Package 'RNetica'

July 5, 2013

Version 0.3-1

Date 2012/11/22

Title R interface to Netica(R) Bayesian Network Engine

Author Russell Almond

Maintainer Russell Almond <ralmond@fsu.edu>

Depends R (>= 2.0), methods, utils, grDevices

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

License Artistic-2.0 + file LICENSE

URL http://ralmond.net/RNetica

R topics documented:

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

Description

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

Details

Package:	RNetica
Version:	0.2-6
Date:	2012/11/22
Depends:	R (>= 2.0), methods, utils, grDevices
(http:	//norsys.com/) Bayesian network library API
License:	Artistic-2.0 + file LICENSE
URL:	http://ralmond.net/RNetica
Built:	R 2.15.2; x86_64-pc-linux-gnu; 2012-11-22 19:38:18 UTC; unix

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 two ways. First, it can be used as an argument to the function `StartNetica()`. As this function is called when RNetica is loaded, you may need to call `StopNetica()` first and restart the licensed version. Alternatively, if you set the variable `NeticaLicenseKey` in the R top-level environment before the call to `library(RNetica)`, RNetica 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.

Index

	Netica.
NeticaNode	An object referencing a node in a Netica Bayesian network.
NeticaVersion	Fetches the version number of Netica.
NetworkFindNode	Finds nodes in a Netica network.
NetworkFootprint	Returns a list of names of unconnected edges.
NetworkName	Gets or Sets the name of a Netica network.
NetworkNodeSetColor	Returns or sets a display colour to use with a netica node.b
NetworkNodeSets	Returns a list of node sets associated with a Netica network.
NetworkNodesInSet	Returns a list of node labeled with the given node set in a Netica Network.
NetworkSetPriority	Changes the priority order of the node sets.
NetworkTitle	Gets the title or comments associated with a Netica network.
NetworkUndo	Undoes (redoes) a Netica operation on a network.
NetworkUserField	Gets user definable fields associated with a Netica network.
NewDiscreteNode	Creates (or destroys) a node in a Netica Bayesian network.
NodeBeliefs	Returns the current marginal probability distribution associated with a node in a Netica network.
NodeChildren	Returns a list of the children of a node in a Netica network.
NodeFinding	Returns of sets the observed value associated with a Netica node.
NodeInputNames	Associates names with incomming edges on a Netica node.
NodeKind	Gets or changes the kind of a node in a Netica network.
NodeLevels	Accesses the levels associated with a Netica node.
NodeLikelihood	Returns or sets the virtual evidence associated with a Netica node.
NodeName	Gets or set of a Netica node.
NodeNet	Finds which Netica network a node comes from.
NodeParents	Gets or sets the parents of a node in a Netica network.
NodeProbs	Gets or sets the conditional probability table associated with a Netica node.
NodeSets	Lists or changes the node sets associated with a Netica node.
NodeStateTitles	Accessors for the titles and comments associated with states of Netica nodes.
NodeStates	Accessor for states of a Netica node.

NodeTitle	Gets the title or Description associated with a Netica node.
NodeUserField	Gets user definable fields associated with a Netica node.
NodeVisPos	Gets, sets the visual position of the node on the Netica display.
NodeVisStyle	Gets/sets the nodes visual appearence in Netica.
ParentStates	Returns a list of the names of the states of the parents of a Netica node.
RetractNodeFinding	Clears any findings for a Netica node or network.
ReverseLink	Reverses a link in a Netica network.
StartNetica	Starting and stopping the Netica shared library.
WriteNetworks	Reads or writes a Netica network from a file.
is.NodeRelated	Computes topological properities of a 'Netica' network.
is.active	Check to see if a Netica network or node object is still valid.
is.discrete	Determines whether a Netica node is discrete or continuous.
normalize	Normalizes a conditional probability table.

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.

The function `StartNetica()` (invoked automatically when `library(RNetica)`) builds a Netica environment which can be accessed from R. Networks created and loaded into the RNetica environment can then be manipulted from inside of R. 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.

The key to this process is that the two most common functions for creating networks, `CreateNetwork()` and `ReadNetworks()` both return a special object of class `NeticaBN` which encapsulates a pointer back to the Bayesian network in the RNetica environment. This object can be manipulated with the functions in this package.

Netica nodes (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) are represented as special objects of class `NeticaNode` which contain pointers to the node in a Netica network. Netica nodes know which network they belong to, so each node implicitly references its network.

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 correspoinding 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.

Because of the way R likes to hang onto references to objects, it is quite possible for a NeticaBN or NeticaNode object to hang around after it has been deleted, renamed or otherwise rendered invalid. The function is.active() does a quick check to make sure that the pointer to the object in the RNetica environment has not be set to NULL.

Note that unlike ordinary R objects, NeticaBN and NeticaNode objects only last as long as the RNetica environment lasts. In particular, if StopNetica() is called to close the RNetica environment, or the R session is exited (either cleanly or through a crash), then all of the NeticaBN and NeticaNode objects should become inactive. It is an error to execute RNetica functions with the old objects.

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.

For nodes, the best solution is to use a query function to return a list of the desired nodes, in particular, NetworkFindNode() or NetworkAllNodes(). If a particular subset of nodes should be loaded every time the network is loaded, then they can be placed in a node set, and the function NetworkNodesInSet() can be used to retrieve just the interesting nodes. All of these functions return a list of NeticaBN objects, which can be used to provide convenient access. For example, if net was previously saved and "Proficiency" is a node in net, then:

```
net <- ReadNetworks(net)
net.nodes <- NetworkAllNodes(net)
```

will load all of the nodes in net, and the expression net.nodes$Proficiency will access the "Proficiency" node.

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 provies 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 `[.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 thoses 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 raplidly calculate conditional probabilities. In Netica language, one enters "findings" (conditions) on the known or hypothesized variables and then calculates "beliefs" (conditional probalities) 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()`. The function `JointProbability()` calculates the joint distribution over a collection of nodes, and the function `FindingsProbability()` calculates the prior probability of the observed findings. The function `MostProbableConfig()` 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 indended 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 late alpha to early 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 possiblity 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 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.

Examples

```
############################################################
## Network Construction:

abc <- CreateNetwork("ABC")
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(paste(library(help="RNetica")$path,
                           "sampleNets","System.dne",
                           sep=.Platform$file.sep))

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

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

## 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))
```

`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 probalility 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

```
anet <- CreateNetwork("Absorbant")

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])),
  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)
```

AddLink *Adds or removes a link between two nodes in a Netican 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 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 decendent of child (that is if is.NodeRelated(child), "decendent", 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

```
abnet <- CreateNetwork("AABB")
A <- NewDiscreteNode(abnet, "A")
B <- NewDiscreteNode(abnet, "B")

AddLink(A,B)

stopifnot(
  match(A,NodeParents(B),nomatch=0)>0,
  match(B,NodeChildren(A),nomatch=0)>0
)

DeleteLink(A,B)

stopifnot(
  match(A,NodeParents(B),nomatch=0)==0,
  match(B,NodeChildren(A),nomatch=0)==0
)

DeleteNetwork(abnet)
```

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

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.

Details

This follows the System Model–Evidence Model 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 proceedure, 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

```
## System/Student model
EMSMSystem <- ReadNetworks(paste(library(help="RNetica")$path,
                           "sampleNets","System.dne",
                           sep=.Platform$file.sep))

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

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

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


## 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)

## This should be the same
```

```
stopifnot(
  sum(abs(student1.probt1a-student1.probt1ax)) <.0001
)

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

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

student1.probt1a2a <- JointProbability(student1.prof)

AbsorbNodes(student1.t2a)
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)

DeleteNetwork(student1)
DeleteNetwork(EMTask2b)
DeleteNetwork(EMSMSystem)
```

`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)
CaseFileMissingCode(newcode = NULL)
```

Arguments

newdelimiter	A character scalar containing the new delimiter. It must be either a comma, a space, or a tab.
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 mising 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

WriteFindingsToFile

Examples

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

defaultMiss <- CaseFileMissingCode() # Get default
m1 <- CaseFileMissingCode("*")
m2 <- CaseFileMissingCode("?")
m3 <- CaseFileMissingCode(" ")
m4 <- CaseFileMissingCode("")
## Not run:
 ## This should thow an error.
```

```
 CaseFileDelimiter(" ")

## End(Not run)

m5 <- CaseFileMissingCode("?")

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

## End(Not run)
## But this is okay
CaseFileMissingCode("*")

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

## restore defaults
CaseFileDelimiter(defaultDelim)
CaseFileMissingCode(defaultMiss)
```

`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 probabilty 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.

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

```
irt5 <- ReadNetworks(paste(library(help="RNetica")$path,
                           "sampleNets","IRT5.dne",
                           sep=.Platform$file.sep))
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)
```

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

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

copy1 <-CopyNetworks(net1,"Copy1")
stopifnot(is(copy1,"NeticaBN"))
stopifnot(as.character(copy1) == "Copy1")
stopifnot(copy1 != net1)

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

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

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 lenght 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

```
System <- ReadNetworks(paste(library(help="RNetica")$path,
                             "sampleNets","System.dne",
                             sep=.Platform$file.sep))

EMTask1a <- ReadNetworks(paste(library(help="RNetica")$path,
                             "sampleNets","EMTask1a.dne",
```

```
                          sep=.Platform$file.sep))

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))
```

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 CPF is a special data.frame object which represents a conditional probability table.

Usage

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

Arguments

x	Object to be tested or coerced into a CPF.

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 existance of this meta-data, and as.CPA() will raise a warning if an array without the appropriate dimnames is suppled.

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 proability 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 frunction as.CPA returns an object of class c("CPA","array"), which is essentailly 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.

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 proability table is a table in which the first severl 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 cas 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 proabillities 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 `"[.NeticaNode"` returns a `CPF` (if the node is not deterministic).

The function `as.CPF()` is designed to convert between `CPA`s (that is, conditional proability 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 frunction `as.CPF` returns an object of class `c("CPF","data.frame")`, which is essentailly 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 probabilty tables where the first parent variable varies fastest. The Netica GUI displays tables in which the last parent variable varies fastest.

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. `DeleteNetwork()` frees the memory associated with the named network inside of Netica.

Usage

```
CreateNetwork(names)
DeleteNetwork(nets)
```

Arguments

`names`	A character vector giving the name or names of the network to be created.
`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 link{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 suct 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

Currently, the NeticaBN object uses the name of the networks as the pointer into the network. Thus either a character vector of names or a list of NeticaBN objects is mostly equivalent.

Future versions may actually use pointers to the Netica objects.

Note

The function DeleteNetwork() implicitly deletes any nodes assocaited 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

CopyNetworks(), is.active()

Examples

```
net1 <- CreateNetwork("EmptyNet")
stopifnot(is(net1,"NeticaBN"))
stopifnot(as.character(net1)=="EmptyNet")
stopifnot(is.active(net1))

netd <- DeleteNetwork(net1)
```

```
stopifnot(!is.active(netd))
stopifnot(!is.active(net1))
stopifnot(as.character(netd)=="EmptyNet")
```

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

```
a1 <- CreateNetwork("AB1")
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)
```

`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 neighbors 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

```
EMSMMotif <- ReadNetworks(paste(library(help="RNetica")$path,
                           "sampleNets","EMSMMotif.dne",
                           sep=.Platform$file.sep))

## 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 varaibles 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)
```

`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 correspoinding 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(), EnterFindings(), RetractNodeFinding(), NodeLikelihood(), JointProbability(), MostProbableConfig(), FindingsProbability()

Examples

```
Motif <- ReadNetworks(paste(library(help="RNetica")$path,
                            "sampleNets","EMSMMotif.dne",
                            sep=.Platform$file.sep))

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)
```

EnterNegativeFinding *Sets findings for a Netaca 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 assers 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 virutal findings set through `NodeLikelihood()` or simplie findinding 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

```
irt5 <- ReadNetworks(paste(library(help="RNetica")$path,
                           "sampleNets","IRT5.dne",
                           sep=.Platform$file.sep))

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)
```

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

Description

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

Usage

```
## S3 method for class 'NeticaNode'
x[..., drop=FALSE]
## S3 method for class 'NeticaNode'
x[[...]]
## S3 replacement method for class 'NeticaNode'
x[...] <- value
EVERY_STATE
```

Arguments

`x`	An active, discrete `NeticaNode` whose conditional probability table is to be accessed.
`...`	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 scaler (discrete node) or numeric scaler (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 funciton allows the syntax node[...] to be used to access only a portion of the table. Ther 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 becasue 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 probabity 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 discritized 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 (IsNodeDetermistic(node)==TRUE), then the probilities 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`, wheree `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 varaibles 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 mechansims 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 Meanining 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 probabillity 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).

Netcia 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 programmers 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 probabililty 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 unrestriced probabilities, however, the deterministic rows will then print out as unconstrained probabilites 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 ofr have been discritized.

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 asigned 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.

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
xnet <- CreateNetwork("X")

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
  )
}

DeleteNetwork(xnet)
```

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 funciton is often used together with LearnFindings to downweight 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

```
aaa <- CreateNetwork("AAA")
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)
```

`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 findinds 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.

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,NodeBeliefs(),EnterNegativeFinding(),RetractNodeFinding(),NodeLikelihood()

Examples

```
EMSMMotif <- ReadNetworks(paste(library(help="RNetica")$path,
                          "sampleNets","EMSMMotif.dne",
                          sep=.Platform$file.sep))

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)
```

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

Description

This searches through the currently open Netica networks 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.

Usage

```
GetNamedNetworks(namelist)
```

Arguments

namelist	A character vector giving the name or names of the networks to be found.

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.

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

This function does a linear search through all networks, so it could be pretty slow if there are a large number of networks open.

Author(s)

Russell Almond

References

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

See Also

CreateNetwork(), GetNthNetwork()

Examples

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

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

DeleteNetwork(net2)
```

`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 expresssion to be evaluated with automatica 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 mechansim for turning updating off while performing a series of operations.

Value

`GetNetworkAutoUpdate()` and `SetNetworkAutoUpdate` both returns the current autoupdate flage 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

```
autoNet <- CreateNetwork("AutomaticTest")

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)
```

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)
```

Arguments

n	A vector of integers greater than 1.

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().

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

CreateNetwork(), GetNamedNetworks()

Examples

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

`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 probabilty table has been established, that is has `NodeProbs()` been set. The second value tests to see whether the conditional probabilty 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

```
ab1 <- CreateNetwork("AB1")
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 ", node, " still needs a conditional probability table.\n")
  }
}

DeleteNetwork(ab1)
```

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")
```

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.

Details

Netica identifiers (net names, node names, state names, and similar) are limited to 30 characters which must be a valid letter, number of the character '_'. The first chacter 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 30 characters in length.

Note that these rules guarentee that the result will conform to the IDname convention. They do not guarentee that the resulting values will be unique. In particular, watch out for long names that differ only in the last few characters.

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("yaFish","yWanda1","yfeed_me","yfish_food","y1more","yUS_",
                      "ya123456789012345678901234567O")
)
```

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

Description

Both `NeticaBN` and `NeticaNode` objects contain embedded pointers into Netica's memory. The function `is.active()` checks to see that the corresponding Netica objetct still exists.

Usage

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

Arguments

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

Details

Internally, both `NeticaBN` and `NeticaNode` objects contian 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.

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.

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 `NeticaBN` or `NeticaNode`.

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

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()

Examples

```
anet <- CreateNetwork("ActiveNet")
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)))
```

---

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

Description

A NeticaNode objet 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 vise 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 value representing the midpoint of the node.

Value

TRUE or FALSE depedning 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 attribute "node_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

```
netx <- CreateNetwork("netx")

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)
```

`is.NodeRelated` *Computes topological properities 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", "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 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`.

`"decendent"`, `"decendents"`. True if there is a directed (parent to child) path from `node2` to `node1`, or returns a list of decendents 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

```
testnet <- CreateNetwork("ABCDEFG")
###  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)
```

`IsNodeDeterministic` *Determines if a node in a Netica Network is deterministic or not.*

Description

A node in a Bayesian network is all of its conditional probabilities are determined by its parent states, that is they are all deterministic.

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

```
ab <- CreateNetwork("AB")
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)
```

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 caluclates 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 intserting 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-Kauphman

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

See Also

NeticaNode,NodeBeliefs() MakeCliqueNode(),AddLink(),JunctionTreeReport(),MostProbableConfig()

Examples

```
EMSMMotif <- ReadNetworks(paste(library(help="RNetica")$path,
                          "sampleNets","EMSMMotif.dne",
                          sep=.Platform$file.sep))
## 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)
```

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

```
EMSMMotif <- ReadNetworks(paste(library(help="RNetica")$path,
                          "sampleNets","EMSMMotif.dne",
                          sep=.Platform$file.sep))

CompileNetwork(EMSMMotif)

JunctionTreeReport(EMSMMotif)

NetworkCompiledSize(EMSMMotif)

DeleteNetwork(EMSMMotif)
```

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 funciton FadeCPT is often used between calls to LearnFindings to downweight 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 NodeFindings. NodeFindings 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.

Author(s)

Russell G. Almond

References

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

See Also

NodeExperience, NodeProbs, NodeFinding, FadeCPT, RetractNetFindings

Examples

```
abb <- CreateNetwork("ABB")
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)
```

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 (formally, the class is c("CliqueNode", "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" attiribute 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 newtwork, but the special "clique" attribute will be lost.

Currently Netica only allows virutal evidence at the node level (NodeLikelihood()). I'm loppying 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-Kauphman

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

See Also

NeticaNode, JointProbability(), AddLink(), JunctionTreeReport()

Examples

```
EMSMSystem <- ReadNetworks(paste(library(help="RNetica")$path,
                           "sampleNets","System.dne",
                           sep=.Platform$file.sep))
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)
```

`MostProbableConfig` *Finds the configuration of the nodes most likel 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 unnecssary 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 probabiity 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

```
EMSMMotif <- ReadNetworks(paste(library(help="RNetica")$path,
                          "sampleNets","EMSMMotif.dne",
                          sep=.Platform$file.sep))

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)
```

NeticaBN *An object referencing a Bayesian network in Netica.*

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)
## S3 method for class 'NeticaBN'
toString(x, ...)
## S3 method for class 'NeticaBN'
print(x, ...)
## S3 method for class 'NeticaBN'
Ops(e1, e2)
```

```
e1 == e2
e1 != e2
```

Arguments

x	The object to print or test
...	Other arguments to print() or toString()
e1	A NeticaBN object to test.
e2	A NeticaBN object to 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 dending on whether the objects are equal.

Note

Internally, the NeticaBN objects are character strings with extra attributes. So as.character(net) will return the name of the network.

Note that if a NeticaBN object is stored in an R object, and the network is subsquently 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()

Examples

```
net1 <- CreateNetwork("aNet")
stopifnot(is.NeticaBN(net1))
stopifnot(is.active(net1))
stopifnot(as.character(net1)=="aNet")

net2 <- GetNamedNetworks("aNet")
stopifnot(as.character(net2)=="aNet")
stopifnot(net1==net2)

NetworkName(net1) <- "Unused"
stopifnot(net1==net2)
## Warning:  The following expression is true!
as.character(net1) != as.character(net2)

netd <- DeleteNetwork(net1)
stopifnot(!is.active(net1))
stopifnot(!is.active(net2))
stopifnot(as.character(netd)=="Unused")
stopifnot(netd == net1)
## Warning:  The following expression is true!
net1 != net2
```

NeticaNode *An object referencing a node in a Netica Bayesian network.*

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 fucntion is.active() tests whether the node is still a valid reference.

Usage

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

```
## S3 method for class 'NeticaNode'
print(x, ...)
## S3 method for class 'NeticaNode'
Ops(e1, e2)
e1 == e2
e1 != e2
```

Arguments

x	The object to print or test
...	Other arguments to print() or toString()
e1	A NeticaNode object to test.
e2	A NeticaNode object to test, or a list of such objects.

Details

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. Not 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.

The function Ops.NeticaNode() returns a logical value dending on whether the objects are equal. If the second argument is a list of NeticaNode objects, then a logical vector is returned, testing e1 against every element of e2.

Note

Internally, the NeticaNode objects are character strings with extra attributes. So as.character(node) will return the name of the node.

Note that if a NeticaNode object is stored in an R object, and the Node is subsquently 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()

Examples

```
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)
```

NeticaVersion *Fetches the version number of Netica.*

Description

The version number of Netica is returned as both an integer and a string.

Usage

```
NeticaVersion()
```

Details

This must be called after the call to StartNetica().

Value

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

StartNetica()

Examples

```
print(NeticaVersion()$message)
stopifnot(NeticaVersion()$number > 409) ## Version 4.09 is a popular one.
```

NetworkFindNode *Finds nodes in a Netica network.*

Description

The funciton 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.

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 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

NodeNet() retrieves the network from the node.

Examples

```
tnet <- CreateNetwork("TestNet")
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(any(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)
tnet <- ReadNetworks(tnet)
nodes <- NetworkFindNodes(tnet,as.character(nodes))

## End(Not run)
DeleteNetwork(tnet)
```

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 one of the values of `NodeParents(node)` to `NULL`, or by copying a node (`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()` accorind 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 provies 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 Mislevey, 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

```
## System/Student model
EMSMSystem <- ReadNetworks(paste(library(help="RNetica")$path,
                           "sampleNets","System.dne",
                           sep=.Platform$file.sep))

CompileNetwork(EMSMSystem)
JunctionTreeReport(EMSMSystem)

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

```
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(paste(library(help="RNetica")$path,
                           "sampleNets","EMTask2a.dne",
                           sep=.Platform$file.sep))
NetworkFootprint(EMTask2a)
## This is already a clique, so nothing to do.

DeleteNetwork(list(EMSMSystem,EMTask1a,EMTask2a))
```

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)
NetworkName(net) <- value
```

Arguments

net	A NeticaBN object which links to the network.
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.

Value

The name of the network as a character vector of length 1.

The setter method returns the modified object.

Note

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.

Author(s)

Russell Almond

References

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

See Also

CreateNetwork(), NeticaBN, GetNamedNetworks(), NetworkTitle()

Examples

```
net <- CreateNetwork("funNet")
netcached <- net

stopifnot(NetworkName(net)=="funNet")

NetworkName(net)<-"SomethingElse"
stopifnot(as.character(net)=="SomethingElse")

##Warning, the following expression is true!
as.character(netcached) != NetworkName(netcached)
## But this one holds
stopifnot(NetworkName(net)==NetworkName(netcached))
## And this one
stopifnot(net==netcached)

## This fixes the problem
NetworkName(netcached) <- NetworkName(netcached)
stopifnot(as.character(netcached) == NetworkName(netcached))

DeleteNetwork(net)
```

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 priorty 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 color as hexidecimal 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

```
nsnet <- CreateNetwork("NodeSetExample")

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)
```

`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(node) returns the node sets that are currently associated with any node. 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

```
nsnet <- CreateNetwork("NodeSetExample")

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)
```

`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)
```

Arguments

`net`	An active `NeticaBN` object representing the network.
`setname`	A character scalar giving the node set to look for.

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. This value cannot be set directly, only indirectly through the use of NodeSets, or through the use of system primitives.

Note that it is acceptable to use the system built-ins. For examble 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

```
nsnet <- CreateNetwork("NodeSetExample")

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")

## setequal doesn't deal well with arbitrary objects, so
```

```
## just use the names.
nodeseteq <- function(x,y) {
  setequal(as.character(x),as.character(y))
}

stopifnot(
  nodeseteq(NetworkNodesInSet(nsnet,"ReportingVariable"),
            list(Ability,EssayScore)),
  nodeseteq(NetworkNodesInSet(nsnet,"Observable"),
            list(X1,EssayScore)),
  nodeseteq(NetworkNodesInSet(nsnet,"Observables"),
            list()),
  nodeseteq(NetworkNodesInSet(nsnet,":Nature"),
            list(Ability,EssayScore,X1)),
  nodeseteq(NetworkNodesInSet(nsnet,":Decision"),
            list(Placement)),
  nodeseteq(NetworkNodesInSet(nsnet,":Utility"),
            list(Value))
)

DeleteNetwork(nsnet)
```

`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 beging 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

```
nsnet <- CreateNetwork("NodeSetExample")

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)
```

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 freeform 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

```
firstNet <- CreateNetwork("firstNet")

NetworkTitle(firstNet) <- "My First Bayesian Network"
stopifnot(NetworkTitle(firstNet)=="My First Bayesian Network")

NetworkComment(firstNet)<-c("Network created on",date())
stopifnot(NetworkComment(firstNet) ==
  paste(c("Network created on",date()),collapse="\n"))

## Print here escapes the newline, so is harder to read
cat(NetworkComment(firstNet),"\n")

DeleteNetwork(firstNet)
```

`NetworkUndo` *Undoes (redoes) a Netica operation on a network.*

Description

Netica mantians an internal queue of reversable 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:
activeNet <- CreateNetwork("undoRedoTest")

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)

## 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 aribtrary string.

Usage

```
NetworkUserField(net, fieldname)
NetworkUserField(net, fieldname) <- value
NetworkAllUserFields(net)
```

Arguments

net	A NeticaBN object indicating the network.
fieldname	A character scalar conforming to the IDname rules.
value	An arbitrary character string containing the new value. Only the first element is used.

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. This meachnism can be used to store arbitrary values, but the user is responsible for encoding/decoding them as strings.

Value

A character string with the value stored in the field `fieldname`, or `NA` if no such field exists.

The function `NetworkAllUserFields` returns a character vector containing all user data stored with the network. The names of the result are the names of the fields.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html GetNetUserField_bn(), SetNetUserField_bn(), GetNetNthUserField_bn()

See Also

NeticaBN, NetworkComment()

Examples

```
userNet <- CreateNetwork("UserNet")
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")))

DeleteNetwork(userNet)
```

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() both 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 continus 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 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()).

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.

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.

Internally, node objects character vectors giving the name of the node with special properties representing the node. Thus, as.character(node) will usually return the name of the node. However, if the node is renamed, that might not work properly. Using NodeName(node) is generally safer.

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()

Examples

```
safetyNet <- CreateNetwork("safetyNet")

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)
```

`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 propogated 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).

Author(s)

Russell Almond

References

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

See Also

NeticaBN, NodeProbs(), NodeFinding(), JointProbability(), MostProbableConfig(), FindingsProbability()

Examples

```
irt5 <- ReadNetworks(paste(library(help="RNetica")$path,
                           "sampleNets","IRT5.dne",
                           sep=.Platform$file.sep))

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 updateing
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 updateing
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)
```

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 connnected 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 decendents of parent. A list of all decendents 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

```
chnet <- CreateNetwork("ChildcareCenter")

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)
```

`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 alterantive 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 Speigelhalter and Laurtizen (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

```
abc <- CreateNetwork("ABC")
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)
```

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 scaler 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 virutal 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"` — Uncertiain 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(), JointProbability(), MostProbableConfig(), FindingsProbability()

Examples

```
irt5 <- ReadNetworks(paste(library(help="RNetica")$path,
                           "sampleNets","IRT5.dne",
                           sep=.Platform$file.sep))

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 updateing
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 updateing
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)
```

`NodeInputNames` *Associates names with incomming 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 detacted, 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

```
abnet <- CreateNetwork("AB")

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)
```

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 detacted 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

```
knet <- CreateNetwork("kNet")

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"
## Detatch node
NodeParents(pretestScore)$SkillTested <- list(NULL)
stopifnot(
  NodeKind(NodeParents(pretestScore)$SkillTested) == "Stub"
)

DeleteNetwork(knet)
```

`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 increaseing 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 compatability.

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

```
lnet <- CreateNetwork("LeveledNet")

## 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)
```

NodeLikelihood *Returns or sets the virtual evidence associated with a Netica node.*

Description

The findings associated with a node can be exrpressed as the probability of the evidence occuring 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 representes the conditional probability $Pr(e|node == state)$. Note that the likelihood can be throught 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 (or else the evidence represtents an impossible case) or Netic will generate an error.

Entering findings through NodeFinding(node,state) sets a special likelihood. In this case, the likelihood value corresponding to state will be 1, and all others will be 0. Similarly, EnterNegativeFinding(node,statel 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 accont 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 evidnece. 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 "NodFinding<-" 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

```
irt5 <- ReadNetworks(paste(library(help="RNetica")$path,
                           "sampleNets","IRT5.dne",
                           sep=.Platform$file.sep))

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)
```

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)
NodeName(node)<- value
```

Arguments

node	An active NeticaNode object that references the node.
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.

Value

The name of the node as a character vector of length 1.

The setter method returns the NeticaNode object

Note

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.

Author(s)

Russell Almond

References

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

See Also

NewDiscreteNode(), NeticaNode, NetworkFindNode(), NodeTitle()

Examples

```
net <- CreateNetwork("funNet")

pnode <- NewDiscreteNode(net,"play")
nodecached <- pnode

stopifnot(NodeName(pnode)=="play")

NodeName(pnode)<-"work"
stopifnot(as.character(pnode)=="work")
```

```
##Warning, the following expression is true!
as.character(nodecached) != NodeName(nodecached)
## But this one holds
stopifnot(NodeName(pnode)==NodeName(nodecached))
## And this one
stopifnot(pnode==nodecached)

## This fixes the problem
NodeName(nodecached) <- NodeName(nodecached)
stopifnot(as.character(nodecached) == NodeName(nodecached))

snode <- NewContinuousNode(net,"sleep")
NodeName(snode) <- "work" ## This should issue a warning
## And not change the name.
stopifnot(NodeName(snode)=="sleep")

allNodes <- NetworkAllNodes(net)
NodeName(allNodes$work) <- "effort"


DeleteNetwork(net)
```

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)
```

Arguments

node	A NeticaNode object.

Details

Two nodes with the same details in different networks are not identical inside of Netica.

This function only works for active nodes. If is.active(node) returns false, this function will return NULL.

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.

Author(s)

Russell Almond

References

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

See Also

NeticaBN, NeticaNode, is.active(), NetworkAllNodes(), NetworkFindNode()

Examples

```
neta <- CreateNetwork("Net_A")
netb <- CreateNetwork("Net_B")

nodea <- NewContinuousNode(neta,"Node")
nodeb <- NewContinuousNode(netb,"Node")

stopifnot(NodeNet(nodea)==neta)
stopifnot(NodeNet(nodeb)==netb)

## Note
stopifnot(nodea != nodeb)
## But:
stopifnot(as.character(nodea) == as.character(nodeb))

DeleteNodes(nodeb)
stopifnot(is.null(NodeNet(nodeb)))

DeleteNodes(nodea)

## Now:
stopifnot(nodea == nodeb)

DeleteNetwork(list(neta,netb))
```

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 diffent 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 decendents 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

```
abnet <- CreateNetwork("AB")

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)
```

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

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

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

Examples

```
abc <- CreateNetwork("ABC")
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)
```

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
```

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.

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.

Value

A character vector giving the names of the node sets node is associated with. The setter form returns 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

```
nsnet <- CreateNetwork("NodeSetExample")

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
)

DeleteNetwork(nsnet)
```

`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) <- 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.

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()`). The number of states may not be changed, but they can be renamed.

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).

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()

See Also

`NewDiscreteNode()`, `NeticaNode`, `NodeName()`, `is.discrete()`, `is.active()`, `NodeStateTitles()`, `NodeLevels()`, `NodeStateComments()`,

Examples

```
anet <- CreateNetwork("Annette")

## 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"
)

## 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(vnode) <- 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)
```

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 coments 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

```
cnet <- CreateNetwork("CreativeNet")

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)
```

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 freeform 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

```
net2 <- CreateNetwork("secondNet")

firstNode <- NewDiscreteNode(net2,"firstNode")

NodeTitle(firstNode) <- "My First Bayesian Network Node"
stopifnot(NodeTitle(firstNode)=="My First Bayesian Network Node")

NodeDescription(firstNode)<-c("Node created on",date())
stopifnot(NodeDescription(firstNode) ==
  paste(c("Node created on",date()),collapse="\n"))

## Print here escapes the newline, so is harder to read
cat(NodeDescription(firstNode),"\n")

DeleteNetwork(net2)
```

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 aribtrary string.

Usage

```
NodeUserField(node, fieldname)
NodeUserField(node, fieldname) <- value
NodeAllUserFields(node)
```

Arguments

node	A NeticaBN object indicating the node.
fieldname	A character scalar conforming to the IDname rules.
value	An arbitrary character string containing the new value. Only the first element is used.

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. This meachnism can be used to store arbitrary values, but the user is responsible for encoding/decoding them as strings.

Value

A character string with the value stored in the field fieldname, or NA if no such field exists.

The function NodeAllUserFields returns a character vector containing all user data stored with the node. The names of the result are the names of the fields.

Author(s)

Russell Almond

References

http://norsys.com/onLineAPIManual/index.html GetNodeUserField_bn(), SetNodeUserField_bn(), GetNodeNthUserField_bn()

See Also

NeticaBN, NodeDescription()

Examples

```
usedNet <- CreateNetwork("UsedNet")

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")))

DeleteNetwork(usedNet)
```

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 appearence when the node 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

```
pnet <- CreateNetwork("PositionNet")

pnode <- NewDiscreteNode(pnet,"PlaceMe")

NodeVisPos(pnode) <- c(100,300)
pos <- NodeVisPos(pnode)
stopifnot(
  pos["x"] ==100,
  pos["y"] ==300
)

## Netica rounds nonitenger 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)
```

NodeVisStyle *Gets/sets the nodes visual appearence in Netica.*

Description

Netica internally has a number of styles it can use to draw a node, thes 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 appearence 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 occured.

The setter method returns the modified node object.

Note

The Netica documentation indicates that in the future additionl 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

```
snet <- CreateNetwork("StylishNet")

snode <- NewDiscreteNode(snet,"StyleMe")
stopifnot(NodeVisStyle(snode)=="Default")

NodeVisStyle(snode) <- "Meter"
stopifnot(NodeVisStyle(snode)=="Meter")

DeleteNetwork(snet)
```

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 varaible. 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 properities 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 assiged 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 correponds to the states of a parent variable. If node has no parents, it returns a list of length 0.

The function ParentName(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

```
abc1 <- CreateNetwork("ABC1")
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)
```

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, NodeBeliefs(), EnterNegativeFinding(), NodeFinding(), NodeLikelihood()

Examples

```
irt5 <- ReadNetworks(paste(library(help="RNetica")$path,
                           "sampleNets","IRT5.dne",
                           sep=.Platform$file.sep))

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)
```

`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 probality 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 connecte 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

```
abcnet <- CreateNetwork("ABC")

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)
```

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

```
StartNetica(license = LicenseKey, checking = NULL, maxmem = NULL)
StopNetica()
```

Arguments

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 NeticaLicenceKey is set before RNetica is loaded, then the value of that variable at the time the package is loaded will become the deafult 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 function StartNetica() calls the Netica functions NewNeticaEnviron_ns() and InitNetica2_bn() to create and set up a Netica environment.

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 verison is used instead of the licensed version (could be useful for testing).

If you set the value of a variable NeticaLicenseKey, then when RNetica is loaded, then its value at the time the package is loaded is used as the default value for license. If no value for as NeticaLicenseKey, the default value for license is set to NULL, which loads the demo version of Netica.

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.

The function `StartNetica` is called when the package is attached (in the `.onAttach()` function. The function `StopNetica()` is called by `.Last.lib()`, Normally, users should not need to call these functions, but they may wish to do so if they need to restart Netica with different arguments.

Value

These functions are called for side effects and do not return meaningful values.

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

I'm looking into a way to burry the license key into RNetica during the installation process. Probably will happen in a future version. Until then, best bet is to save a value for `NeticaLicenseKey` in the workspace.

The Netica environment pointer, which is used by the Netica shared library is defined inside of the RNetica shared library, and not visible at the R level.

Author(s)

Russell Almond

References

`http://norsys.com/onLineAPIManual/index.html`: NewNeticaEnviron_ns(), InitNetica2_bn(), CloseNetica_bn(), LimitMemoryUsage_ns(), ArgumentChecking_ns()

See Also

`NeticaVersion()`, `CreateNetwork()`

Examples

```
## Not run:
## Restart licensed version
StopNetica()
StartNetica("License key from Norsys")
## Get the version of Netica.
print(NeticaVersion())

## Commonly done next step is to create a network.
net1 <- CreateNetwork("myNet")


## End(Not run)
```

`WriteFindingsToFile` *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

```
WriteFindingsToFile(nodes, filename, id = -1, freq = -1)
```

Arguments

`nodes`	The a list of active `NeticaNode` objects to be written out.
`filename`	A character scaler giving the path name of the file to which the results are to be written. 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. `WriteFindingsToFile` 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 `id` should contain an integer case number. The column `freq` 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 field does not get appended to the output.

The function `WriteFindingsToFile` 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 `WriteFindingsToFile` 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 `NULL` invisibly.

Author(s)

Russell G. Almond

References

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

See Also

CaseFileDelimiter, CaseFileMissingCode, NodeFinding, RetractNetFindings

Examples

```
abc <- CreateNetwork("ABC")
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
outdir <- ifelse(.Platform$OS.type == "windows", "C:/temp","/tmp")
casefile <- paste(outdir,"testcases.cas",sep=.Platform$file.sep)
if (file.exists(casefile)) file.remove(casefile)

## Case 1
NodeFinding(A) <- "A1"
NodeFinding(B) <- "B1"
NodeFinding(C) <- "C1"
WriteFindingsToFile(list(A,B,C),casefile,1)
RetractNetFindings(abc)

## Case 2
```

```
NodeFinding(A) <- "A2"
NodeFinding(B) <- "B2"
NodeFinding(C) <- "C2"
WriteFindingsToFile(list(A,B,C),casefile,2)
RetractNetFindings(abc)

## Case 3
NodeFinding(A) <- "A3"
NodeFinding(B) <- "B3"
## C will be missing
WriteFindingsToFile(list(A,B,C),casefile,3)
RetractNetFindings(abc)


DeleteNetwork(abc)
```

`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)
GetNetworkFileName(net)
```

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.

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, elments 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.

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) will not work after quitting and restarting Netica, but attr(net,"Filename") 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) right after the call to library(RNetica), 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 "Filename" attribute 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.

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

```
peanut <- CreateNetwork("peanut")
NetworkTitle(peanut) <- "The Peanut Network"
peanutFile <- tempfile("peanut",fileext=".dne")
WriteNetworks(peanut,peanutFile)
stopifnot(GetNetworkFileName(peanut)==peanutFile)
```

```
pecan <- CreateNetwork("pecan")
NetworkTitle(pecan) <- "The Pecan Network"
pecanFile <- tempfile("pecan",fileext=".dne")
almond <- CreateNetwork("almond")
NetworkTitle(almond) <- "The Almond Network"
almondFile <- tempfile("almond",fileext=".neta")
WriteNetworks(list(pecan,almond),c(pecanFile,almondFile))

DeleteNetwork(peanut)
DeleteNetwork(pecan)
DeleteNetwork(almond)
stopifnot(!is.active(almond))

peanut <- ReadNetworks(peanutFile)
stopifnot(is.active(peanut))
stopifnot(NetworkTitle(peanut)=="The Peanut Network")

nets <- ReadNetworks(c(pecanFile,almondFile))
stopifnot(length(nets)==2)
stopifnot(all(sapply(nets,is.active)))
stopifnot(NetworkTitle(nets[[1]])=="The Pecan Network")
almond <- GetNamedNetworks("almond")
stopifnot(is.NeticaBN(almond),is.active(almond))

DeleteNetwork(peanut)
DeleteNetwork(nets[[1]])
DeleteNetwork(almond)

## Not run:
## Safe way to preserve node and network objects across R sessions.
tnet <- WriteNetworks(tnet,"Tnet.neta")
q(save="yes")
# R
library(RNetica)
tnet <- ReadNetworks(tnet)
nodes <- NetworkFindNodes(tnet,as.character(nodes))

## End(Not run)
```

Index