node sets are returned as well as the user defined ones.
value	A character vector containing the names of the node sets that *node* should be associated with. These names must follow the is.IDname() rules.
sets	A character vector containing the names of the node sets that *node* should (or should not) be associated with. These names must follow the is.IDname() rules.

Details

Netica node sets are a collection of string labels that can be associated with various nodes in a network. Node sets do not have any meaning to Netica: node set membership only affect the way the node is displayed (see NetworkNodeSetColor()). One purpose of node sets is to label a set of nodes that play a similar role in the model. For example, "ReportingVariable" or "Observable".

The expression NodeSet(*node*) returns the node sets currently associated with *node*. If incSystem=TRUE, then the internal Netica system node sets will be included as well. These begin with a colon (':').

The expression NodeSet(*node*)<-*value* removes any node sets previously associated with *node* and adds node to the node sets named in *value*. The elements of *value* need not correspond to existing node sets, new node sets will be created for new values. (Warning: this implies that if the name of the node set is spelled incorrectly in one of the calls, this will create a new node set. For example, "Observable" and "Observables" would be two distinct node sets.) Setting the node set associated with a node only affects user-defined node sets, the Netica system node sets cannot be set using NodeSet.

The functions AddNodeToSets and RemoveNodeFromSets operate on the current node set in the expected way.

Value

A character vector giving the names of the node sets *node* is associated with. The setter form, AddNodeToSets and RemoveNodeFromSets return *node*.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: AddNodeToNodeset_bn(), RemoveNodeFromNodeset_bn(), IsNodeInNodeset_bn()

See Also

NeticaNode, NodeKind(), NetworkNodeSets(), NetworkSetPriority(), NetworkNodesInSet(), NetworkNodeSetColor(), is.IDname()

Examples

```
sess <- NeticaSession()
startSession(sess)

nsnet <- CreateNetwork("NodeSetExample", session=sess)

Ability <- NewContinuousNode(nsnet,"Ability")

EssayScore <- NewDiscreteNode(nsnet,"EssayScore",paste("level",5:0,sep="_"))

Value <- NewContinuousNode(nsnet,"Value")
NodeKind(Value) <- "Utility"
```

```
Placement <- NewDiscreteNode(nsnet,"Placement",
     c("Advanced","Regular","Remedial"))
NodeKind(Placement) <- "Decision"

stopifnot(
  length(NodeSets(Ability)) == 0, ## Nothing set yet
  setequal(NodeSets(Ability,TRUE),
           c(":Continuous", ":Nature", ":TableIncomplete",
             ":Parentless", ":Childless", ":Node")),
  !is.na(match(":Utility",NodeSets(Value,TRUE))),
  !is.na(match(":Decision",NodeSets(Placement,TRUE)))
)

NodeSets(Ability) <- "ReportingVariable"
stopifnot(
  NodeSets(Ability) == "ReportingVariable"
)
NodeSets(EssayScore) <- "Observable"
stopifnot(
  NodeSets(EssayScore) == "Observable"
)
## Make EssayScore a reporting variable, too
NodeSets(EssayScore) <- c("ReportingVariable",NodeSets(EssayScore))
stopifnot(
  setequal(NodeSets(EssayScore),c("Observable","ReportingVariable"))
)

## Clear out the node set
NodeSets(Ability) <- character()
stopifnot(
  length(NodeSets(Ability)) == 0
)

NodeSets(Ability) <- c("Set1","Set2")
AddNodeToSets(Ability,c("Set2","Set3"))
stopifnot(
  length(NodeSets(Ability)) == 3,
  setequal(NodeSets(Ability),c("Set1","Set2","Set3"))
)
RemoveNodeFromSets(Ability,c("Set1","Set4"))
stopifnot(
  length(NodeSets(Ability)) == 2,
  setequal(NodeSets(Ability),c("Set2","Set3"))
)


DeleteNetwork(nsnet)
stopSession(sess)
```

NodeStates *Accessor for states of a Netica node.*

Description

This function returns a list associated with a Netica node. The function NodeNumStates() returns the number of states, NodeStates returns or manipulates them.

Usage

```
NodeStates(node)
NodeNumStates(node)
NodeStates(node, resize=FALSE) <- value
```

Arguments

node	An active NeticaNode object whose states are to be accessed.
value	A character vector of length NodeNumStates(*node*) giving the names of the states. State names must conform to the IDname rules.
resize	A logical scalar. If true, the number of states of the node will be adjusted to the length of value. If false (the default), an error will be raised. Note: changing the number of states could loose information if there is a conditional probability table or values associated with a node.

Details

States behave slightly differently for discrete and continuous nodes (see is.discrete(). For discrete nodes, the random variable represented by the node can take on one of the values represented by NodeStates(*node*).

Discrete. The number of states for a discrete node is determined when the node is created (through a call to NewDiscreteNode()). By default, setting the node states will not change the number of states in the node.

The states are important when building conditional probability tables (CPTs). In particular, the state names are used to label the columns of the CPT. Thus, state names can be used to address arrays in the same way that dimnames can. In particular, the state names can be used to index the vectors returned by NodeStates(), NodeStateTitles(), NodeStateTitles(), and NodeLevels() (for discrete nodes).

Calling NodeStates(*node*,resize=TRUE) <- *value* will adjust the number of states in the node to match the length of *value*. Note that this is a somewhat dangerous operation. If there is a CPT associated with the node, Netica will adjust it to the right size using an operation which has not been documented, but seems like a sensible default. If there is a finding associated with the node, Netica may raise an error if this state is deleted (RNetica simply deletes the unneeded states from the end of the list). It is probably safe to resize the node only in early stages of development. This is why the default is to raise an error.

Continuous. States for a continuous node are determined by the NodeLevels() of the node, which describe a series of endpoints for intervals that cut the continuous space into the states. The function NodeNumStates(*node*) should return length(NodeLevels(*node*))-1 unless the levels have not been set in which case it will be zero. If NodeStates are set for a continuous node, they must have length length(NodeLevels(*node*))-1.

Value

The function NodeNumStates() returns an integer giving the number of states.

The function NodeStates() returns a character vector of length NodeNumStates(*node*) whose values and names are both set to the state names. The setter version of this function invisibly returns the *node* object.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeNumberStates_bn(), GetNodeStateName_bn(), SetNodeStateNames_bn(), GetNodeLevels_bn() SetNodeLevels_bn(), AddNodeStates_bn(), RemoveNodeState_bn()

See Also

NewDiscreteNode(), NeticaNode, NodeName(), is.discrete(), is.active(), NodeStateTitles(), NodeLevels(), NodeStateComments(),

Examples

```
sess <- NeticaSession()
startSession(sess)
anet <- CreateNetwork("Annette", session=sess)

## Discrete Nodes
node12 <- NewDiscreteNode(anet,"TwoLevelNode")
stopifnot(
  NodeNumStates(node12)==2,
  NodeStates(node12)==c("Yes","No")
)

NodeStates(node12) <- c("True","False")
stopifnot(
  NodeStates(node12)==c("True","False")
)


node13 <- NewDiscreteNode(anet,"ThreeLevelNode",c("High","Med","Low"))
stopifnot(
  NodeNumStates(node13)==3,
  NodeStates(node13)==c("High","Med","Low"),
  NodeStates(node13)[2]=="Med"
)

NodeStates(node13)[2] <- "Median"
stopifnot(
  NodeStates(node13)[2]=="Median"
)
```

```
NodeStates(node13)["Median"] <- "Medium"
stopifnot(
  NodeStates(node13)[2]=="Medium"
)

## Adjusting size

## Not run:
## Don't run this it will generate an error.
NodeStates(node12) <- c("Low","Medium","High")

## End(Not run)

## Should work if we pass resize=TRUE
NodeStates(node12,resize=TRUE) <- c("Low","Med","High")
NodeStates(node13,resize=TRUE) <- c("Low","High")
stopifnot(
  NodeNumStates(node12)==3,
  NodeStates(node12)==c("Low","Med","High"),
  NodeNumStates(node13)==2,
  NodeStates(node13)==c("Low","High")
)

## Continuous Nodes
wnode <- NewContinuousNode(anet,"Weight")

## Not run:
## Don't run this until the levels for wnode have been set,
## it will generate an error.
NodeStates(wnode) <- c("Low","Medium","High")

## End(Not run)

## First set levels of node.
NodeLevels(wnode) <- c(0, 0.1, 10, Inf)
## Then can set States.
NodeStates(wnode) <- c("Low","Medium","High")


DeleteNetwork(anet)
stopSession(sess)
```

`NodeStateTitles` *Accessors for the titles and comments associated with states of Netica nodes.*

Description

Each state of a `NeticaNode` can have a longer title or comments associated with it. These functions get or set the titles or comments.

Usage

```
NodeStateTitles(node)
NodeStateTitles(node) <- value
NodeStateComments(node)
NodeStateComments(node) <- value
```

Arguments

node	An active NeticaNode object whose state titles or comments will be accessed.
value	A character vector of length NodeNumStates(*node*) which provides the new state titles or names.

Details

The titles are meant to be a more human readable version of the state names and are not subject the the IDname restrictions. These are displayed in the Netica GUI in certain display modes. The comments are meant to be a longer free form notes.

Both titles and comments are returned as a named character vector with names corresponding to the state names. Therefore one can change a single state title or comment by accessing it either using the state number or the state name.

Value

Both NodeStateTitles() and NodeStateComments() return a character vector of length NodeNumStates(*node*) giving the titles or comments respectively. The names of this vector are NodeStates(*node*).

The setter methods return the modified NeticaNode object invisibly.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeStateTitle_bn(),SetNodeStateTitle_bn(), GetNodeStateComment_bn(),SetNodeStateComment_bn()

See Also

NeticaNode, NodeStates(), NodeLevels()

Examples

```
sess <- NeticaSession()
startSession(sess)
cnet <- CreateNetwork("CreativeNet", session=sess)

orig <- NewDiscreteNode(cnet,"Originality", c("H","M","L"))
NodeStateTitles(orig) <- c("High","Medium","Low")
NodeStateComments(orig)[1] <- "Produces solutions unlike those typically seen."
```

```
stopifnot(
  NodeStateTitles(orig) == c("High","Medium","Low"),
  grep("solutions unlike", NodeStateComments(orig))==1,
  NodeStateComments(orig)[3]==""
  )

sol <- NewDiscreteNode(cnet,"Solution",
       c("Typical","Unusual","VeryUnusual"))
stopifnot(
  all(NodeStateTitles(sol) == ""),
  all(NodeStateComments(sol) == "")
  )

NodeStateTitles(sol)["VeryUnusual"] <- "Very Unusual"
NodeStateComments(sol) <- paste("Distance from typical solution",
                      c("<1", "1--2", ">2"))
stopifnot(
  NodeStateTitles(sol)[3]=="Very Unusual",
  NodeStateComments(sol)[1] == "Distance from typical solution <1"
  )

DeleteNetwork(cnet)
stopSession(sess)
```

NodeTitle *Gets the title or Description associated with a Netica node.*

Description

The title is a longer name for a node which is not subject to the Netica IDname restrictions. The description is a free form text associated with a node.

Usage

```
NodeTitle(node)
NodeTitle(node) <- value
NodeDescription(node)
NodeDescription(node) <- value
```

Arguments

node	A NeticaNode object.
value	A character object giving the new title or description.

Details

The title is meant to be a human readable alternative to the name, which is not limited to the IDname restrictions. The title also affects how the node is displayed in the Netica GUI.

The description is any text the user chooses to attach to the node. If *value* has length greater than 1, the vector is collapsed into a long string with newlines separating the components.

Value

A character vector of length 1 providing the title or description.

Note

Node descriptions are called "Descriptions" in the Netica GUI, but "Comments" in the API.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeTitle_bn(), SetNodeTitle_bn(), GetNodeComments_bn(), SetNodeComments_bn()

See Also

NeticaNode, NodeName()

Examples

```
sess <- NeticaSession()
startSession(sess)
net2 <- CreateNetwork("secondNet", session=sess)

firstNode <- NewDiscreteNode(net2,"firstNode")

NodeTitle(firstNode) <- "My First Bayesian Network Node"
stopifnot(NodeTitle(firstNode)=="My First Bayesian Network Node")

now <- date()
NodeDescription(firstNode)<-c("Node created on",now)
stopifnot(NodeDescription(firstNode) ==
  paste(c("Node created on",now),collapse="\n"))

## Print here escapes the newline, so is harder to read
cat(NodeDescription(firstNode),"\n")

DeleteNetwork(net2)
stopSession(sess)
```

NodeUserField *Gets user definable fields associated with a Netica node.*

Description

Netica provides a mechanism for associating user defined values with a node as a series of key/value pairs. The key must be a IDname and the value can be an arbitrary string. The function NodeUserField accesses the string value associated with the key, and the function NodeUserObj accesses an R object associated with the key.

Usage

```
NodeUserField(node, fieldname)
NodeUserField(node, fieldname) <- value
NodeUserObj(node, fieldname)
NodeUserObj(node, fieldname) <- value
NodeAllUserFields(node)
```

Arguments

node	A NeticaNode object indicating the node.
fieldname	A character scalar conforming to the IDname rules.
value	For NodeUserField, an arbitrary character vector containing the new value. Only the first element is used. For NodeUserObj, an arbitrary object which is serialized with dputToString and then saved.

Details

Netica contains a mechanism for associating user data with nodes. In the Netica documentation, they note that only strings are really supported as only strings are portable across implementations.The function NodeUserField provides direct access for storing strings.

The function NodeUserObj wraps the call to NodeUserField with a call to dputToString or dgetFromString to allow the serialization of arbitrary objects.

Value

The function NodeUserField returns a character scalar with the value stored in the field *fieldname*, or NA if no such field exists.

The function NodeUserObj returns an arbitrary object created by calling dgetFromString on the value stored in the field *fieldname*, or NULL if no such field exists. If the string cannot be interpreted as an R object, it generates an error.

The function NodeAllUserFields returns a character vector containing all user data stored with the node (this will be the serialized versions of the objects, not the objects themselves). The names of the result are the names of the fields.

Note

In his book *Extending R* John Chambers suggest serializing R objects through XML or JSON mechanisms rather than the older dump protocol. I may move to that later, although it will likely cause backwards compatability issues.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html GetNodeUserField_bn(), SetNodeUserField_bn(), GetNodeNthUserField_bn()

See Also

NeticaNode, NodeDescription() NetworkUserField, dputToString()

Examples

```
sess <- NeticaSession()
startSession(sess)
usedNet <- CreateNetwork("UsedNet", session=sess)

userNode <- NewContinuousNode(usedNet, "UserNode")
NodeUserField(userNode,"Author") <- "Russell Almond"
NodeUserField(userNode,"Status") <- "In Progress"

stopifnot(NodeUserField(userNode,"Author")=="Russell Almond")
stopifnot(NodeUserField(userNode,"Status")=="In Progress")

fields <- NodeAllUserFields(userNode)
stopifnot(length(fields)==2)
stopifnot(all(!is.na(match(c("Russell Almond","In Progress"),fields))))
stopifnot(all(!is.na(match(c("Author","Status"),names(fields)))))

stopifnot(is.na(NodeUserField(userNode,"gender")))
stopifnot(is.null(NodeUserObj(userNode,"gender")))

x <- sample(1L:10L)
NodeUserObj(userNode,"x") <- x
x1 <- NodeUserObj(userNode,"x")
stopifnot(all(x==x1))


DeleteNetwork(usedNet)
stopSession(sess)
```

NodeValue *Sets the numeric value of a continuous node*

Description

This enters a numeric value (finding) for a continuous node or a discrete node which has numeric values assigned to the states.

Usage

```
NodeValue(node)
NodeValue(node) <- value
```

Arguments

node	An active NeticaNode object that references the node. The node should be continuous or have a numeric value associated with each level (see NodeLevels).
value	A real value for the node.

Details

The behavior of the levels depends on whether the node is discrete (is.discrete(*node*)==TRUE) or continuous (is.discrete(*node*)==FALSE).

Discrete. For if *node* is a discrete node, then the states of *node* must have been assigned numeric values with NodeLevels(*node*) in order for NodeValue() to make sense. If all states have not been assigned values, NodeValue() will generate an error.

If the value of the node is determined, either because the value of the node has been set with NodeFinding() or because the value can be determined exactly from the value of other nodes in the network (through logical probability distributions or formulae), then NodeValue(*node*) will return the value associated with the state of the node. Otherwise, it will return NA.

The expression NodeValue(*node*)<-*value*, can be used to set the value of *node* to the state associate with the numerical *value*. If *value* does not correspond to one of the node levels, this will generate an error.

Continuous. For continuous nodes, NodeFinding(*node*) returns the value associated with the node, either set with a previous call to NodeValue(*node*)<-*value*, or which can be determined through formulae.

The expression NodeValue(*node*) <- *value* will set the value (the equivalent of a finding for a discrete node) to *value*. If *node* has been associated with cut scores through a previous call to NodeLevels(*node*), then this will also associate a finding with the node.

Value

The function NodeValue(*node*) returns the value of *node* if that can be determined (see Details) or NA if it cannot. It may generate an error if node is discrete and has not had numeric values associated with its states.

The expression NodeValue(*node*) <- *value* returns *node*.

Note

Netica manual is not particularly clear on how continuous nodes are handled.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeValueEntered_bn(), EnterNodeValue_bn()

See Also

NodeFinding(), NodeLevels(), EnterNegativeFinding(), EnterFindings(), RetractNodeFinding(), NodeStates(), NodeEquation(), is.continuous(), NodeExpectedValue(), NodeBeliefs()

Examples

```
sess <- NeticaSession()
startSession(sess)

aNet <- CreateNetwork("aNet", session=sess)

dTheta <- NewDiscreteNode(aNet, "ThetaD",
          c("neg2","neg1","zero","pos1","pos2"))
NodeLevels(dTheta) <- c(-2,-1,0,1,2)

NodeFinding(dTheta) <- "pos1"
stopifnot(NodeValue(dTheta)==1)
NodeValue(dTheta) <- 0
stopifnot(NodeFinding(dTheta)=="zero")
## Not run:
## The error handling seems broken under Windows.
cat("This next statement generates an error as 1/2 is not a legal value.")
stopifnot(class(try(NodeValue(dTheta) <- 1/2)) == "try-error")

## End(Not run)

cTheta <- NewContinuousNode(aNet, "ThetaC")
NodeLevels(cTheta) <- qnorm(c(.001,1/5,2/5,2/5,4/5,.999))
## Netica doesn't allow - sign or decimal point in state name, need to
## jump through a few hoops here.
midpoints <- round(qnorm((1:5)/5-.1),2)
NodeStates(cTheta) <- sub(".","o",
    paste(ifelse(midpoints<0,"n","p"),abs(midpoints),sep=""),
    fixed=TRUE)

NodeValue(cTheta) <- -1
stopifnot(NodeFinding(cTheta)=="n1o28")
NodeFinding(cTheta) <- "p0"
## No value associated with this finding.
```

```
stopifnot(is.na(NodeValue(cTheta)))


DeleteNetwork(aNet)
stopSession(sess)
```

NodeVisPos *Gets, sets the visual position of the node on the Netica display.*

Description

When displayed in the GUI, Netica nodes have a position. The `NodeVisPos()` attribute controls where the node will be displayed.

Usage

```
NodeVisPos(node)
NodeVisPos(node) <- value
```

Arguments

`node`	A `NeticaNode` object whose position is to be determined.
`value`	A numeric vector of length 2 giving the x and y coordinates.

Details

The visual position of the node doesn't make much different in RNetica, as R does not display the node. However, it will control the appearance when the net is loaded into the Netica GUI.

Value

A numeric vector of length 2 with names `"x"` and `"y"`.

Note

The minimum possible node position appears to be (0,0) and the maximum is never stated. Netica appears to round positions to the nearest integer. Also, if the position appears too close to the boarder (Netica positions the center of the node), Netica will move it away from the edge.

Author(s)

Russell Almond

References

`http://norsys.com/onLineAPIManual/index.html`: GetNodeVisPosition_bn(), SetNodeVisPosition_bn(),

See Also

NeticaNode, NodeVisPos()

Examples

```
sess <- NeticaSession()
startSession(sess)
pnet <- CreateNetwork("PositionNet", session=sess)

pnode <- NewDiscreteNode(pnet,"PlaceMe")

NodeVisPos(pnode) <- c(100,300)
pos <- NodeVisPos(pnode)
stopifnot(
  pos["x"] ==100,
  pos["y"] ==300
)

## Netica rounds noninteger positions.
NodeVisPos(pnode) <- c(74.3,88.8)
pos <- NodeVisPos(pnode)
stopifnot(
  pos["x"] ==74,
  pos["y"] ==88
)

## Warning, setting a node too close to the edge can cause Netica to
## reposition the node
NodeVisPos(pnode) <- c(1,1)
pos <- NodeVisPos(pnode)
stopifnot(
  pos["x"] >1,
  pos["y"] >1
)

DeleteNetwork(pnet)
stopSession(sess)
```

NodeVisStyle *Gets/sets the nodes visual appearance in Netica.*

Description

Netica internally has a number of styles it can use to draw a node, these including, "Default", "Absent", "Shape", "LabeledBox", "BeliefBars", "BeliefLine", and "Meter". The function NodeVisStyle() returns how the node will be displayed, or sets how it will be displayed.

Usage

```
NodeVisStyle(node)
NodeVisStyle(node) <- value
```

Arguments

node	A NeticaNode object whose style is to be determined.
value	A character string giving the new style. Must be one of "Default", "Absent", "Shape", "LabeledBox", "BeliefBars", "BeliefLine", or "Meter".

Details

The visual style of the node doesn't make much different in RNetica, as R does not display the node. However, it will control the appearance when the node is loaded into the Netica GUI.

Value

A character string which is one of the values "Default", "Absent", "Shape", "LabeledBox", "BeliefBars", "BeliefLine", or "Meter", or NA if an error occurred.

The setter method returns the modified node object.

Note

The Netica documentation indicates that in the future additional parameters can be added to the style, for example: "LabeledBox,CornerRoundingRadius=3,LineThickness=2"

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: GetNodeVisStyle_bn(), SetNodeVisStyle_bn(),

See Also

NeticaNode, NodeVisPos()

Examples

```
sess <- NeticaSession()
startSession(sess)
snet <- CreateNetwork("StylishNet", session=sess)

snode <- NewDiscreteNode(snet,"StyleMe")
stopifnot(NodeVisStyle(snode)=="Default")

NodeVisStyle(snode) <- "Meter"
stopifnot(NodeVisStyle(snode)=="Meter")
```

```
DeleteNetwork(snet)
stopSession(sess)
```

`normalize` *Normalizes a conditional probability table.*

Description

A conditional probability table (CPT) represents a collection of probability distribution, one for each configuration of the parent variables. This function normalizes the CPT, insuring that the probabilities in each conditional distribution sum to 1.

Usage

```
normalize(cpt)
## S3 method for class 'CPF'
normalize(cpt)
## S3 method for class 'data.frame'
normalize(cpt)
## S3 method for class 'CPA'
normalize(cpt)
## S3 method for class 'array'
normalize(cpt)
## S3 method for class 'matrix'
normalize(cpt)
## Default S3 method:
normalize(cpt)
```

Arguments

`cpt`	A conditional probability table stored in either array (`CPA` format) or data frame (`CPF` format). A general data vector is treated like an unconditional probability vector.

Details

The `normalize` function is a generic function which attempts to normalize a conditional probability distribution.

A conditional probability table in RNetica is represented in one of two ways. In the conditional probability array (`CPA`) the table is represented as a $p+1$ dimensional array. The first p dimensions correspond to configurations of the parent variables and the last dimension the child value. The `normalize.CPA` method adjusts the data value so that the sum across all of the child states is 1. Thus, `apply(result,1:p,sum)` should result in a matrix of 1's. The method `normalize.array` first coerces its argument into a `CPA` and then applies the `normalize.CPA` method.

The second way to represent a conditional probability table in RNetica is to use a data frame (`CPF`). Here the factor variables correspond to a configuration of the parent states, and the numeric columns

correspond to states of the child variable. Each row corresponds to a particular configuration of parent variables and the numeric values should sum to one. The `normalize.CPF` function makes sure this constraint holds. The method `normalize.data.frame` first applies `as.CPF()` to make the data frame into a CPF.

The method `normalize.matrix` ensures that the row sums are 1. It does not change the class.

The default method only works for numeric objects. It ensures that the total sum is 1.

`NA`'s are not allowed and will produce a result that is all `NA`s.

Value

An object with similar properties to `cpt`, but adjusted so that probabilities sum to one.

For `normalize.CPA` and `normalize.array` an normalized `CPA` array.

For `normalize.CPF` and `normalize.data.fram` an normalized `CPF` data frame.

For `normalize.matrix` an matrix whose row sums are 1.

For `normalize.default` a numeric vector whose values sum to 1.

Note

May be other functions for CPTs later.

Author(s)

Russell Almond

See Also

`NodeProbs()`

Examples

```
n14 <- normalize(1:4)
stopifnot (abs(sum(n14)-1.0) <.0001)

normalize(matrix(1:6,2,3))
normalize(array(1:24,c(4,3,2)))
arr <- array(1:24,c(4,3,2),
             list(a=c("A1","A2","A3","A4"),
                  b=c("B1","B2","B3"),
                  c=c("C1","C2")))
arr <- as.CPA(arr)
narr <- normalize(arr)
stopifnot(
  is(narr,"CPA"), is(narr,"array"),
  all(abs(apply(narr,1:2,sum)-1) <.0001)
)

arf <- as.CPF(arr)
narf <- normalize(arf)
stopifnot(
```

```
  is(narf,"CPF"), is(narf,"data.frame"),
  all(abs(apply(narf[sapply(narf,is.numeric)],1,sum)-1) <.0001)
)

df2 <- data.frame(parentval=c("a","b"),
                  prob.true=c(1,1),prob.false=c(1,1))
ndf2 <- normalize(df2)
stopifnot(
  is(ndf2,"CPF"), is(ndf2,"data.frame"),
  all(abs(apply(ndf2[2:3],1,sum)-1) <.0001)
)
```

`ParentStates` *Returns a list of the names of the states of the parents of a Netica node.*

Description

This function returns a list each of whose elements is a character vector giving the states of the parent variables (i.e., the result of calling `NodeStates`) on each of the elements of `NodeParents(`*node*`)`). The names of this list are the names assigned to the edges through `NodeInputNames(`*node*`)`, or the names of the parent variables if edge names were not supplied.

Usage

```
ParentStates(node)
ParentNames(node)
```

Arguments

`node`	An active `NeticaNode` object whose parent states are to be determined.

Value

For `ParentStates(`*node*`)`, named list where each element corresponds to the states of a parent variable. If *node* has no parents, it returns a list of length 0.

The function `ParentNames(`*node*`)` returns `names(ParentNames(`*node*`)`, only is faster.

Note

This is a slightly more sophisticated version of `lapply(NodeParents(node), NodeStates)`. It does minimal checking so that it can be fast.

Author(s)

Russell Almond

See Also

NodeStates(), NodeParents(), NodeInputNames()

Examples

```
sess <- NeticaSession()
startSession(sess)
abc1 <- CreateNetwork("ABC1", session=sess)
A <- NewDiscreteNode(abc1,"A",c("A1","A2","A3","A4"))
B <- NewDiscreteNode(abc1,"B",c("B1","B2","B3"))
C <- NewDiscreteNode(abc1,"C",c("C1","C2"))

stopifnot(
  length(ParentStates(A)) == 0
)

AddLink(A,B)

Bpars <- ParentStates(B)
stopifnot(
  length(Bpars) == 1,
  names(Bpars) == "A",
  Bpars$A==NodeStates(A)
)

AddLink(A,C)
AddLink(B,C)

NodeInputNames(C) <- c("A_type","B_type")

Cpars <- ParentStates(C)
stopifnot(
  length(Cpars) == 2,
  names(Cpars) == c("A_type","B_type"),
  Cpars[[1]]==NodeStates(A),
  Cpars$B_type==NodeStates(B)
)

DeleteNetwork(abc1)
stopSession(sess)
```

ReadFindings *Retrieves a record from a Netica Case Stream*

Description

This function reads a row from a Netica case stream and instantiates the values of the listed nodes to the values found in that row of the case stream.

Usage

```
ReadFindings(nodes, stream, pos = "NEXT", add = FALSE)
```

Arguments

nodes	The a list of active NeticaNode objects to be read. The findings of these nodes will be modified by this call.
stream	A CaseStream object which references the file or string object to be read from.
pos	A character or integer scalar. This should almost certainly be one of the two string values "FIRST" or "NEXT". It also can be an integer giving the position (in characters) where to start the machine. This is likely to produce surprising results unless the integer value is a value obtained from calling getCaseStreamPos on this stream earlier after a call to either ReadFindings or WriteFindings.
add	A logical scalar. If true, the findings from the case stream are added to the existing node. If false, they are ignored.

Details

A case file is a table where the rows represent cases, and the columns represent variables. ReadFindings reads a row out the table and instantiates (NodeFinding) the nodes in nodeset to those values. If a the value corresponding a node is the value of CaseFileMissingCode(), then it is not instantiated. The values in the columns are separated by the value of CaseFileDelimiter().

If add is false, it will first retract any findings associated with the nodes in nodeset. If a finding is associated with a node, the the case file would cause it to be set to an inconsistent value, then an error will be generated.

The argument pos determines which record will be read next. If the value is "NEXT" the next code will be read. If the value is "FIRST" the first code will be read. If the value is a positive integer, then the record which starts at that character will be read. On completion of the read, the value of getCaseStreamPos(stream) is set to the starting position of the last read stream. This is also true when WriteFindings is called. It is almost certainly an error to set the pos argument to anything but either one of the special string constants or a value which was previously cached after calling getCaseStreamPos. If the case stream is at the end, then getCaseStreamPos(stream) will be set to NA.

There are two special columns in the file. The column "IDnum" contains ID numbers for the cases. The value of getCaseStreamLastId(stream) is set to the value of this column if it is present in the case stream, otherwise it will be set to -1. The value of the column "NumCases" contains a weight to give to the current row. The value of getCaseStreamLastFreq(*stream*) is set to this value, if it is present. The returned stream object will have these updated properties, otherwise it will be set to -1.

Value

Returns the *caseOrStream* argument invisibly. Note that the values of getCaseStreamPos(*stream*) will return the position of the next record or NA if there are no records left in the stream. The values of getCaseStreamLastId(*stream*), and getCaseStreamLastFreq(*stream*) will be updated to reflect the values from the last read record, or will be -1 if these values are not provided in the stream.

Note

The first time that ReadFindings is called on a stream it must be called with pos="FIRST". Failing to do so produces a fatal error in Netica.

The value of case_posn returned by the Netica ReadNetFindings2_bn function (which is the value to which getCaseStreamPos(stream)) is undocumented. I confirmed with Brent that this is in fact the position in characters from the start of the stream to the record. It is not recommended, however, that program rely on that fact.

The fact that the case positions are difficult to compute makes random access difficult. If it is needed, programmers will need to save the values of getCaseStreamPos on previous calls to ReadFindings or WriteFindings. Fetching cases by the ID requires scanning through the case file (see WithOpenCaseStream for an example).

Author(s)

Russell G. Almond

References

http://norsys.com/onLineAPIManual/index.html: ReadNetFindings2_bn()

See Also

CaseFileDelimiter, CaseFileMissingCode, NodeFinding, RetractNetFindings ReadFindings, CaseStream, WithOpenCaseStream

Examples

```
sess <- NeticaSession()
startSession(sess)

abc <- CreateNetwork("ABC", session=sess)
A <- NewDiscreteNode(abc,"A",c("A1","A2","A3","A4"))
B <- NewDiscreteNode(abc,"B",c("B1","B2","B3"))
C <- NewDiscreteNode(abc,"C",c("C1","C2"))

AddLink(A,B)
AddLink(A,C)
AddLink(B,C)

## Input filename
## Note, this is a cached copy of the file written in the WriteFindings
## documentation.
casefile <- file.path(library(help="RNetica")$path,
                          "testData","abctestcases.cas")
filestream <- CaseFileStream(casefile, session=sess)
## Case 1
filestream <- ReadFindings(list(A,B,C),filestream,"FIRST")
stopifnot( NodeFinding(A) == "A1",
           NodeFinding(B) == "B1",
           NodeFinding(C) == "C1",
```

```
            getCaseStreamLastId(filestream)==1001,
            abs(getCaseStreamLastFreq(filestream)-1.0) < .0001)

pos1 <- getCaseStreamPos(filestream)

## Case 2
filestream <- ReadFindings(list(A,B,C),filestream,"NEXT")
stopifnot( NodeFinding(A) == "A2",
            NodeFinding(B) == "B2",
            NodeFinding(C) == "C2",
            getCaseStreamLastId(filestream)==1002,
            abs(getCaseStreamLastFreq(filestream)-2.0) < .0001)

## Case 3
filestream <- ReadFindings(list(A,B,C),filestream,"NEXT")
stopifnot( NodeFinding(A) == "A3",
            NodeFinding(B) == "B3",
            NodeFinding(C) == "@NO FINDING",
            getCaseStreamLastId(filestream)==1003,
            abs(getCaseStreamLastFreq(filestream)-1.0) < .0001)


## At end of file
filestream <- ReadFindings(list(A,B,C),filestream,"NEXT")
stopifnot(is.na(getCaseStreamPos(filestream)))

## Restart from Case 1
filestream <- ReadFindings(list(A,B,C),filestream,"FIRST")
stopifnot( NodeFinding(A) == "A1",
            NodeFinding(B) == "B1",
            NodeFinding(C) == "C1",
            getCaseStreamLastId(filestream)==1001,
            abs(getCaseStreamLastFreq(filestream)-1.0) < .0001,
            pos1 == getCaseStreamPos(filestream))

## Test with memory stream
cases <- read.CaseFile(casefile, session=sess)
abcstream <- CaseMemoryStream(cases, session=sess)
MemoryStreamContents(abcstream)


abcstream <- ReadFindings(list(A,B,C),abcstream,"FIRST")
stopifnot( NodeFinding(A) == "A1",
            NodeFinding(B) == "B1",
            NodeFinding(C) == "C1",
            getCaseStreamLastId(abcstream)==1001,
            abs(getCaseStreamLastFreq(abcstream)-1.0) < .0001)


##Clean Up
CloseCaseStream(filestream)
CloseCaseStream(abcstream)
DeleteNetwork(abc)
```

```
stopSession(sess)
```

`RetractNodeFinding` *Clears any findings for a Netica node or network.*

Description

The function `RetractNodeFinding(`*node*`)` clears any findings or virtual findings set with `NodeFinding()`, `EnterNegativeFinding()` or `NodeLikelihood()` and associated with *node*. The function `RetractNetFindings(`*net*`)` clears any findings associated with any node in the network.

Usage

```
RetractNodeFinding(node)
RetractNetFindings(net)
```

Arguments

`node`	An active `NeticaNode` whose findings are to be retracted.
`net`	An active `NeticaBN` whose findings are to be retracted.

Details

This is an undo function for `NodeFinding()`, `EnterNegativeFinding()` or `NodeLikelihood()`. In particular, it allows for entering hypothesized findings for various calculations.

Value

Returns its argument invisibly.

Note

If `SetNetworkAutoUpdate()` has been set to `TRUE`, then this function could take some time as each finding is individually propagated. Consider wrapping multiple calls setting `NodeFinding()` in `WithoutAutoUpdate(net, ...)`.

The Netica functions for setting node findings require the programmer to call `RetractNodeFindings_bn()` before setting values to clear out old findings. The RNetica functions do this internally, so the user does not need to worry about this.

Author(s)

Russell Almond

References

`http://norsys.com/onLineAPIManual/index.html`: RetractNetFindings_bn(), RetractNodeFindings_bn()

See Also

NeticaBN, NeticaNode, NodeBeliefs(), EnterNegativeFinding(), EnterGaussianFinding(), EnterIntervalFinding(), NodeFinding(), NodeLikelihood()

Examples

```
sess <- NeticaSession()
startSession(sess)
irt5 <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","IRT5.dne"), session=sess)

irt5.theta <- NetworkFindNode(irt5,"Theta")
irt5.x <- NetworkFindNode(irt5,paste("Item",1:5,sep="_"))

CompileNetwork(irt5) ## Ready to enter findings

stopifnot(NodeFinding(irt5.x[[1]]) == "@NO FINDING")

NodeFinding(irt5.x[[1]]) <- "Right"
stopifnot(NodeFinding(irt5.x[[1]]) == "Right")

RetractNodeFinding(irt5.x[[1]])
stopifnot(NodeFinding(irt5.x[[1]]) == "@NO FINDING")

NodeFinding(irt5.x[[1]]) <- "Wrong"
NodeFinding(irt5.x[[2]]) <- 1
NodeFinding(irt5.x[[3]]) <- 2
stopifnot(
  NodeFinding(irt5.x[[1]]) == "Wrong",
  NodeFinding(irt5.x[[2]]) == "Right",
  NodeFinding(irt5.x[[3]]) == "Wrong",
  NodeFinding(irt5.x[[4]]) == "@NO FINDING",
  NodeFinding(irt5.x[[5]]) == "@NO FINDING"
)

RetractNetFindings(irt5)
stopifnot(
  NodeFinding(irt5.x[[1]]) == "@NO FINDING",
  NodeFinding(irt5.x[[2]]) == "@NO FINDING",
  NodeFinding(irt5.x[[3]]) == "@NO FINDING",
  NodeFinding(irt5.x[[4]]) == "@NO FINDING",
  NodeFinding(irt5.x[[5]]) == "@NO FINDING"
)

DeleteNetwork(irt5)
stopSession(sess)
```

ReverseLink *Reverses a link in a Netica network.*

Description

This reverses the link between *parent* and *child* so that it now points from *child* to *parent*. If *child* has additional parents, they are connected to *parent* and the conditional probability tables are adjusted so that the joint probability distribution across all nodes in the network remains the same.

Usage

```
ReverseLink(parent, child)
```

Arguments

`parent`	An active `NeticaNode` which is currently a parent of *child* and which will be the child after the transformation.
`child`	An active `NeticaNode` which is currently a child of *parent* and which will be the parent after the transformation.

Details

This is not just a simple reversal of a single edge, but rather the influence diagram operation of *arc reversal*. Netica will add additional links to enforce any conditional probability relationship. For example, Consider a net where `A` and `B` are both parents of `C`, but `A` is not directly connected to `C`. After reversing the arc between `B` and `C`, `A` will also become a parent of `B` to maintain the joint distribution.

Value

Returns `NULL` if successful and `NA` if there was a problem.

Author(s)

Russell Almond

References

`http://norsys.com/onLineAPIManual/index.html`: ReverseLink_bn()

Shachter, R. D. (1986) "Evaluating Influence Diagrams." *Operations Research*, **34**, 871–82.

See Also

`NeticaNode`, `AddLink()`, `NodeChildren()`, `NodeParents()`, `AbsorbNodes()`, `is.NodeRelated()`

Examples

```
sess <- NeticaSession()
startSession(sess)

abcnet <- CreateNetwork("ABC", session=sess)

A <- NewDiscreteNode(abcnet,"A")
B <- NewDiscreteNode(abcnet,"B")
```

```
C <- NewDiscreteNode(abcnet,"C")

AddLink(A,C)
AddLink(B,C)
stopifnot(
  is.NodeRelated(A,C,"parent"),
  is.NodeRelated(C,B,"child"),
  !is.NodeRelated(A,B,"parent")
)

ReverseLink(B,C)
stopifnot(
  is.NodeRelated(A,C,"parent"),
  is.NodeRelated(C,B,"parent"),
  is.NodeRelated(A,B,"parent")
)

DeleteNetwork(abcnet)
stopSession(sess)
```

StartNetica *Starting and stopping the Netica shared library.*

Description

This function creates (or destroys) a Netica environment. The StartNetica function also allows you to set various parameters associated with the Netica environment.

Usage

```
startSession(session)
stopSession(session)
restartSession(session)
StartNetica(license = LicenseKey, checking = NULL, maxmem = NULL,
           session=NeticaSession(LicenseKey=license, checking=checking,
                                 maxmem=maxmem))
StopNetica(session=getDefaultSession())
```

Arguments

session	An object of class NeticaSession which encapsulates the connection to Netica.
license	A string containing a license key from Norsys. If this is NULL the limited student/demonstration version of Netica is used rather than the full version. If the variable NeticaLicenseKey is set before RNetica is loaded, then the value of that variable at the time the package is loaded will become the default for *license*.

checking — A character string containing one of the keywords: "NO_CHECK", "QUICK_CHECK", "REGULAR_CHECK", "COMPLETE_CHECK", or "QUERY_CHECK", which controls how rigorous Netica is about checking errors. A value of NULL uses the Netica default which is "REGULAR_CHECK".

maxmem — An integer containing the maximum amount of memory to be used by the Netica shared library in bytes. If supplied, this should be at least 200,000.

Details

The generic functions startSession and stopSession start and stop the connection to Netica. Note that the session is active (is.active) if the session has been started, and inactive if it was stopped (or not yet started). The generic function restartSession stops the session and starts it again.

The functions StartNetica and StopNetica are depricated, but still included for backwards compatability. The function StartNetica will now create a new object of class NeticaSession passing its arguments to the constructor. It will also set a variable DefaultNeticaSession in the global environment to the new session. As this is the value returned by getDefaultSession() this will cause RNetica to behave like the previous versions where the session pointer was stored internally to the C code. The function StopNetica operates on the default session if no explicit argument is given.

Netica is commercial software. The RNetica package downloads and installs the demonstration version of Netica which is limited in its functionality (particularly in the size of the networks it handles). Unlocking the full version of Netica requires a license key which can be purchased from Norsys (http://www.Norsys.com/). They will send a license key which unlocks the full capabilities of the shared library. This can be passed as the first argument to StartNetica(). If the value of the first argument is NULL then the demonstration version is used instead of the licensed version (could be useful for testing).

Prior to RNetica version 0.5, RNetica looked for a variable called NeticaLicenseKey in the global workspace when RNetica is loaded. This then becomes the default value for both StartNetica and getDefaultSession(). If no value is set for NeticaLicenseKey, the default value for license is set to NULL, which loads the demo version of Netica.

In version 0.5 and later of RNetica, the recommended way to store the license is to create a default session and store it in the variable DefaultNeticaSession in the global environment. This session object then contains the license key.

The checking argument, if supplied, is used to call the Netica function ArgumentChecking_ns(). See the documentation of that function for the meaning of the codes. The default value, "REGULAR_CHECK" is appropriate for most development situations.

The maxmem argument, if supplied, is used to limit the amount of memory used by Netica. This is passed in a call to the Netica function LimitMemoryUsage_ns(). Netica will complain if this value is less than 200,000. Leaving this as NULL will not place limits on the size of Netica's memory for tables and things.

The function StopNetica() calls the Netica function CloseNetica_bn(). It is mainly used when one wants to stop Netica and restart it with other parameters.

As of RNetica 0.5, the function StartNetica is no longer called when the package is attached (in the .onAttach() function). Instead, users should start Netica scripts with startSession(DefaultNeticaSession).

Note that the pathnames of recently loaded networks are stored in the session object, so that networks can be quickly re-read from a saved session.

Value

These functions now all return an object of class NeticaSession. Note that StartNetica sets the value of DefaultNeticaSession in the global environment to the value of the session argument.

License

The Netica API is not free-as-in-speech software, the use of the Netica shared library makes you subject to the Netica License agreement (which can be found in the RNetica folder in your R library. If you do not agree to the terms of that license, please uninstall RNetica.

The Netica API is also not free-as-in-beer software. The demonstration version of the Netica API, however, is. In order for you to make full use of the RNetica API, you must purchase a Netica API license from Norsys (http://norsys.com/).

RNetica itself (the glue layers between R and Netica) is free (in both the speech and beer senses) software. Suggestions for improvements and bug fixes are welcome.

Implementation Notes

Starting with RNetica 0.5, the Netica environment pointer, which is used by the Netica shared library is stored inside of the NeticaSession object instead of as a global object in the C code. The function is.active() checks to see whether that pointer is set (active) or null (inactive). Note that when a session object is saved, the session pointer is not saved and should be set to null.

The pre verison 0.5 method used a variable NeticaLicenseKey in the global environment to store the license key. The post 0.5 method uses a variable DefaultNeticaSession to store a session object instead, but for backwards compatability, it will try to create a session using the value of NeticaLicenseKey if no default session exists.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: NewNeticaEnviron_ns(), InitNetica2_bn(), CloseNetica_bn(), LimitMemoryUsage_ns(), ArgumentChecking_ns()

See Also

NeticaSession, NeticaSession(), NeticaVersion(), CreateNetwork()

Examples

```
## Not run:
## Recommended way of doing things
DefaultNeticaSession <- NeticaSession(LicenseKey="Code from Norsys")
startSession(DefaultNeticaSession)
```

```
## If DefaultNeticaSession was created in a previous R session
## Re-read the networks.
for (netname in DefaultNeticaSession$listNets()) {
  net <- DefaultNeticaSession$findNet(netname)
  ReadNetworks(GetNetworkFileName(net),DefaultNeticaSession)
}

restartSession(DefaultNeticaSession)
stopSession(DefaultNeticaSession)

## Depricated methods
StartNetica("License key from Norsys")
StopNetica()
## Get the version of Netica.


## End(Not run)
```

`WithOpenCaseStream` *Evaluate an expression and then close the Netica Case Stream.*

Description

This function evaluates *expr* in a context where the `CaseStream` is open. The stream is closed when the evaluation is complete. The evaluation of *expr* is surrounded with a `tryCatch` so that the stream is closed whether or not the expression is successfully executed.

Usage

```
WithOpenCaseStream(stream, expr)
```

Arguments

`stream`	A `CaseStream` object. This can be open or closed. If closed it is reopened.
`expr`	An arbitrary R expression to be executed.

Value

Either the result of evaluating *expr* unless executing *expr* results in an error in which case it returns a `try-error`.

Author(s)

Russell Almond

See Also

`CaseStream`,`ReadFindings`

Examples

```
## This function reads findings from a stream until it finds one
## matching a certain case ID.
ReadCase <- function (stream,nodes,caseID) {
  WithOpenCaseStream(stream,
    {stream <- ReadFindings(nodes,stream,"FIRST")
     while(!is.na(getCaseStreamPos(stream)) &&
           getCaseStreamLastId(stream) != caseID) {
       ReadFindings(nodes,stream,"NEXT")
     }
     if (is.na(getCaseStreamPos(stream))) {
       warning("Case ID:",caseID," not found in stream.")
     }
     stream
   })
}

sess <- NeticaSession()
startSession(sess)


## Test it.
abc <- CreateNetwork("ABC", session=sess)
A <- NewDiscreteNode(abc,"A",c("A1","A2","A3","A4"))
B <- NewDiscreteNode(abc,"B",c("B1","B2","B3"))
C <- NewDiscreteNode(abc,"C",c("C1","C2"))

AddLink(A,B)
AddLink(A,C)
AddLink(B,C)

## Input filename
## Note, this is a cached copy of the file written in the WriteFindings
## documentation.
casefile <- file.path(library(help="RNetica")$path,
                          "testData","abctestcases.cas")


filestream <- ReadCase(CaseFileStream(casefile, session=sess),list(A,B,C),1002)
stopifnot( !isCaseStreamOpen(filestream),
           NodeFinding(A) == "A2",
           NodeFinding(B) == "B2",
           NodeFinding(C) == "C2",
           getCaseStreamLastId(filestream)==1002,
           abs(getCaseStreamLastFreq(filestream)-2.0) < .0001)

##Clean Up
DeleteNetwork(abc)
stopSession(sess)
```

woe *Calculates the weight of evidence for a hypothesis*

Description

Calculates the weight of evidence provided by the current findings for the specified hypothesis. A hypothesis consists of a statement that a particular set of nodes (*hnodes*) will fall in a specified set of states (*hstatelists*).

Usage

```
woe(enodes, estates, hnodes, hstatelists)
```

Arguments

enodes	A list of NeticaNodes which are providing evidence. As a special case, a single NeticaNode is treated as a list of length one.
estates	A list of character vectors the same length as *enodes* corresponding to the observed or hypothesized state of the evidence nodes and representing states of the corresponding node. As a special case, a character vector is turned into a list of length one.
hnodes	A list of NeticaNodes whose values are of interest. As a special case, a single NeticaNode is treated as a list of length one.
hstatelists	A list of character vectors the same length as *hnodes* corresponding to the hypothesized state of the nodes and representing states of the corresponding node. As a special case, a character vector is turned into a list of length one.

Details

Good (1985) defines the weight of evidence E for a hypothesis H as

$$W(H:E) = log\frac{P(E|H)}{P(E|\not H)} = log\frac{P(H|E)}{P(\not H|E)} - log\frac{P(H)}{P(\not H)}.$$

Author(s)

Russell Almond

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (hnodes, hstatelists)
{
    if (!is.list(hnodes))
```

```
    hnodes <- list(hnodes)
if (!is.list(hstatelistss))
    hstatelists <- list(hstatelists)
if (!all(sapply(hnodes, is.NeticaNode))) {
    stop("Expected a list of Netica nodes, got ", hnodes)
}
if (length(hstatelists) > length(hnodes)) {
    stop("More statelists than nodes.")
}
net <- NodeNetwork(hnodes[[1]])
hlikes <- mapply(function(hnode, hstatelist) {
    stnames <- NodeStateNames(hnode)
    hlike <- rep(0, length(stnames))
    names(hlike) <- stnames
    hlike[hstatelist] <- 1
    hlike
}, hnodes, hstatelists, SIMPLIFY = FALSE)
tryCatch({
    for (i in 1:length(hnodes)) {
        RetractNodeFinding(hnodes[[i]])
        NodeLikelihood(hnodes[[i]]) <- hlike[[i]]
    }
    p_Htrue <- FindingsProbability(net)
    for (i in 1:length(hnodes)) {
        RetractNodeFinding(hnodes[[i]])
        NodeLikelihood(hnodes[[i]]) <- 1 - hlike[[i]]
    }
    p_Hfalse <- FindingsProbability(net)
    100 * log10(p_Htrue/p_Hfalse)
}, finally = sapply(hnodes, RetractNodeFinding))
}
```

`write.CaseFile` *Read or write data frame in Netica Case File format.*

Description

These functions our wrapper around `read.table` and `write.table` to format the file in the expected Netica case file format.

Usage

```
write.CaseFile(x, file, ..., session=getDefaultSession())
read.CaseFile(file, ..., session=getDefaultSession())
```

Arguments

`x`	A data frame to be written to the file. See details.
`file`	A file name or a connection object. By convention, Netica expects case files to end in the ".cas" suffix.

...	Other arguments to read.table or write.table
session	An object of class NeticaSession which encapsulates the connection to Netica. Used to find the current delimiters.

Details

A Netica case file has a format that very much resembles the output of write.table. The first row is a header row, which contains the names of the variables, the second and subsequent rows contain a set of findings: an assignment of values to the nodes indicated in the columns. There are no row numbers, and the separator and missing value codes are the values of CaseFileDelimiter(), and CaseFileMissingCode() respectively.

In addition to columns representing variables, two special columns are allowed. The column named "IDnum", if present should contain integers which correspond to ID numbers for the cases (this correspond to the id argument of WriteFindings). The column named "NumCases" should contain number values and this allows rows to be differentially weighted (this correspond to the freq argument of WriteFindings). If these special arguments are present, write.table permutes the columns if necessary to make them first in the order (as Netica does in WriteFindings).

The function read.CaseFile overrides following arguments of read.table: header = TRUE, sep = CaseFileDelimiter(), and na.strings = CaseFileMissingCode(). The function write.CaseFile overrides following arguments of write.table: col.name = TRUE, row.names = FALSE, quote = FALSE, sep = CaseFileDelimiter(), and na = CaseFileMissingCode().

Value

The function read.CaseFile returns a data frame containing the information in the case file. The function write.CaseFile returns the output of the write.table call (which is undocumented).

Author(s)

Russell Almond

See Also

CaseFileDelimiter, CaseFileMissingCode, WriteFindings, ReadFindings, CaseMemoryStream,CaseFileStream, MemoryStreamContents, read.table,write.table

Examples

```
sess <- NeticaSession()
startSession(sess)

casefile <- file.path(library(help="RNetica")$path,
                          "testData","abctestcases.cas")
CaseFileDelimiter("\t", session=sess)
CaseFileMissingCode("*", session=sess)
cases <- read.CaseFile(casefile, session=sess)

outfile <- tempfile("testcase",fileext=".cas")
write.CaseFile(cases,outfile, session=sess)
```

```
stopSession(sess)
```

`WriteFindings` *Appends the current findings to a Netica case file.*

Description

This function writes the current findings for a network as a row in a Netica case file. If *filename* already exists, the new row is appended on the end of the file. Variables that are not instantiated are written out using the missing code.

Usage

```
WriteFindings(nodes, pathOrStream, id = -1L, freq = -1.0)
```

Arguments

`nodes`	The a list of active `NeticaNode` objects to be written out.
`pathOrStream`	Either a character scalar giving the path name of the file to which the results are to be written, or a `CaseStream` object. It is recommended that it have the extension ".cas".
`id`	An integer scalar giving the case ID. The default value of `-1` suppresses the writing of cases. If an ID is supplied for the first case, it should be supplied for all cases.
`freq`	An integer scalar giving the number of cases with the currently instantiated set of findings. The default value `-1` suppresses writing the cases, implicitly assuming that all cases have weight 1. If supplied for the first row, this should be supplied for all rows.

Details

A case file is a table where the rows represent cases, and the columns represent variables. `WriteFindings` writes out the currently instantiated value of the nodes in *nodeset*. If a node in *nodeset* does not currently have a finding attached, then the value of `CaseFileMissingCode()` is printed out instead. The values in the columns are separated by the value of `CaseFileDelimiter()`.

There are two special columns in the file. The column "IDnum" is set to the value `id`, which should contain an integer case number. The column "NumCases" is set to the value of `freq` which should give a weight to assign to the case (in various algorithms when `freq` is supplied, it is treated as if that case was repeated `weight` times). Assigning either of these fields a value of `-1` means the corresponding column is appended to the output.

The function `WriteFindings` will create a new file associated with `filename` if it does not exist. In that case it will write out a header row containing the variable names followed by the current findings as the first case row. Subsequent calls to `WriteFindings` with the same `filename` append additional rows to the end of the file. In such cases, the `nodelist` should be the same, and if `id` or `freq` was `-1`, it should be in the following calls as well.

Value

Returns the *caseOrStream* argument invisibly. Note that the values of `getCaseStreamPos(stream)`, `getCaseStreamLastId(stream)`, and `getCaseStreamLastFreq(stream)` will be updated to reflect the values from the last read record.

Author(s)

Russell G. Almond

References

http://norsys.com/onLineAPIManual/index.html: WriteNetFindings_bn()

See Also

`CaseFileDelimiter`, `CaseFileMissingCode`, `NodeFinding`, `RetractNetFindings` `ReadFindings`, `CaseStream`

Examples

```
sess <- NeticaSession()
startSession(sess)

abc <- CreateNetwork("ABC", session=sess)
A <- NewDiscreteNode(abc,"A",c("A1","A2","A3","A4"))
B <- NewDiscreteNode(abc,"B",c("B1","B2","B3"))
C <- NewDiscreteNode(abc,"C",c("C1","C2"))

AddLink(A,B)
AddLink(A,C)
AddLink(B,C)

## Outputfilename
casefile <- tempfile("testcase",fileext=".cas")
filestream <- CaseFileStream(casefile, session=sess)
stopifnot(is.CaseFileStream(filestream),
          isCaseStreamOpen(filestream))

## Case 1
NodeFinding(A) <- "A1"
NodeFinding(B) <- "B1"
NodeFinding(C) <- "C1"
WriteFindings(list(A,B,C),casefile,1)
RetractNetFindings(abc)

## Case 2
NodeFinding(A) <- "A2"
NodeFinding(B) <- "B2"
NodeFinding(C) <- "C2"
WriteFindings(list(A,B,C),casefile,2)
RetractNetFindings(abc)
```

```
## Case 3
NodeFinding(A) <- "A3"
NodeFinding(B) <- "B3"
## C will be missing
WriteFindings(list(A,B,C),casefile,3)
RetractNetFindings(abc)


DeleteNetwork(abc)
stopSession(sess)
```

`WriteNetworks` *Reads or writes a Netica network from a file.*

Description

This function writes a Netica network to a `.neta` or `.dne` file or reads a network written by such a file. This allows networks created with RNetica to be shared with other Netica users.

Usage

```
WriteNetworks(nets, paths)
ReadNetworks(paths, session)
GetNetworkFileName(net, internal=FALSE)
```

Arguments

`nets`	A single `NeticaBN` object or a list of such objects.
`net`	A single `NeticaBN` object.
`paths`	A character vector of pathnames to `.neta` files. For `ReadNetworks()`, the pathnames must exist. For `WriteNetworks()`, the `length(`*paths*`)` must equal `length(`*nets*`)`. For `WriteNetworks()` if `paths` are missing, then `GetNetworkFileName()` will be called to try and determine any path associated with the node.
`session`	An object of type `NeticaSession` which defines the reference to the Netica workspace. Read networks will be created in this session.
`internal`	A logical scalar. If true, the actual Netica object will be consulted, if false, a cached value in the R object will be used.

Details

This method invokes the native Netica open and save functions to read and write networks to `.neta` or `.dne` files. The `.neta` format is binary and more compact, while the `.dne` format is ASCII and may be safer in some circumstances (such as when used with a source control system). Netica figures out which format to use based on the extension of the file, elements of `paths` should end with `.neta` or `.dne`.

The function GetNetworkFileName() returns the name of the last file this network was saved to or read from. It cannot be set other than through the WriteNetworks() or ReadNetworks() functions. Note that the filename is saved in both the R object and the Netica object. If internal=TRUE, then the Netica object will be consulted. This will raise an error if the *net* is not currently active.

To facilitate saving and restoring files across R sessions, both ReadNetworks() and WriteNetworks() attach a "Filename" attribute to the object, which records the file just read or written. GetNetworkFileName(*net*,internal= will not work after quitting and restarting Netica, but net$PathnameName should contain the same pathname. If ReadNetworks() is passed a NeticaBN object (or a list of such objects), it will attempt to read from the file referenced by the "Filename" attribute. Thus, calling *net* <- WriteNetworks(*net*,*path*) right before shutting down R and *net* <- ReadNetworks(*net*,*session*) right after the call to startSession() should restore the network.

Value

Both ReadNetworks() and WriteNetworks() return a list of NeticaBN objects corresponding to the new networks. In the case of a problem with one of the networks, the corresponding entry will be set to NULL. If the return list has length 1, a single NeticaBN object will be returned instead of a list.

A "PathnameName" field is added to the NeticaBN object that is returned. This can be used to restore NeticaBN objects after an R session is restarted.

Note

The demonstration version of Netica is limited to the size of the networks it will write (the limit is somewhere around 10 nodes). If you are running across errors saving large networks, you need to purchase a Netica API license from Norsys (http://norsys.com/).

ReadNetworks() and WriteNetworks() are vectorized, and can take either scalars or vectors as arguments (thus, a whole collection of networks can be read or written at once). When the argument is a scalar, a scalar is returned. This is probably the 80% case, but may produce unexpected behavior in certain coding circumstances.

Netica does not seem to expand the character '~' to your home directory (unlike R which follows the Unix convention in this regard).

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html: WriteNet_bn(), ReadNet_bn()

See Also

NeticaBN, CreateNetwork(), NetworkFindNode() (for recreating links to nodes after restoring a net)

Examples

```
sess <- NeticaSession()
startSession(sess)

peanut <- CreateNetwork("peanut", session=sess)
NetworkTitle(peanut) <- "The Peanut Network"
peanutFile <- tempfile("peanut",fileext=".dne")
WriteNetworks(peanut,peanutFile)
stopifnot(GetNetworkFileName(peanut)==peanutFile)

pecan <- CreateNetwork("pecan", session=sess)
NetworkTitle(pecan) <- "The Pecan Network"
pecanFile <- tempfile("pecan",fileext=".dne")
almond <- CreateNetwork("almond", session=sess)
NetworkTitle(almond) <- "The Almond Network"
almondFile <- tempfile("almond",fileext=".neta")
WriteNetworks(list(pecan,almond),c(pecanFile,almondFile))
stopifnot(GetNetworkFileName(pecan)==pecanFile,
          GetNetworkFileName(almond)==almondFile)

DeleteNetwork(peanut)
DeleteNetwork(pecan)
DeleteNetwork(almond)
stopifnot(!is.active(almond))

peanut <- ReadNetworks(peanutFile, session=sess)
stopifnot(is.active(peanut))
stopifnot(NetworkTitle(peanut)=="The Peanut Network")
stopifnot(GetNetworkFileName(peanut)==peanutFile)

nets <- ReadNetworks(c(pecanFile,almondFile), session=sess)
stopifnot(length(nets)==2)
stopifnot(all(sapply(nets,is.active)))
stopifnot(NetworkTitle(nets[[1]])=="The Pecan Network")
almond <- GetNamedNetworks("almond", session=sess)
stopifnot(is.NeticaBN(almond),is.active(almond))
stopifnot(GetNetworkFileName(pecan)==pecanFile,
          GetNetworkFileName(almond)==almondFile)

DeleteNetwork(peanut)
DeleteNetwork(nets[[1]])
DeleteNetwork(almond)
stopSession(sess)

## Not run:
## Safe way to preserve node and network objects across R sessions.
tnet <- WriteNetworks(tnet,"Tnet.neta")
q(save="yes")
# R
library(RNetica)
sess <- startSession(getDefaultSession())
tnet <- ReadNetworks(tnet, session=sess)
```

```
nodes <- NetworkFindNodes(tnet,tnet$listNodes())

## End(Not run)
```

Index