Netica-J Manual

Version 3.25 and Higher

Java Version of Netica API

Norsys Software Corp

Netica-J Reference Manual
Version 3.25
July 23, 2009

Published by:

Norsys Software Corp.
3512 West 23rd Avenue
Vancouver, BC,
CANADA
V6S 1K5
www.norsys.com

Contents

1 Introduction

This reference manual is for Netica-J, the Java version of the Netica API Programmer's Library. It is meant to be used in conjunction with the onscreen Netica-J javadocs reference (see below). Netica-J is a set of Java classes and an accompanying Java Native Interface (JNI) library that allow a Java developer to use the Netica API Programmer's Library for working with Bayesian networks.

This manual is not a manual for Netica Application, which is an easy to use point-and-click application program with much of the same functionality (see http://www.norsys.com/netica.html). Users of the API will typically want to have the Application handy for visually inspecting and modifying nets. A version of Netica soon to be released will allow Netica API to use the GUI of Netica Application.

Besides Java, other versions of Netica API exist for C/C++, C# and Visual Basic each offering the full Netica functionality. Visit http://www.norsys.com/netica_api.html to learn more about the other members of the Netica API family, and to obtain their documentation. The C version can be used by programs written in any language which can call C functions, such as C++, Python, Perl, Prolog, Lisp, Matlab, Delphi Pascal, Fortran or Cobol). Interface files for some of these languages, developed by the Netica community, are available from Norsys. Matlab is supported through this, the Java API.

This manual assumes that you are familiar with the Java programming language. It also assumes familiarity with Bayesian networks or influence diagrams, although it has a little introductory material, especially on issues that are new or generally not well understood. Questions and comments about material in this manual may be sent to: netica-j@norsys.com.

1.1 Netica-Java API

The Netica-J API is a complete library of Java classes for working with Bayesian networks (also known as Bayes nets, belief networks, graphical models or probabilistic causal models) and influence diagrams (also known as decision networks). It contains functions to build, learn from data, modify, transform, performance-test, save and read nets, as well as a powerful inference engine. It can manage "cases" and sets of cases, and can connect directly with most database software. Bayes nets can be used for diagnosis, prediction, classification, sensor fusion, risk analysis, decision analysis, combining uncertain information and numerous probabilistic inference tasks.

Programs that use Netica-J completely control it. For example, Netica functions will not take any action until called, Netica will not do any I/O unless requested to, and its functions will not take an unpredictable amount of time or memory before returning. Netica-J is threadsafe in multi-threaded environments. It may be used in conjunction with other Java or JNI C libraries and it won't interfere with them.

Versions of Netica-J are available for MS Windows, Linux, and Macintosh (and for many other platforms from cell phones to mainframes - contact us for info), and each of these has an identical interface, so you can move your code between these platforms without changing anything to do with the Netica API. For the latest versions for the more common platforms, visit http://www.norsys.com/download_api.html

Before releasing any new version of the Netica API, every function is put through rigorous quality assurance testing to make sure it operates as designed. Hundreds of real nets and millions of random nets are generated and solved in multiple ways to check the inference results. This level of QA, combined with a careful initial design and over ten years of extensive customer usage, has resulted in a rock-solid product.

The Netica API has been designed to be easily extended in the future without changing what already exists. Many new features are currently under development, and it will continue to be extended for years to come.

Netica API features:

- **Dynamic Construction**: Can build and modify networks "on the fly" in memory (to support working with dynamic Bayes nets), and can save/read them to file.
- **Equations:** Probability tables may be conveniently expressed by equations, using a Java/C type syntax and taking advantage of an extensive library of built-in functions, including all the standard math functions and common probability distributions, as well as some functions and distributions specially suited to Bayes nets, such as noisy-or, noisy-max, noisy-sum, etc.
- **Learning from Data:** Probabilistic relations can be learned from case data, even while the net is being used for probabilistic inference. Learning from data can be combined with manual construction of tables and representation by equations. It can handle missing data and latent variables or hidden nodes. Learning algorithms include: counting, sequential updating, fractional updating, EM (expectation maximization), and gradient descent.
- **Database Connectivity:** Allows direct connection to most database software.
- **Threadsafe:** Can be used safely in multi-threaded environments.
- **Encryption:** Can save and read nets to file in encrypted form, which allows deploying solutions relying on Bayes nets kept private to an organization.

- **Sensitivity:** Netica can efficiently measure the degree to which findings at any node can influence the beliefs at another node, given the findings currently entered. The measures can be in the form of mutual information (entropy reduction), or the expected reduction of real variance.
- **Advanced Decision Nets:** Can solve influence diagrams which have multiple utility and decision nodes to find optimal decisions and conditional plans, using a junction tree algorithm for speed. Handles multi-stage decision problems, where later decisions depend on the outcomes of earlier ones, and on observations not initially known. No-forgetting links need not be explicitly specified.
- **Junction Tree Algorithm:** Can compile Bayes nets and influence diagrams into a junction tree of cliques for fast probabilistic inference. An elimination order can be specified or Netica can determine one automatically, and Netica can report on the resulting junction tree.
- **Soft Evidence:** Accepts likelihood findings (i.e., “virtual evidence”), findings of the form that some variable is not is some state, Gaussian findings, and interval findings, as well as regular real-valued or state findings.
- **Link Reversal:** Can reverse specified links or "sum out" (absorb) nodes of a Bayes net or influence diagram while maintaining the same overall joint probability distribution, properly accounting for any findings in the removed nodes or other nodes.
- **Disconnected Links:** Links may be individually named and disconnected from parent or child nodes, thus making possible libraries of network fragments (which you may then copy and connect to other networks or node configurations).
- **Case Support:** Can save individual cases (i.e. sets of findings) to file, and manipulate files of cases. Works with the UVF file format, which allows cases to be incomplete or have uncertain values (Gaussian, interval, sets of possibilities, sets of impossibilities, etc.), and associates an ID number and multiplicity with each case.
- **Simulation:** Can do sampling (i.e. stochastic simulation) to generate random cases with a probability distribution matching the Bayes net. Can use a junction tree algorithm for speed, or do direct sampling for nets too large to generate CPTs or a junction tree.
- **User Data:** Every node and network can store by name arbitrary data fields defined by you. They may contain numbers, strings, byte data, etc., and are saved to file when the object in question is being saved. As well, there are fields not saved to file, which can contain a pointer to anything you wish.
- **Error Handling:** Has a simple but powerful method for handling usage errors, which can generate very detailed error messages if desired.
- **Argument Checking:** Allows programmers to control how carefully API functions check their arguments when they are called, including a “development mode” to extensively check everything passed to an API function.
- **Compatibility:** Can work hand-in-hand with the Netica Application standalone product (for example, sharing the same files), and with Netica API versions for other languages.
- **Efficient:** Is optimized for speed, and is not too large (2 MB typical).
- **Many Platforms:** Is available for a wide range of platforms including MS Windows (95/NT to Vista), Linux, Macintosh, AIX, etc. Contact Norsys for other platforms.

• **Memory Limiting:** You can set a bound on how much total heap space Netica-J API is allowed to allocate for large tables, thereby preventing virtual memory thrashing or the memory-starving of other parts of your application.

• **Java Oriented Features:**

- Clean object-oriented design
- Comprehensive javadocs and manual
- Sample java source applications to get you started
- Uses Java's exception handling mechanism in the natural way
- Supports event listening by any Java object for events such as the creation, deletion, duplication, etc. of Nets or Nodes
- Supports user data fields for any Serializable Java object
- Supports standard Java I/O streams
- Supplies graphical visualization of Bayes nets with AWT/SWING classes

• **More Features:** A more extensive list of features is available from: http://www.norsys.com/netica_api.html

1.2 License Agreement and Password

Before using Netica API, make sure you accept the license agreement that is included in this package as the file **License Agreement.pdf.**

If you have purchased a license to use Netica API, you will have received a license password by email, on the invoice, and/or on the shipped disk. You pass the license password to the `Environ` constructor. For example:

```
   Environ env = new Environ("your unique license");
```

If you do not have a license password, then you can simply supply `null` in place of it, in which case Netica-J API will be fully functional, but limited in problem size (e.g. size of nets, size of data sets).

The license password you have purchased also licenses you to use versions of Netica API for other languages, such as the C version (Netica-C), the C# or Visual Basic version, or the C++ version. Simply supply that license string to the appropriate Environment constructor in those languages. The same rights and obligations granted by the API license apply to all the language versions.

If your license password enables Netica API, it will have a "310-" within it. The digit immediately following that is the version number of the license. It must be at least 3 to fully enable this version (3.xx) of Netica API. If it is less, then after you call `new Environ()`, a warning message will be available for viewing if you call `NeticaError.getWarnings(),` and Netica API will continue operation in limited mode. To upgrade your license, contact Norsys, or see: https://www.norsys.com/order_v3_upgrade.htm.

1.3 Files Included

The following files are included in the distribution of Netica-J, the Java version of Netica API:

Directory	**File**	**Description**
docs	• NeticaJ_Man.pdf	• the file for this document
	• javadocs/	• the javadocs directory for Netica-J
	• License Agreement.pdf	• a legal document relating to the use of Netica API
bin	• NeticaJ.jar	• the Java class library that defines Netica-J
	• NeticaJ.dll	• the Java-to-Native interface library (Windows only)
	(libNeticaJ.so)	" " " (Unix/Linux only)
	(libNeticaJ.jnilib)	" " " (MacOSX only)
	• Netica.dll	• the native Netica API library (Windows only)
	(libnetica.a)	" " " (MacOSX only)
src/neticaEx/	• NetEx.java	• A class containing useful Net methods
	• NodeEx.java	• " " Node "
	• NodeListEx.java	• " " NodeList "
src/neticaEx/ aliases	• Net.java	• A convenience class that renames NodeEx as Node
	• Node.java	• " " " NetEx as Net
	• NodeList.java	• " " " NodeListEx as NodeList
demo	• Demo.java	• a sample application to test your Netica-J installation
	• compile.bat (.sh)	• a sample batch file for compiling Demo.java (.bat for Windows, .sh for Unix/Linux/MacOSX)
	• run.bat (.sh)	• a sample batch file for running Demo.class (.bat for Windows, .sh for Unix/Linux/MacOSX)
examples	• BuildNet.java	• demonstrates building a Bayes net from scratch
	• DoInference.java	• demonstrates doing inference
	• SimulateCases.java	• demonstrates creating case instances that statistically derive from a given net
	• LearnCPTs.java	• demonstrates learning from cases
	• LearnLatent.java	• demonstrates EM Learning
	• ClassifyData.java	• demonstrates Naive Bayesian Classification of real-world medical data
	• MakeDecision.java	• demonstrates building a decision net and choose an optimal decision with it
	• DrawNet.java	• demonstrates use of the gui package for drawing nets
	• NetViewer.java	• demonstrates use of the gui package for editing nets and their findings
	• TestNet.java	• demonstrates testing the performance of a learned net with the net tester tool
	• compile.bat (.sh)	• a sample batch file for compiling all the java files in this directory
	• run.bat (.sh)	• a sample batch file for running all the java programs in this directory, after they have been compiled
examples/ Data Files	• ChestClinic.dne	• an example net file required by SimulateCases/LearnCPTs/TestNet.java
	• BreastCancer.dne	• an example net file required by ClassifyData.java
	• ChestClinic.cas	• a case file created by SimulateCases.java and required by TestNet.java
	•ChestClinic_WithVisuals.dne	• ChestClinic.dne but including all the size/position/color display information
	• LearnLatent.cas	• a case file required by LearnLatent.java
	• BreastCancer.cas	• a case file required by ClassifyData.java

The Netica-J directory structure

The **docs/** directory contains manuals, javadocs, license agreements, and any other documentation.
The **bin/** directory contains the Netica-J runtime software without which Netica-J will not function.
The **src/** directory contains source software that is distributed with Netica-J. You are free to examine, compile, or copy from these source files. We suggest that you leave the original files unmodified. These functions may change in future version of Netica.
The **demo/** directory contains a simple program that should be compiled and run after installation to establish that your Netica-J system is correctly installed and ready to use.
The **examples/** directory contains assorted sample data and source code that you may examine, copy, and edit freely.

1.4 Getting Started

Recommended Installation steps:

1. A Java-2 platform is required. There are many suppliers, for example SUN Microsystems at http://java.sun.com/products/. Version 3.25 was constructed using Java 1.4.2 and should be compatible with any 1.4 and higher platform.

2. Download Netica-J from the Norsys website: http://www.norsys.com/netica-j.html (older versions can be found at http://www.norsys.com/downloads/old_versions). Choose a version that matches your OS/platform.

3. Unzip it, and it will form a directory called NeticaJ_325 (or the current version number).

4. Test your installation with the Demo application provided:

   a) Change to the **demo/** directory and at the command line, type: `compile.bat` (`compile.sh` on Unix/Linux/MacOSX). Or click on the compile.bat icon. This will compile Demo.java and create Demo.class.

   b) At the command line, type: `run.bat` (`run.sh`). Or click on the run.bat icon. This will run Demo.class.

   c) If it displays a welcome message, and does simple probabilistic inference without declaring any errors, then your installation was successful.

5. Now that you have the example program running, you can duplicate the **demo/** directory, replace Demo.java with your own source files, and you are ready to build your own application. Don't forget to replace "`null`" in "`new Environ(null)`" with your own license password, if you want to have the full functionality of Netica.

6. Demo.java, is a good starting point for developing your own applications. You may wish to "cut-and-paste" from it. Similar examples showing how to build a net from scratch, do inference, generate cases, and learn from cases are provided in the **examples/** directory.

7. If you are familiar with the Hugin or JavaBayes systems and would like information on equivalent Netica functions, contact Norsys.

1.4 Complete Javadocs Reference

For javadocs-style documentation for Netica-J, simply point your browser at the **index.html** file in the **docs/javadocs/** directory. The javadocs very thoroughly document every class and every function of the Netica API. You will find it an invaluable companion during development.

1.5 IDE Installation

Using Java IDEs (Eclipse, JBuilder, NetBeans, JDeveloper, Forte, etc.)

You must inform your IDE of the locations of the three library files: **NeticaJ.dll** (**libNeticaJ.so**), **NeticaJ.jar**, and **Netica.dll** (**libnetica.a**). Assuming Netica-J was installed at the following location on your filesystem:

Windows:	`C:\NeticaJ_325`
Unix/Linux/MacOSX:	`/home/NeticaJ_325`

1) NeticaJ.dll(libNeticaJ.so/.jnilib) must appear on the java library path. Typically this is done with a -D option to the JVM. For example:

Windows:	`java -Djava.library.path=C:\NeticaJ_325\bin`
Unix/Linux/MacOSX:	`java -Djava.library.path=/home/NeticaJ_325/bin`

2) NeticaJ.jar must appear on the java CLASSPATH. For example:

Windows:	`java -classpath C:\NeticaJ_325\bin\NeticaJ.jar`
Unix/Linux/MacOSX:	`java -classpath /home/NeticaJ_325/bin/NeticaJ.jar`

3) **Windows Only**: Netica.dll must appear on the Windows execution "path, so that Windows can find it. For example:

Windows:	`set PATH=C:\NeticaJ_325\bin;%PATH%`

Eclipse instructions:

1. Create your Java project as usual

2. In the "Project Properties" dialog, choose the "Java Build Path" link, then click on the "Libraries¨ tab, and then click on the "Add External Jars" link. Navigate to the C:\NeticaJ_325\bin directory and select NeticaJ.jar

3. In the "Run As" dialog, go to the "Arguments" tab and in the "VM Arguments" window create the following argument: -Djava.library.path=C:\NeticaJ_325\bin

4. Windows Only: Still in the "Run As" dialog, go to the "Environment" tab and create a new "PATH" variable with value: C:\NeticaJ_325\bin;%PATH%

1.6 Upgrades, Support and Mailing List

New versions of Netica API are available for download from the Norsys website (from the “Downloads” menu at www.norsys.com). If you are using a license password, it will work with any version released within a year of the password being issued (and often longer).

If you would like to be notified of version updates and other news regarding Netica-J, please visit https://www.norsys.com/mailing_list.html?interests=Netica-J and supply us with your e-mail address. Mailings are infrequent, and your privacy will be respected.

We at Norsys have worked hard to make Netica-J a very high quality and robust package that is easy and natural to use. If you have any ideas for how it can be improved, we would be very happy to hear them. Please send your suggestions to: netica-j@norsys.com

1.7 Other Resources

The following resources at the Norsys website may be helpful when using Netica API:

Netica Application - This program has an easy-to-use graphical interface, and most developers working with Netica API use it to visualize and/or edit the Bayes nets they are working with. It is also useful for experimentation, and trying out concepts that are to be implemented using Netica API, since it operates in much the same way.

Website location: http://www.norsys.com/netica.html

Resources Page - Describes training, consulting, literature and websites available for Netica.

Website location: http://www.norsys.com/resources.htm

Bayes Net Library - A website containing many example Netica files that are ready to download into Netica (Application or API). They are Bayes nets and decision nets that have become classics in the literature, or are contributed by other Netica users. This is a good place to look for inspiration and ideas.

Website location: http://www.norsys.com/net_library.htm

DNET File Format - Describes the file format for Netica DNET files (which have file extension .dne or .dnet).

Website location: http://www.norsys.com/dl/DNET_File_Format.txt

2 Netica-J Package Design and Usage

This section outlines programming principles and issues as they relate to Netica-J's operation and organization. If you are an experienced Java developer or you are planning a sizeable development effort with Netica-J, you will definitely want to read and understand this section before beginning your development.

2.1 The “Ex” classes NetEx, NodeEx, and NodeListEx

The “Ex” classes inherit from their parent class (NodeEx extends Node, NetEx extends Net, etc).

They are built on top of the core Netica system to provide convenience of use (the “Ex” stands for “Extra”, “Example”, “External”, “Experimental”, and “Excellent!”). These are utilities and shortcuts that were deemed useful, but not basic enough to belong in the base class. Some of them are “Ex” methods because they are more useful in source code form, so that you can customize them to your needs.

Because their Java source is included, the “Ex” classes are a good place to look for coding examples. Indeed, many of the coding examples found in the javadocs are taken from the “Ex” classes.

Unlike the core Netica system, the “Ex” classes may change in future versions; methods may be added, removed or modified. For this reason, you may want to keep copies of the Ex classes for future reference, or you may want to copy out any methods you need to form your own extensions of the parent classes.

Since the “Ex” classes contain so many useful methods, many users will want to use the “Ex” classes in place of the more basic parent classes. See Section 3, Inheritance, below, for considerations when doing this.

The “Ex” classes are in part supported by the Netica-J user community, so please feel welcome to submit additional methods that you have found useful, or to suggest improvements to the ones already there.

Some of the “Ex” class methods are static, while others are not. The basic criterion of choosing to make a method static was whether that method could be thought of as a “standalone-utility” that would be useful to have around even when you didn’t have an “Ex” object present. Since none of the “Ex” classes define new state data, it is a trivial exercise to convert a static method to be non-static or vice versa, should you prefer the alternate.

Because the “Ex” classes are so useful, many developers will want to use them directly. To make this easy, their compiled classes have been included in the NeticaJ.jar distribution. All you need do is `import norsys.neticaEx.*;` and you are ready to use them without the need to compile your own versions of them.

Finally, as a convenience, we also supply in the **norsys.neticaEx.aliases** package, three wrapper classes for NetEx, NodeEx, and NodeListEx, that are named Net, Node, and NodeList, respectively. They allow you to use the base class names and still use the Ex classes. See **demo/Demo.java** and **examples/BuildNet.java** for examples of how to use these convenience classes.

2.2 Inheritance of the Node and Net classes

Advanced users will want to create their own specialized Node and Net classes. To make this task easier, and avoid the need for copy constructors, we have supplied you with a means to inform Netica-J what class you would like it to use when constructing a Net or Node (for example, when Netica-J is reading a net in from a file). The static methods:

```
Net.setConstructorClass (String className)    and
Node.setConstructorClass (String className)
```

have been supplied for this purpose. All they require is that your Net or Node extension have a default constructor. See their javadocs pages for examples.

Some users will want to use the words “Net” and “Node” for their own net and node classes, that inherit from `norsys.netica.Net` and `norsys.netica.Node`, respectively. The supplied files in **src/neticaEx/aliases/** have examples of this. Although, overloading the terms “Net” and “Node” like this is not difficult, namespace conflicts may arise. In general, if you explicitly import your Node or Net class, the Java compiler will use those as the default classes.

2.3 Multithreading

If you are running Netica-J within a single process and are not creating more than one thread in that process, you don’t need to consider this issue. However, if you are operating in a concurrent usage environment, then you need to consider threading issues. Netica-J is threadsafe, in that if one thread calls a Netica-J function, and while it is executing another thread calls a Netica-J function, the new call will not interfere, even if they are both trying to operate on the same object (the new call will execute after the original is done). Of course, your software must do its own appropriate synchronization, and consider race possibilities, if you have more than one thread working on the same object (such as net or node) at the same time. Threads operating on separate nets will not have any interference.

For efficiency reasons, you may want to consider the following: Many Netica-J functions will block other Netica-J functions until they return. This is an efficiency concern only, and not a deadlock concern, since the executing Netica-J function will not be waiting on any other thread (unless you do that yourself through the use of Netica-J callbacks).

2.4 Event Handling

If you wish your program to receive events, Netica-J has the ability to call your program when certain types of events occur.

Any Java object can choose to listen to Netica events by simply implementing the NeticaEventListener interface and asking the node or net that generates the events to add itself to that node or net's listener list. The methods `Node.addListener` and `Net.addListener` are supplied for this purpose. Since Node and Net objects are already NeticaEventListeners, they each possess an `eventOccurred (NeticaEvent)` method. If you should choose to override this method, it is important that you call the base class method `super.eventOccurred(event)` in your method, so that this node or net will still be able to handle deleting events properly.

Currently events are generated for the creation, removal and duplication of Nodes and Nets. Future versions of Netica-J will include more types of events. If you have a request, please let us know.

2.5 Java Objects and Native Object Peers

Since Netica-J is a JNI API, many of the Java objects created are "proxies" of their native or "peer" counterpart objects internal to the core Netica binary. This is true of Environ, Net, Node, NetTester, NeticaError, Sensitivity, and Streamer. The remaining Java classes (General, NeticaEvent, NeticaException, NeticaListener, NodeList, State, User, Util, Value, and VisualNode) do not have peer equivalents.

The existence of peer relationships is usually transparent to the Java developer. Netica-J was designed to give the developer as much as possible the sense he/she is working in a 100% pure Java environment. That provides the best of both worlds: the productivity, safety and memory management of Java with the speed and reliability of a highly optimized, highly tested, widely used native binary.

The only situations where you need to know about peer objects is when considering finalization and the cleanup of native resources (discussed in the Finalizers section, below), or when working in a model-view-controller (MVC) environment where things could be happening to the native model objects, and the Java environment is presenting but one view on that model. This can happen, for instance, if Netica-J is communicating with peer objects inside Netica Application. A user of Netica Application could delete a native node via the GUI, and the Java environment would then find that its Node object had been disconnected from its peer. Netica-J has a standard Java Publish-and-Subscribe mechanism (using NeticaEventListeners) for Java objects to be made aware of such occurrences on the native side of the universe (see the Event Handling section, below).

2.6 Exception Handling

Exception handling in Netica-J works in the normal Java way. If a method encounters an unexpected situation that it cannot resolve, a NeticaException is thrown. The vast majority of Netica-J methods are able to throw a NeticaException. The toString() method of NeticaException details the reason for the Exception. Hence, your typical try-catch block could look something like this:

```
try {
      // call Netica-J methods
}
catch (NeticaException e) {
      e.printStackTrace();
}
```

If you are familiar with the Netica C API, you will find that Netica-J's exception handling mechanism makes coding much more convenient and straightforward, since you no longer need to actively check if an error has occurred. Netica-J looks after that for you, and will throw a NeticaException automatically if any "serious" ("show stopper") error occurs. By "serious" we mean any errors of severity level `ERROR_ERR` or `XXX_ERR` which means the requested operation was not completed.

Note that this means that `WARNING_ERR` and lower warnings do not result in a NeticaException being thrown, so in those cases where such warnings can occur, you can actively call the static method `NeticaError.getWarnings` after the method call, to determine if a warning has occurred and, if so, what the warning was about. See the javadocs for `NeticaError.getWarnings` for examples of this.

It is okay to call `NeticaError.getWarnings` only once in awhile, since warnings will accumulate until the next `getWarnings` invocation, whereupon they are cleared from the warnings list.

2.7 Finalizers & Memory Management

For large networks and large node tables, Netica can consume large amounts of memory. Often Java developers cease to worry about memory management, as the JVM's garbage collector will automatically collect Java objects that can no longer be referenced. However, the Java specification does not require that a JVM actually call the garbage collector whenever a Java object reference is no long used. It may or may not do so, and it may choose to do so on its own schedule. Accordingly, you may want to actively call the `delete()` or `finalize()` methods on resource-hungry objects when you are done with those objects, rather than wait for the JVM to free them.

For most Netica objects, calling `finalize()` frees all their native resources, but not for Node and State objects. For them, `finalize()` just indicates that you are done with the reference, but the native resources won't be freed until the owning Net or Node is freed. However, calling `Node.delete()` will remove the Node from its owning Net and free its resources, and calling `State.delete()` will remove the State from its owning Node and free its resources.

Note, if you ever override the `finalize()` method of any Netica-J class, be certain that you always call the base class finalizer method `super.finalize()` as your last instruction, so that Netica-J can do its own housekeeping upon the Java object being collected. For example, if your class extends norsys.netica.Streamer, and you need to override the `finalize()` method to perform special close-down handling of files and such, then your finalize method would look something like this:

```
/**
 *  overrides Streamer.finalize().
 */
public void finalize() throws NeticaException {
    . . .  your own finalization logic . . .
    super.finalize();
}
```

3 Probabilistic Inference

3.1 Bayes nets and Probabilistic Inference

A Bayes net (also known as a Bayesian network, BN, BBN, belief network, probabilistic causal network or graphical model) captures our believed relations (which may be uncertain, or imprecise) between a set of variables that are relevant to some problem. They might be relevant because we will be able to observe them, because we need to know their value to take some action or report some result, or because they are intermediate or internal variables that help us express the relationships between the rest of the variables.

Some Bayes nets are designed to be used only once for a single world situation. More often, Bayes nets are designed for repetitively occurring situations. They may be constructed using expert knowledge provided by some person, by an automatic learning process which examines many previous cases, or by a combination of the two. If the net is to be used repetitively, then it may be considered as a *knowledge base*. Sometimes nets that are built to be used only once are constructed automatically on-the-fly, perhaps by pasting together pieces of nets from libraries using templates. Then the libraries and templates together make up a knowledge base. Netica is designed to work for either type of application. It allows probabilities to be entered directly, perhaps originally coming from an expert, and it can learn probabilities from data. It will not handle templates directly, but it has the facilities for libraries and on-the-fly construction that such a program requires.

A classic example of the use of Bayes nets is in the medical domain. Here each new patient typically corresponds to a new case, and the problem is to diagnose the patient (i.e., find beliefs for the undetectable disease variables), or predict what is going to happen to the patient, or find an optimal prescription, given the values of observable variables (symptoms). A doctor may be the expert used to define the structure of the net, and provide initial conditional probabilities, based on his medical training and experience with previous cases. Then the net probabilities may be fine-tuned by using statistics from previous cases, and from new cases as they arrive.

When the Bayes net is constructed, one *node* is used for each scalar variable, which may be discrete, continuous, or propositional (true/false). Because of this, the words "node" and "variable" are used interchangeably throughout this manual, but "variable" usually refers to the real world or the original problem, while "node" usually refers to its representation within the Bayes net.

The nodes are then connected up with directed *links*. Usually a link from node A (the *parent*) to node B (the *child*) indicates that A causes B, that A partially causes or predisposes B, that B is an imperfect observation of A, that A and B are functionally related, or that A and B are statistically correlated. The precise definition of a link is based on conditional independence, and is explained in detail in an introductory work like RussellNorvig95 or Pearl88. Finally, probabilistic relations are provided for each node, which express the probability of that node having different values depending on the values of its parent nodes.

After the Bayes net is constructed, it may be applied. For each variable we know the value of, we enter that value into its node as a *finding* (also known as "evidence"). Then Netica does *probabilistic inference* to find beliefs for all the other variables. Suppose one of the nodes corresponds to the variable "temperature", and it can take on the values cold, medium and hot. Then an example belief for temperature could be: [cold - 0.1, medium - 0.5, hot - 0.4], indicating the probabilities that the temperature is cold, medium or hot. The final beliefs are sometimes called *posterior probabilities* (with *prior probabilities* being the probabilities before any findings were entered). Probabilistic inference done within a Bayes net is called *belief updating*.

Probabilistic inference only results in a set of beliefs at each node; it does not change the net (knowledge base) at all. If the findings that have been entered are a true example that might give some indication of cases which will be seen in the future, you may think that they should change the knowledge base a little bit as well, so that next time it is used its conditional probabilities more accurately reflect the real world. To achieve this you would also do *probability revision*, which is described in the "Learning From Case Data" chapter. As well as regular probabilistic inference, Netica can do a number of other types of inference, such as finding the most probable explanation (MPE), finding mutual information, solving decision nets, node absorption, etc.

3.2 Netica's Probabilistic Inference

There are three ways that Netica can do regular probabilistic inference: by junction tree compiling, by node absorptions, and by sampling. For most applications you will want to use the junction tree method, because usually it is most convenient and executes much faster. You may want to use node absorptions when you have some findings that are going to be repeated in many inferences (e.g. if you discover that something is always true in the context of interest), or large parts of a network that are irrelevant to a query, so can be pruned away. This section deals with junction trees; see the "Modifying Nets" chapter for information on link reversals and node absorption. Sampling is an inexact method, and is usually used only when the Bayes net is too large to compile into a junction tree, or there are continuous variables whose value you want to provide by equation, and don't want to discretize. It is accomplished by calling `Net.generateRandomCase()` many times (say 1000), with argument `method`=2 (`FORWARD_SAMPLING`), and recording what percentage of the cases resulted in the node of interest having a given value.

Netica uses the fastest known algorithm for exact general probabilistic inference in a compiled Bayes net, which is message passing in a *junction tree* (or "join tree") of cliques. This is based upon the work of LauritzenSpiegelhalter88, which is described in much simpler and more extensive terms in CowellDLS99 and SpiegelhalterDLC93.

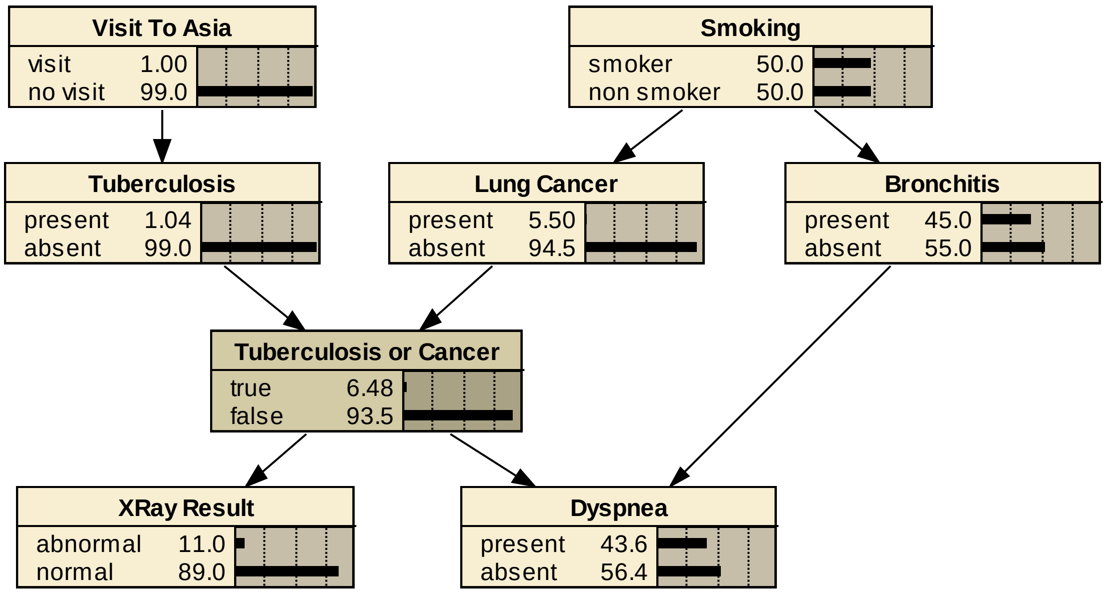
In this process the Bayes net is first "compiled" into a junction tree. The junction tree is implemented as a large set of data structures connected up with the original Bayes net, but invisible to you as a user of Netica. You enter findings for one or more nodes of the original Bayes net, and then when you want to know the resultant beliefs for some of the other nodes, belief updating is done by a message-passing algorithm operating on the underlying junction tree. It determines the resultant beliefs for each of the nodes of the original Bayes net, which it attaches to the nodes so that you can retrieve them. You may then enter some more findings (to be added to the first), or remove some findings, and when you request the resultant beliefs, belief updating will be performed again to take the new findings into account.

The amount of memory required by the junction tree, and the speed of belief updating are approximately proportional to each other, and are determined by the quality of the compilation. The quality of the compilation depends upon the *elimination order* used, which is a list of all the nodes in the net. Any order of the nodes will produce a successful compilation, but some do a much better job than others. You may specify an elimination order (perhaps from your own program, or by using Netica Application's "optimize compile"), or just let Netica API find a good one itself.

3.3 Example of Probabilistic Inference

Now let's look at an example of using the Netica API to do probabilistic inference. In this example we will read in a simple Bayes net from a file, compile it into a form suitable for fast inference, enter some findings, and see how the beliefs of a particular node change with each finding. The example program, **DoInference.java**, can be found in the **examples/** directory of the Netica-J installation.

The net we will use, called ChestClinic, is shown below. Although reasonable, it is a toy medical diagnosis example from LauritzenSpiegelhalter88 that has often been used in the past for demonstration purposes. To a certain degree, the links of the net correspond to causation. The two top nodes are "predispositions" which influence the likelihood of the diseases in the row below them. At the bottom are symptoms for the disease. Each possible state of the node is shown in the box. Ignore the bars for now; they were produced by the Netica Application program, and just show the probabilities of each state before any findings have arrived.



Before the example program below will work, the file containing the net “ChestClinic.dne” must exist in the “Data Files” subdirectory of the directory running the program. If you are running this example straight from examples/ directory of the Netica API distribution, that will already be the case. Otherwise you should obtain the file from the “examples/Data Files” directory of the Netica API distribution. Or you can build it yourself; the next chapter shows how, and at the end of that chapter is a file listing of the net (it is missing the Bronchitis and Dyspnea nodes, but they are not needed now anyway).

```
/*
 *  DoInference.java
 *
 *  Example use of Netica-J for doing probabilistic inference.
 */
import norsys.netica.*;

public class DoInference {
  public static void main (String[] args){
      try {
              Environ env = new Environ (null);

              // Read in the net created by the BuildNet.java example program.
              Net net = new Net (new Streamer ("Data Files/ChestClinicBuilt.dne"));
```

```
                Node visitAsia   = net.getNode ("VisitAsia");
                Node tuberculosis        = net.getNode ("Tuberculosis");
                Node xRay       = net.getNode ("XRay");

                net.compile();

                double belief = tuberculosis.getBelief ("present");
                System.out.println ("\nThe probability of tuberculosis is " + belief);

                xRay.finding().enterState ("abnormal");
                belief = tuberculosis.getBelief ("present");
                System.out.println ("\nGiven an abnormal X-ray,\n" +
                        "the probability of tuberculosis is " + belief);

                visitAsia.finding().enterState ("visit");
                belief = tuberculosis.getBelief ("present");
                System.out.println ("\nGiven an abnormal X-ray and a visit to Asia,\n" +
                        "the probability of tuberculosis is " + belief + "\n");
                net.finalize();
        }
        catch (Exception e){
                e.printStackTrace();
        }
    }
}
```

The program starts by using `new Environ()` as described in the previous chapter. Next, `new Net()` is used to read the file and create the net in memory. If you wish to have detailed descriptions of any of these functions, remember that you can look them up in the javadocs.

You can see that the entire program is wrapped within single try/catch block. Most Netica-J API methods throw NeticaException exceptions, if anything erroneous is attempted or results.

Next, `net.compile()` builds the junction tree of cliques and attaches it to the data structure of the Bayes net, but does not discard any of the information from the original Bayes net. We can now use this net to diagnose a new patient who has just entered the clinic.

In the next line `Node.getBelief()` is called to determine the probability tuberculosis is present:

```
        double belief = tuberculosis.getBelief ("present");
```

This causes a "belief updating" to be done, which finds new beliefs for all the nodes in the net. This step can be time consuming if the net is very large or highly connected. If `Node.getBelief()` is then called for some other node, it would return almost immediately, because the calculated beliefs have been saved at each node.

The program then prints out the probability of tuberculosis, which we can see is 1.04% from the listing of the program output below. This is the probability that the new patient has tuberculosis before we know anything else about him. The number may seem high, but then perhaps this net was built for people entering a certain clinic, and many of them wouldn't be there unless they have some kind of illness.

An X-ray is taken of the patient, and it comes out "abnormal". A Bayes net to be used for anything practical would define the X-ray outcome in more detail, but this will do for the example. We enter this finding into the net with:

```
    xRay.finding().enterState ("abnormal");
```

Then we use `Node.getBelief()` to cause belief updating to occur again (to incorporate the latest finding) and return the probability that the patient has tuberculosis given that his X-ray came out abnormal. The probability has now jumped to 9.24%, so we ask him if he has recently made a trip to Asia. When he answers to the affirmative, and we enter that finding, we then get a tuberculosis probability of 33.8%.

Exercise for the Reader: After further testing you might discover that our patient has lung cancer, and want to enter that as a finding. The lung cancer "explains away" the abnormal X-ray, and so our probability that he has tuberculosis would fall to 5.00%. Try editing and running DoInference.java.

The output produced will be:

```
>java […] DoInference


The probability of tuberculosis is 0.0104

Given an abnormal X-ray,
the probability of tuberculosis is 0.0924109

Given an abnormal X-ray and a visit to Asia,
the probability of tuberculosis is 0.337716

Given abnormal X-ray, Asia visit, and lung cancer,
the probability of tuberculosis is 0.05

>
```

For examples involving more complex types of findings, and the retraction of findings, see the "Findings and Cases" chapter.

4 Building and Saving Nets

In the previous chapter we loaded a Bayes net into memory from a file and then did probabilistic inference using it. Now we consider how to obtain the net file in the first place. Some possibilities are:

- Obtain a net file of interest from Norsys, another company or a colleague (by email, disk, downloading from a website, etc.). The file is machine and operating system independent. For examples of Bayes nets, see: http://www.norsys.com/netlibrary/index.htm
- Create the file using a text editor, according to the DNET file specification, or write a program that creates the DNET text file.
- Use the Netica Application program to construct the net on the screen of your computer using simple point-and-click drawing, and then save it to a file.
- Call functions in the Netica API to construct the net in memory. Once the net is in memory you may use it for probabilistic inference, learning, etc., or you can save it to a file for later usage.

In this chapter we will discuss the last method. Below is a complete program which constructs the ChestClinic net used in the previous chapter (except, to be more brief, it doesn't include the two nodes Bronchitis and Dyspnea, which are not required for the inference examples of that chapter – but the code in the examples directory does). This program, **BuildNet.java**, can be found in the **examples/** directory of your Netica-J installation.

```
/*
 *  BuildNet.java
 *
 *  Example use of Netica-J to construct a Bayes net and save it to file.
*/
import norsys.netica.*;
import norsys.neticaEx.aliases.Node;

public class BuildNet {
  public static void main (String[] args){
```

```
try {
        Node.setConstructorClass ("norsys.neticaEx.aliases.Node");
        Environ env = new Environ (null);

        Net net = new Net();
        net.setName ("ChestClinic");

        Node  visitAsia  = new  Node  ("VisitAsia",       "visit, no_visit", net);
        Node  tuberculosis     = new  Node  ("Tuberculosis",   "present, absent", net);
        Node  smoking  = new  Node  ("Smoking",        "smoker, nonsmoker", net);
        Node  cancer    = new  Node  ("Cancer",         "present, absent", net);
        Node  tbOrCa    = new  Node  ("TbOrCa",         "true, false", net);
        Node  xRay      = new  Node  ("XRay",  "abnormal, normal", net);

        visitAsia.setTitle ("Visit to Asia");
        cancer.setTitle ("Lung Cancer");
        tbOrCa.setTitle ("Tuberculosis or Cancer");

        visitAsia.state("visit").setTitle ("Visited Asia within the last 3 years");

        tuberculosis.addLink (visitAsia);      // puts link from visitAsia to tuberculosis
        cancer.addLink (smoking);
        tbOrCa.addLink (tuberculosis);
        tbOrCa.addLink (cancer);
        xRay.addLink (tbOrCa);

        visitAsia.setCPTable (0.01, 0.99);
        smoking.setCPTable (0.5,  0.5);

                // VisitAsia       present absent
        tuberculosis.setCPTable         ("visit",  0.05,     0.95);
        tuberculosis.setCPTable         ("no_visit",       0.01,     0.99);

                // Smoking        present absent
        cancer.setCPTable       ("smoker",       0.1,      0.9);
        cancer.setCPTable       ("nonsmoker",   0.01,     0.99);

                // TbOrCa         abnormal        normal
        xRay.setCPTable         ("true",  0.98,     0.02);
        xRay.setCPTable         ("false", 0.05,     0.95);
```

```
                tbOrCa.setEquation ("TbOrCa (Tuberculosis, Cancer) = Tuberculosis || Cancer");
                tbOrCa.equationToTable (1, false, false);

                Streamer stream = new Streamer ("Data Files/ChestClinicBuilt.dne");
                net.write (stream);

                net.finalize();        // free resources immediately and safely
        }
        catch (Exception e){
                e.printStackTrace();
        }
    }
  }
```

First, the above program constructs a new empty net with `new Net()` and then adds each of the nodes with `new Node()`. Each node represents some scalar variable of interest, either discrete or continuous. The first string passed to the Node constructor is the name of the node, and the second is a comma-delimited list of state names for that node. The states must be *mutually exclusive* (value can't be two different states at the same time), and *exhaustive* (it is always in one of the states). Sometimes it is easiest to satisfy the exhaustive condition by having a state called "other".

The names of the net, nodes and states are passed as Strings. These strings must meet the requirements of an *IDname*, which are:

- The name must be between 1 and `General.NAME_MAX` (= 30) characters long, inclusive.
- The name must consist entirely of alphabetic characters (a-z and A-Z), digits and underscores ('_').
- The name must start with an alphabetic character.
- Often they must be unique within the object they apply to. Comparisons are case-sensitive.

In general, Netica restricts names for all objects in this way. If you find that overly restrictive, then you can also give the object a "title", which is an unrestricted Unicode string. Some objects can have a "comment" as well, which is also an unrestricted Unicode string, and it would not be out of the ordinary if this were thousands of characters long.

The states do not need to be named, so instead of the list of state names, a "2" could be passed to `Node()` indicating the number of states the node can take on (0 would be passed for a continuous node). Later, the program could set the state names of the nodes using `Node.setStateNames()`. Or they could be left unnamed, but in general it is recommended to name them in order to keep track of the meanings of the states, and to be able to refer to the states by names, as was done in the last chapter. Then a couple of nodes are given titles, which also aren't really required, but are a bit more descriptive than their names (the idea is to keep names short for convenience).

Next, the nodes are linked together with `Node.addLink()`. A call of the form `nodeC.addLink (nodeP)` makes nodeP a "parent" of nodeC, which means we wish to express the probabilities of nodeC as a function of (i.e. "conditioned on") values of nodeP. Usually the link indicates that nodeP causes nodeC, that nodeC is an imperfect observation of nodeP, or that the two nodes are statistically correlated.

Finally, the conditional probability tables (CPTs) are added. For each node, these are the probabilities of each of its states, conditioned on the states of its parent nodes. They are built up by multiple calls to `NodeEx.setCPTable` (which is defined in NodeEx.java as a convenient way to call `Node.setCPTable()`). The first argument in each call is the names of the conditioning states of its parents as a String. Finally comes a list of numbers, being the probabilities for each of the states of the node.

For example: `cancer.setCPTable ("smoker", 0.1, 0.9)` means that the probability that cancer is in its first state given that its parent is in state "smoker" is 0.1, and the probability that it's in its second state is 0.9. In probabilistic notation: P(cancer=present | smoking=smoker) = 0.1

As another example, `tbOrCa.setCPTable ("present", "absent", 1.0, 0.0)` means:
P(TbOrCa=true | Tuberculosis = present, Cancer= absent) = 1.0

If "*" is used as the name of a conditioning state, then it will apply to all values of that parent node. Likewise `State.EVERY_STATE` can be used with `setCPTable()`.

There is one thing to be cautious of when using `setCPTable`. If speed is critical, and you must set large probability tables, use `Node.setCPTable()` instead of `NodeEx.setCPTable()`. For example, `tbOrCa.setCPTable (TbOrCa, "present", "absent",  1.0, 0.0)` could be accomplished by:

```
parentStates[0] = 0;     parentStates[1] = 1;     // present absent
probs[0] = 1.0;  probs[1] = 0.0;
tbOrCa.setCPTable (parent_states, probs);
```

There is an even faster way to set the whole CPT table with one function call. You call `Node.setCPTable(double[] cptTable)`, the whole table for the probability array. The table you pass in should be in row-major form with the last parent varying fastest (the same order the table is displayed in the CPT editor of Netica Application).

If you wish to give a node a deterministic relationship, rather than probabilistic, you may use Node.setStateFuncTable().

Now the net is fully constructed in memory, and we could use it for inference, do net transforms, etc., but in this example we just save it to a file for later use, by calling `Net.write()`. The resulting file is a pure ASCII text file which can be read back by Netica API or by Netica Application, whether they are running on the same computer or another type of computer. The file adheres to the DNET format, which is described in the document "DNET File Format". It will look similar to the below:

```
// ~->[DNET-1]->~

bnet Built_ChestClinic {

   node VisitAsia {
      kind = NATURE;
      discrete = TRUE;
      states = (visit, no_visit);
      parents = ();
      probs =
         // visit        no_visit
           (0.01,        0.99);
      };

   node Tuberculosis {
      kind = NATURE;
      discrete = TRUE;
      states = (present, absent);
      parents = (VisitAsia);
      probs =
         // present      absent        // VisitAsia
           (0.05,        0.95,         // visit
            0.01,        0.99);        // no_visit
      };

   node Smoking {
      kind = NATURE;
      discrete = TRUE;
      states = (smoker, nonsmoker);
      parents = ();
      probs =
         // smoker       nonsmoker
           (0.5,         0.5);
      };
node Cancer {
   kind = NATURE;
   discrete = TRUE;
   states = (present, absent);
   parents = (Smoking);
   probs =
      // present     absent        // Smoking
        (0.1,        0.9,          // smoker
         0.01,       0.99);        // nonsmoker
   title = "Lung Cancer";
   };


node TbOrCa {
   kind = NATURE;
   discrete = TRUE;
   states = (true, false);
   parents = (Tuberculosis, Cancer);
   probs =
      // true       false         // Tuberculosis Cancer
        (1,         0,            // present      present
         1,         0,            // present      absent
         1,         0,            // absent       present
         0,         1);           // absent       absent
   title = "Tuberculosis or Cancer";
   };

node XRay {
   kind = NATURE;
   discrete = TRUE;
   states = (abnormal, normal);
   parents = (TbOrCa);
   probs =
      // abnormal    normal        // TbOrCa
        (0.98,       0.02,         // true
         0.05,       0.95);        // false
   };
};
```

The DNET file format is a text format, but Netica can also work with a binary format called NETA. The binary files are much smaller, they usually read faster, and Netica can encrypt them. To save the above net in NETA format, you would change the call to `net.write()` to be:

```
net.write (new Streamer ("Built_ChestClinic.neta"));
```

That is, the call is exactly the same as for a DNET file, but the file name has an extension of .neta instead of anything else. The Netica API call for reading the NETA file is the same as for a DNET file; Netica will recognize each and handle it appropriately. If you wish, you can encrypt the net so that only software that knows the password will be able to read it.:

```
Streamer stream = new Streamer ("Built_ChestClinic.neta");
stream.setPassword ("MyPassword123");
net.write (stream);        // writes an encrypted file
```

Encryption is useful when you need to distribute the net with your application for Netica API to use, but the net contains proprietary information. Encrypted nets can also be read (or created) by Netica Application, provided that the user enters the correct password. For a full code example, including reading encrypted files, see the javadocs for `Streamer.setPassword()`.

There are a number of other functions that may be used when constructing a net. For a list of them, see the "Low-Level Net Modification" section of the " Functions by Category" chapter, and for detailed descriptions of each one, see the javadocs for the Net class.

For another example of constructing a net, which demonstrates how to build a decision net, create decision and utility nodes, and work with 3-state and continuous nodes, see the "Decision Nets" chapter.

5 Findings and Cases

In the "Probabilistic Inference" chapter we saw how to enter positive findings into a Bayes net to do probabilistic inference (findings are also known as “evidence”). A *positive finding* is the observation or knowledge that some discrete node definitely has a particular value. However, we may discover that some node definitely does *not* have some particular value, and not have any more information to help us determine what value it does have. This is called a *negative finding*.

For example, say the node 'Temperature' can take on the values cold, medium, and hot. We may obtain information that the temperature is not hot, although it doesn't distinguish between medium and cold at all. This is a single negative finding. If later we receive another negative finding that the temperature is not medium, then we can conclude that it is cold. So, several negative findings can be equivalent to one positive finding.

A third type of finding is a *soft finding* (also known as “virtual evidence”) or *likelihood finding* . In this case we receive uncertain information about the value of some discrete node. It could be from an imperfect sensor, or from a friend who is not always right. Say we have a thermosensor to measure 'Temperature', which is designed so that when the temperature is hot it is supposed to turn on. In actual practice we find that when the temperature is cold the sensor never goes on, when the temperature is medium it goes on 10% of time, and when it is hot it always goes on. If at a certain time we observe the sensor on, and want to enter this finding into the Temperature node, then we do so as a likelihood finding. A likelihood finding consists of one probability for each state of the node, which is the probability that the observation would be made if the node were in that state. For our temperature example, the likelihood finding would be (0, 0.1, 1). A common mistake is to think that the likelihood is the probability of the state given the observation made (in which case the numbers would have to add to one), but it is the other way around.

A positive finding is equivalent to a likelihood finding consisting of all 0s except a single 1. A negative finding is equivalent to a likelihood finding consisting of all 1s (or some other nonzero number) except a single 0. Two *independent* findings for a node can be combined by component-wise multiplication of their likelihood vectors. If they are not independent, and it is too inaccurate to approximate them as independent, then they should be combined by adding 2 child nodes to the observed node in the original net, one for each observation, connecting them together to show the dependency, and then entering positive findings for the child nodes.

Netica has functions for the direct entry of positive findings, negative findings, likelihood findings, and also findings that a continuous node has a certain value. If several findings are entered for the same node, then it combines them as if they were independent observations, and generates an error if they are inconsistent. Checking for consistency between the findings of one node and those of another node (given the inter-node relations encoded in the net), is only done if belief updating is done after each finding is entered, which will be the case if the net is auto-updating (see `Net.setAutoUpdate()`) or if Node.getBeliefs() is called between entering findings.

As an example, consider the following section of code to enter findings for `node`, which has 4 states:

```
(a)     int fst;
(b)     Node node;
(c)     float[] clike, belief;
(d)     float[] like = new float[4];

(1)     like[0] = 0.6F;   like[1] = 0.6F;   like[2] = 1.0F;   like[3] = 1.0F;
(2)     node.finding().enterLikelihood (like);
(3)     node.finding().enterStateNot (1);
(4)     like[0] = 0.5F;   like[1] = 0.6F;   like[2] = 0.0F;   like[3] = 0.5F;
(5)     node.finding().enterLikelihood (like);
(6)     clike = node.finding().getLikelihood();
(7)     //  node.finding().enterState (2);
(8)     belief = node.getBeliefs()
(9)     fst = node.finding().getState();
(10)    node.finding().clear();
(11)    node.finding().enterState (2);
(12)    fst = node.finding().getState();
(13)    clike = node.finding().getLikelihood();
```

Step 1 sets up a likelihood vector, and step 2 enters it as a finding for `node`. The finding means that an observation was made that would certainly be observed if `node` were in state 2 or 3, and that would occur with probability 0.6 if `node` were in state 0 or 1. Step 3 enters a negative finding which means "the value of `node` is not state 1". Steps 4 and 5 enter another likelihood finding, and then step 6 retrieves the likelihood vector for the accumulated findings so far. It will have the values:

```
    clike[0] = 0.3    clike[1] = 0.0    clike[2] = 0.0    clike[3] = 0.5
```

Notice that clike[1] is 0 due to the negative finding of step 3, and clike[2] is 0 due to the 0 in the likelihood finding of steps 4&5.

Step 7 is commented out, but if it weren't it would generate an error because saying "the value of `node` is state 2" is inconsistent with the likelihood finding of steps 4&5.

Step 8 causes a belief updating to be done, and it could return a belief vector with the following values:

```
belief[0] = 0.9    belief[1] = 0.0    belief[2] = 0.0    belief[3] = 0.1
```

Even though the accumulated likelihood (`clike`) said state 3 was the most likely value for `node`, when the findings for other nodes, and their relations with `node`, were taken into account, state 0 became more probable than state 1. In general, it is not possible to determine anything about what the belief of a node is going to be based just on its accumulated likelihood findings, except that states with a zero likelihood will have a zero belief.

Step 9 demonstrates `getState()`being used to query what finding has been entered for `node`. It is designed to retrieve positive findings, and since `node` has likelihood findings, it will just return the constant `Value.LIKELIHOOD_VALUE`.

Step 10 retracts all the findings that have been entered for `node`, thereby undoing all of the above, and step 11 enters the positive finding that the value of `node` is state 2, which won't generate an error this time like it would have in step 7. When `getState()` is called in step 12, it will now return 2, and the values of `clike` after step 13 will be:

```
clike[0] = 0.0    clike[1] = 0.0    clike[2] = 1.0    clike[3] = 0.0
```

5.1 Cases and Case Files

The set of all findings entered into the nodes of a single Bayes net is referred to as a *case*. A case may be saved to a file for later retrieval. Case files may consist of a single case, or of many cases. Case files act as databases; they may be used to swap cases in and out of a net as additional findings are obtained or beliefs required, to transfer a case from one net to another, or as data to learn a new net.

Some ways you can make a case file are:

- Use a text editor to manually construct it, according to the specification below.
- Write a program whose output is a case file.
- Export it (as a CSV or tab-delimited text file) from a spreadsheet or database program. Or you can copy from the spreadsheet or database program, paste into a text editor, and save as a text file.

- Extract it from a database using `Caseset.addCases (DatabaseManager, …)` followed by `Caseset.writeCases(…)`
- Use Netica Application to enter findings by pointing and clicking, and then choose "Save Case" from the menu.
- Call Netica API functions to enter the case as findings into a Bayes net, write the case to a file, and repeat for each case to be put in the file.

Case files (single-case or multi-case) are pure ASCII text files. They may contain `// ~->[CASE-1]->~` somewhere in the first 3 lines, to indicate to Netica what the file contains, but that isn't required. Then comes a line consisting of headings for the columns. Each heading corresponds to one variable of the case, and is the name of the node used to represent the variable (sometimes the variables are called *attributes* and the entries in the column *values*, i.e. *attribute-value*). The headings are separated by spaces and/or tabs (it doesn't matter how many).

The case data is next, with one case per line (a single-case file would only have one such line). The values of the variables are in the same order as the heading line, and are separated by spaces or tabs (the columns don't have to "line up" as they do in the example files below). The value of a discrete variable is given by its state name, or if it doesn't have a state name, then by the number symbol, followed by its state number (e.g. #3). The state names are preferred, since the order of the states may be changed some time, and that would render the file invalid.

The value of a continuous variable is given by a number, expressed as an integer, decimal, or in scientific notation (e.g. -3.21e-7). If the variable has been discretized, then the value may be given by a state name or state number, but the continuous number is preferred if it is available. That way, the case file can be used for different discretizations of that variable in the future. Try to use the correct number of significant figures, since future versions of Netica may use this information.

A single-case file is the same as one with multiple cases, except it just has 1 case. There may be as much whitespace as desired between the lines, including Java/C/C++ style comments. If the values of some of the variables are unknown for some of the cases, then a question mark or asterisk (? or *) is put in the file instead of the value (this is known as *missing data*).

If you read in a case, and the case file has a node value that doesn't correspond to any state of that node in the net (e.g. the states of net node 'color' are 'red' and 'green', and the value for color in the case file is 'blue'), then an error will be generated. An exception to this is if one of the states of the net node is called "other". Then the case will be read without error, and the finding for the node will be 'other'.

There are two special columns that a file may have which don't correspond to nodes. One provides an identification number for each case, which must be an integer between 0 and 2 billion. The heading for this column is "IDnum". Identification numbers do not have to be in order through the file. The other special column has the heading "NumCases", and indicates the frequency or multiplicity of the case. A multiplicity of m indicates m cases with the same variable values. It is not required to be an integer, so it can be used to represent a frequency of occurrence if desired. The missing data symbol ("*") should not appear in either of these columns if they exist.

As an example of a case file, here is a listing of "ChestClinic.cas" which is produced by the program SimulateCases.java, listed below and included in the **examples/** directory of your distribution. Note that the case file you obtain may be a little different, since random numbers are involved. It has an IDnum column, but no frequency column.

```
IDnum  VisitAsia  Tuberculosis  Smoking    Cancer   TbOrCa  XRay      Bronchitis  Dyspnea
1      no_visit   present       smoker     absent   true    abnormal  absent      present
2      no_visit   absent        smoker     absent   false   normal    present     present
3      no_visit   absent        smoker     present  true    abnormal  present     present
4      no_visit   absent        nonsmoker  absent   false   normal    absent      absent
5      no_visit   absent        smoker     present  true    abnormal  present     present
6      no_visit   absent        smoker     absent   false   abnormal  present     present
....
198    no_visit   absent        smoker     absent   false   normal    present     present
200    no_visit   absent        smoker     present  true    abnormal  present     present
```

Here is listing of SimulateCases.java, the program which generated the above case file:

```
/*
 *  SimulateCases.java
 *
 *  Example use of Netica-J for generating random cases that follow
 *  the probability distribution given by a Bayes net.
 */
import java.io.File;
import norsys.netica.*;

public class SimulateCases {

  public static void main (String[] args){
      int numCases = 200;
      System.out.println ("Creating " + numCases + " random cases...");

      try {
              Environ env = new Environ (null);

              // Read in the net created by the BuildNet.java example program.
```

```
                Net  net = new Net (new Streamer ("Data Files/ChestClinicBuilt.dne"));
                NodeList nodes = net.getNodes();

                (new File ("Data Files/ChestClinic.cas")).delete();     // since "ChestClinic.cas" may
                                                    // exist from a previous run and we do not wish to append
                Streamer caseFile = new Streamer ("Data Files/ChestClinic.cas");

                net.compile();

                for (int n = 0;  n < numCases;  n++) {
                        net.retractFindings();
                        int res = net.generateRandomCase (nodes, 0, 20);
                        if (res >= 0)
                                net.writeFindings (caseFile, nodes,  n, -1.0);
                
                net.finalize();
                }
        }
        catch (Exception e){
                e.printStackTrace();
        }
    }
```

First the program reads in the same net that we built in the “Building and Saving Nets” chapter. Then it deletes a file named "ChestClinic.cas" if there is one (otherwise it would try to add the cases to this file). Then, in a loop repeated 20 times it generates a random case from the ChestClinic net. These cases will be distributed according to the probability distribution of that net. Each case is saved to the case file named "ChestClinic.cas", a sample of which we saw above. We will use this case file in the next chapter, “Learning From Case Data”.

Here is another example of a case file, this time for cars brought into a garage (notice BatAge, which is a continuous variable):

```
// ~->[CASE-1]->~
Starts  BatAge  Cranks  Lights   StMotor  SpPlug   MFuse  Alter  BatVolt  Dist  PlugVolt  Timing
false   5.9     false   off      ?        fouled   okay   ?      dead     ?     ?         good
false   1.3     false   off      ?        okay     okay   ?      dead     ?     none      bad
false   5.2     false   off      okay     okay     okay   okay   dead     okay  none      good
true    4.1     true    bright   ?        okay     okay   ?      strong   okay  strong    ?
true    2.7     ?       bright   ?        wide     okay   ?      strong   okay  ?         ?
?       ?       true    bright   ?        fouled   okay   ?      ?        okay  strong    good
false   1.7     true    off      okay     okay     okay   okay   dead     ?     none      good
true    2.9     true    bright   ?        ?        ?      ?      strong   okay  strong    ?
```

5.2 Casesets

Netica-J has a very powerful class called Caseset. A Caseset instance represents a set of cases that may be in a database, in memory or in a disk file (in any of a number of formats). You use the same functions to operate on Casesets no matter where they are or in what format they are.

To make a Caseset, you first create an empty one with one of the Caseset constructors. For example:

```
Caseset cs = new Caseset();
```

Then you add cases to the Caseset. If you want them to come from a database, you use `Caseset.addCases (DatabaseManager, …)`, as described in the next section. Alternatively, you can add cases from a text file of cases in the format described in the previous section. You first create a `Streamer` that refers to the file, using `new Streamer ("yourFile.cas")`. If you are creating the case file dynamically, it is probably much more efficient to just create it in a memory buffer, say a byte array, and then create the `new Streamer (new ByteArrayInputStream (yourByteArray))` instead. Then you add the cases within it to the Caseset using:

```
Caseset.addCases (Streamer streamer, double degree, String control);
```

With the current version of Netica, you can only add cases to a Caseset once.

You can write all the cases in a Caseset to a file with:

```
Caseset.writeCases (Streamer file, String control);
```

That can be used to extract the cases from a database, and then write them out to a text file.

You can use `Learner.learnCPTs()` to learn the conditional probability tables of a Bayes net from a Caseset, as described in the Learning chapter. Future versions of Netica will have many more operations available for Casesets.

When you are done with the Caseset, you may reduce the resources required by calling:

```
Caseset.finalize();
```

5.3 Connecting with a Database

Netica can connect with a database (such as that created by Microsoft SQL Server, Microsoft Access, MySQL or Oracle), and use the data within to learn a Bayes net, performance test a Bayes net, etc. First you create a DatabaseManager instance using:

```
DatabaseManager (String odbcConnectionString, String control, Environ env);
```

The connection string has information on the file location of the database, the driver to use (depending on whether MySQL, MS Access, etc.), any password required to access the database, etc, as described in the javadocs for the `DatabaseManager` constructor.

Now that you have the database manager, you can use it to execute whatever SQL commands you would like on the database, using:

```
    void DatabaseManager.executeSQL (String sqlCmd, String control);
```

If you wish to put all the findings currently entered into a Bayes net as a new record of the database, use:

```
    void DatabaseManager.insertFindings (NodeList nodeList, String columnNames,
                                         String tables, String control);
```

Where ColumnNames is a list of columns in the database table to map to the list of nodes nodeList.

To use the database with Netica functions (such as learning from data), you first create an empty Caseset instance and then add the database to it with:

```
    void Caseset.addCases (DatabaseManager dbMgr,
                           double degree, NodeList nodeList,
                           String columnNames, String tables,
                           String condition, String control);
```

Then the resulting Caseset can be used as described in the "Caseset" section.

The previous two functions assumed that the Bayes net already had nodes that correspond to columns of the database (to form the nodeList parameter for them). If it doesn't, then you can create the nodes with:

```
    void DatabaseManager.addNodes (Net net, String columns, String tables, String condition, String
control)
```

When you are done with the database manager, you may close the connection and free resources by calling:

```
    void DatabaseManager.finalize();
```

Here is an example program to learn Bayes net CPT tables from a database. For more explanation on learning, see the next chapter, and especially a similar code example in the "EM and Gradient Descent Learning" section.

```
DatabaseManager db =
    new DatabaseManager ("driver=Microsoft Access Driver (*.mdb); dbq=.\\myDB.mdb;UID=dba1;",
                         "pooling", env);
Net net = new Net();

// ... Put code to add nodes and links to net here ...
//    You could use DatabaseManager.addNodes();

NodeList nodes = net.getNodes();
Caseset cs = new Caseset();
cs.addCases (db, 1.0, nodes,
                    "Sex, Height, \"Owns a house\", \"Number of dogs\"",
                     null, "'Owns a house' = 'yes'", null);
Learner learner = new Learner (Learner.EM_LEARNING, null, env);
learner.learnCPTs (nodes, cs, 1.0);
cs.finalize();
db.finalize();
```

5.4 Case Files with Uncertain Findings

The case files discussed so far have only had values that were completely certain (or completely missing). But Netica can also create and read case files having values that are known with limited accuracy, or only known to within some likelihood. In fact, Netica has a very elegant, practical and powerful way of expressing uncertain findings, known as the UVF file format.

When Netica reads in a case containing uncertain findings (for example, by `Net.readFindings()`), it will enter them into the Bayes net as soft findings, so any probabilistic inference, node absorption, sensitivity analysis, etc. will properly account for them. Also, the operations on case files, such as learning from cases, test net with cases and process cases, will work properly on case files containing uncertain values. When learning from such cases, some learning algorithms will work better than others. For more information on that, and an example of working with case files having uncertain findings, see the "EM and Gradient Descent Learning" section in the next chapter.

Below is a list of the different types of uncertain findings, their syntax in the case file, and what they mean. Each type of uncertain finding can appear anywhere in a case file where a regular finding normally would. For example, a UVF file could be a regular case file (as described in earlier sections), a CSV file, or tab delimited text file, but with some of the values replaced with entries having the syntax described below.

Gaussian

Syntax:	`m+-s`	m and s are real numbers	
Examples:	`5+-2`	`3.27+-0.03`	`0+-1e-5`

This is for a Gaussian (also known as "normal") soft finding, where the m is the mean and s is the standard deviation. Note that there cannot be any space before or after the +-. The uncertainties in measurements from lab instruments, or polling results, are often expressed with a +- notation, and indicate a Gaussian distribution, so they can be easily input into Netica (although in some contexts they may indicate an interval distribution, which is described below).

<u>Interval</u>

Syntax:	`[a, b]`	a and b are real numbers, state names or state indexes preceded by #

Examples: `[0, 10]` `[-3, 2.27]` `[lo, med]` `[#1, #3]`

This finding indicates the value is known to be within the two endpoints. There may be spaces before or after the comma or brackets. Intervals of states include both endpoints, so [lo, med] includes states lo, med and any states between. Intervals of continuous variables include the lower endpoint, but not the upper endpoint, so [0, 10] for variable X means $0 \leq X < 10$. Likelihood within the interval is one; outside the interval it is zero.

<u>Unbounded Interval</u>

Syntax:	`>b` or `<b`	b is a real number, state name or state index preceded by #

Examples: `>4.75` `<-10` `<med` `>#2`

This finding indicates that the value is above a certain point, or below a certain point. When b is a state, the interval includes the endpoint; when it is for a continuous variable, the interval includes the endpoint only for > intervals (so > is really ≥). The interval can potentially extend to infinity, but in practice will be limited by known maximum values for the variable. Likelihood within the interval is one; outside the interval it is zero.

<u>Set of Possibilities</u>

Syntax:	$\{s_1, s_2, \ldots s_n\}$	each s_i is a state name, state index preceded by #, Gaussian, interval or unbounded interval

Examples: `{lo, med}` `{red, blue, yellow}` `{#5, #7, #1}`

`{[0,3.5], [4.5, 10]}` `{[#35,#122], >#500}`

This finding indicates the value is known to be one of a listed set of possibilities. There may be spaces before or after the comma or brackets. The finding can be considered to be a disjunction of the elements. So if there is only 1 element {x}, the value is known to be that element, and it can be written as just x. The likelihood of elements in the set is one, of those not in the set is zero.

Set of Impossibilities

Syntax: `~{s1, s2, … sn}` each si is a state name, state index preceded by #, interval or unbounded interval

Examples: `~{lo}` `~{black, orange, green}` `~{#5, #7, #1}`

`~{[0, 3.5]}`

This finding indicates the value is known to **not** be any of a listed set of possibilities. There may be spaces before or after the comma or braces, but not between the tilde (~) and the brace. This is the same as "Set of Possibilities" except the "possible" states are those that are not listed, rather than those that are listed. The likelihood of elements in the set is zero; of those not in the set, it is one.

A negative finding can be represented easily by just listing the state(s) eliminated by the observation.

Likelihood

Syntax: `{s1 p1, s2 p2, … sn pn}` each s_i is a state name, state index preceded by #, Gaussian, interval or unbounded interval. Each p_i is a number between 0 and 1. Some p_i may be absent.

Examples: `{female .8, male .3}` `{3+-1 0.2, 7+-2 0.4}`

`{[0,3.5] .05, [3.5,10] 0.1, other 0.5}`

This is the same as a set of possibilities, but each possibility is weighted with a likelihood that appears after it (separated by a space). Since the numbers are likelihoods, they do not need to sum to one. The most common type of likelihood vectors are for discrete variables, where each state is listed, followed by its likelihood. Any states that appear without a number have a likelihood of 1, and any states that don't appear at all have a likelihood of 0.

Arbitrary likelihood distributions for continuous variables can be formed by a series of adjacent intervals, each with its own probability. Or the elements can overlap, and then their likelihoods are combined. For example {[0,10] .1, [2,4] .2} would be the combination of a rect function extending from 0 to 10 with height 0.1, and another rect from 2 to 4 with a height of 0.2.

Another useful distribution that is easy to form is a weighted combination of Gaussians. For example {3+-1 0.2, 7+-2 0.4} is a bi-modal distribution with peaks at 3 and 7.

It is possible to mix weighted Gaussians, intervals, and discrete states within a single { ... } likelihood vector.

Scaled Likelihood

Syntax: `~{s₁ p₁, s₂ p₂, … sₙ pₙ}` each s_i is a state name, state index preceded by #, interval, or unbounded interval. Each p_i is a positive number. Some p_i may be absent.

Examples: `~{red, green, orange .2, yellow .8}`

`~{[0,2] .4, [2,6] .2}`

This is like a set of impossibilities, but with each entry weighted by a number, which appears after it. If no number appears after it, its likelihood is 0. Entries that have numbers above 1 are indicated to be more probable than those not listed, and entries with numbers below 1 are less probable than the unlisted ones (unlisted entries have a likelihood of 1).

Complete Uncertainty

Syntax: * just an asterisk

If nothing is known regarding the value of this variable (i.e. missing data), then a question mark ? or an asterisk * should be used to indicate that. It is equivalent to `~{}` which is a likelihood of all ones.

Complete Certainty

Syntax: `v` v is a real number, state name, or state index preceded by #

Of course, any finding of complete certainty may be represented in the usual way, which is simply the value it is known to be. Likelihood for that value is 1, and for all others 0.

6 Learning From Case Data

Bayes net learning is the process of automatically determining a representative Bayes net given data in the form of cases (called the *training cases*). Each case (i.e. “record”) represents an example, event, object or situation in the world (presumably that exists or has occurred), and the case supplies values for a set of variables which describes the event, object, etc, as specified in the previous chapter. Each variable will become a node in the learned net (unless you want to ignore some of them), and the possible values of that variable will become the node’s states.

The learned net can be used to analyze a new case which comes from the same (or appropriately similar) world as the training cases did. Typically the new case will provide values for only some of the variables. These are entered as findings, and then Netica does probabilistic inference to determine beliefs for the values of the rest of the variables for that case. Sometimes we aren't interested in values for all the rest of the variables, but only some of them, and we call the nodes that correspond to these variables *target nodes*. If the links of the net correspond to a causal structure, and the target nodes are ancestors of the nodes with findings, then you could say that the net has learned to do diagnosis. If the target nodes are descendants, then the net has learned to do prediction, and if the target node corresponds to a "class" variable, then the net has learned to do classification. Of course the same net could do all three, even at the same time.

The Bayes net learning task has traditionally been divided into two parts: structure learning and parameter learning. *Structure learning* determines the dependence and independence of variables and suggests a direction of causation, in other words, the placement of the links in the net. *Parameter learning* determines the conditional probability table (CPT) at each node, given the link structures and the data. Currently Netica only does parameter learning (i.e., you link up the nodes before learning begins). However, you can use Netica to do structure learning by writing your own small program that tests a number of candidate link structures to find the best one. You write a function which searches through some candidate link structures that are plausible and practical in your domain, perhaps also adding trial latent variables. For each structure you use Netica’s parameter learning functions described in this chapter, then test the resulting net with Netica's net testing functions also described in this chapter. The net that scores the highest (perhaps penalized for complexity) is the best structure.

You might not want Netica to learn the CPTs of all the nodes in your Bayes net. Some of the nodes may have CPTs that have already been learned well, were created manually by an expert, or are based on theoretical knowledge of the problem at hand (perhaps expressed by an equation). Netica allows you to restrict the learning process to a subset of the nodes, and those nodes are called the *learning nodes*.

If every case supplies a value with certainty for each of the variables, then the learning process is greatly simplified. If not, there are varying degrees of partial information:

1. If there is a variable for which none of the cases have any information, that variable is known as a *latent variable* or "hidden variable".

2. If some cases have values for a certain variable, and others don't, that is known as *missing data*.

3. Some values for variables may not be given with certainty, but only as *soft findings.*

It may seem strange to be learning a net that has latent variables, since none of the training cases have any information on them. You introduce a latent variable as a parent node (or intermediate node) of multiple child nodes, and Netica uses the correlations among the children to determine relationships between the latent node with others. The result may be a Bayes net that is actually simpler (has fewer CPT entries), and generalizes better (i.e. performs better on new cases seen). For an example of using Netica to learn a latent variable, see the "Learn Latent.dne" net in the examples folder of the Netica Application distribution, or get it from the Norsys net library.

6.1 Algorithms

There are three main types of algorithms that Netica can use to learn CPTs: counting, expectation-maximization (EM) and gradient descent. Of the three, "counting" is by far the fastest and simplest, and should be used whenever it can. It can be used whenever there is not much missing data or uncertain findings for the learning nodes or their parents. When learning the CPT of a node by counting, Netica will only use those cases which supply values of certainty for the node and all of its parents. Obviously, if any of those are latent nodes, counting will not work.

If you can't use counting, then you must use EM learning or gradient descent. For each application area, it is usually best to try each one to see which gives the better results. Generally speaking, EM learning is more robust (i.e., gives good results in wide variety of situations), but sometimes gradient descent is faster. For all three algorithms, the order of the cases doesn't matter.

During Bayes net learning, we are trying to find the *maximum likelihood* Bayes net, which is the net that is the most likely given the data. If N is the net and D is the data, we are looking for the N which gives the highest P(N|D). Using Bayes rule, P(N|D) = P(D|N) P(N) / P(D). Since P(D) will be the same for all the candidate nets, we are trying to maximize P(D|N) P(N), which is the same as maximizing its logarithm: **log(P(D|N)) + log(P(N))**. Below we consider each of the two terms of this equation. The more data you have, the more important the first term will be compared to the second.

There are different approaches to dealing with the second term **log(P(N))**, which is the prior probability of each net (i.e. how likely you think each net is before seeing any data). One approach is to say that each net is equally likely, in which case the term can simply be ignored, since it will contribute the same amount for each candidate net. Another is to penalize complex nets by saying they are less likely (which is of more value when doing structure learning). Netica bases the prior probability of each net on the experience and probability tables that exist in the net before learning starts, which appears to be a unique and elegant approach. If the net has not been given any such tables, then Netica considers all candidate nets equally likely before seeing any data.

The first term **log(P(D|N))** is known as the net's *log likelihood* , If the data D consists of the n independent cases d_1, d_2, … d_n, then the log likelihood is: $\log(P(D|N)) = \log(P(d_1|N)\, P(d_2|N) \ldots P(d_n|N)) = \log(P(d_1|N)) + \log(P(d_2|N)) + \ldots + \log(P(d_n|N))$. Each of the $\log(P(d_i|N))$ terms is easy to calculate, since the case is simply entered into the net as findings, and Netica's regular inference is used to determine the probability of the findings.

Both EM and gradient descent learning work by an iterative process, in which Netica starts with a candidate net, reports its log likelihood, then processes the entire case set with it to find a better net. By the nature of each algorithm the log likelihood of the new net is always as good as or better than the previous. That process is repeated until the log likelihood numbers are no longer improving enough (according to a tolerance that you can specify), or the desired number of iterations has been reached (also a quantity you can specify). Netica uses a conjugate gradient descent, which performs much better than simple gradient descent.

To understand how each algorithm works, it is best to consult a reference, such as Korb&Nicholson04, Russell&Norvig95 or Neapolitan04. Briefly, EM learning repeatedly takes a Bayes net and uses it to find a better one by doing an expectation (E) step followed by a maximization (M) step. In the E step, it uses regular Bayes net inference with the existing Bayes net to compute the expected value of all the missing data, and then the M step finds the maximum likelihood Bayes net given the now extended data (i.e. original data plus expected value of missing data). Gradient descent learning searches the space of Bayes net parameters by using the negative log likelihood as an objective function it is trying to minimize. Given a Bayes net, it can find a better one by using Bayes net inference to calculate the direction of steepest gradient to know how to change the parameters (i.e. CPTs) to go in the steepest direction of the gradient (i.e. maximum improvement). Actually, it uses a much more efficient approach than always taking the steepest path, by taking into account its previous path, which is why it's called *conjugate* gradient descent. Both algorithms can get stuck in local minima, but in actual practice do quite well, especially the EM algorithm.

Most neural network learning algorithms (such as backpropagation and its improvements) are gradient descent algorithms. That invites a comparison between Bayes net learning and neural net learning, with latent variables corresponding to hidden neurons. In the case of Bayes net learning, there are generally fewer hidden nodes, the learned relationships between the nodes are generally more complex, the result of the learning has a direct physical interpretation (by probability theory) rather than just being black-box type weights, and the result of the learning is more modular (parts can be separated off and combined with other learned structures).

6.2 Experience

There has been considerable controversy over the best way to represent uncertainty, with some of the suggestions being probability, fuzzy logic, belief functions, Dempster-Shafer, etc. Currently probability and fuzzy logic are the most practical methods. Of these two, probability has a much sounder theoretical basis (at least with respect to the way they are actually used). However, a deficiency of using nothing but probability is the inability to represent ignorance in an easy way.

As an example, suppose you had to draw a ball from a bag full of black and white balls and you couldn't tell how many white balls and how many black balls there were in the bag. If you had to supply a probability that you were going to draw a white ball, it would be 0.5 providing you had no additional information.

Contrast this with the case where you can count the balls in the bag beforehand (there are 10 of each), and you will shake the bag before you draw. In this situation the probability of drawing a white ball is 0.5, but whereas in the first case you were in a state of ignorance, now you feel much more informed.

If you needed to do probabilistic inference or solve decision problems as in the previous chapters, then the 0.5 probability would be sufficient in either situation. In both situations you should believe and act as if there was an equal chance of drawing a white or a black ball. So the concept of experience is not required for these types of problems, and you do not have to be able to represent ignorance (ignorance is the endpoint of the experience spectrum). However, for learning and communicating knowledge, it is useful to be able to represent the degree of experience as well as the probability, as we shall see.

If you are going to sequentially draw a number of balls from the bag, then things are different. If you drew 4 white balls in a row, then in the first situation your probability that the next ball will be white should be greater than 0.5, because you are learning (perhaps incorrectly) that there seem to be a lot of white balls. In the second situation your probability of the next ball being white should be less than 0.5, because you know that now there are more black than white balls in the bag (10 black and 6 white).

One way to handle this using just probabilities is to keep track of your beliefs about the ratio of white to black balls in the bag. Then you will have many probabilities, one for each possible ratio. Each of these probabilities will change as you draw a ball, and when you are asked to supply a probability that the next ball drawn will be white, they will all be involved in the calculation. This is sometimes called *second order probabilities*, but here it is really just a probability distribution over possible ratios. If you discretized the possible ratios then it would be easy to set up a Bayes net for this, with the ratio being one of its nodes. That approach works fine for this simple problem, but you can imagine that if you had many interrelated variables, that it could become too cumbersome.

If during the learning we consider the conditional probabilities being learned to be independent of each other, and the prior distribution to be Dirichlet, then we can use beta functions to represent the distributions over "probabilities". Each beta function requires 2 parameters to be fully specified, and Netica uses a probability number and an experience number. This way true Bayesian learning of the probabilities is easy to do, since it is easy to express how the beta function should change to account for a new case (i.e., it is easy to find the posterior beta function, given the prior one and the case). In fact, that is what the simple equation at the end of this section does.

At each node Netica stores one experience number for each possible configuration of states of the parent nodes, and with it a vector of probabilities (one probability for each state of the node). The experience level corresponds roughly to the number of cases that have been seen (normally it is 1 more than the number of cases). This experience has sometimes been called the "estimated sample size" or "ess". To save space, Netica doesn't store experience numbers for nodes that haven't been involved in any learning and haven't had a manual entry of experience.

6.3 Counting Learning

Before learning begins (providing there has been no previous learning or entry of probabilities by an expert) the net starts off in a state of ignorance. All probabilities start as uniform, and experience starts off as the number of states of the node (which is like a single 1 in each unnormalized CPT cell). If you would rather that it started from some different value, then you can use `Node.setExperTable()` to initialize the experience values before learning starts, but then you must also initialize the CPTs to uniform. A different way is to apply a simple correction at the end of the learning, which does the same as Netica Application’s **Table → Harden** function.

For each case to be learned the following is done. Only nodes for which the case supplies a value (finding), and supplies a value for all its parents, have their experience and conditional probabilities modified (i.e., no missing data for that node). Each of these nodes are modified as follows. Only the single experience number, and the single probability vector, for the parent configuration which is consistent with the case is modified. The new experience number (exper') is found from the old (exper) by:

$$\text{exper}' = \text{exper} + \mathtt{degree}$$

where `degree` is the multiplicity of the case (passed to the learning routine). It is normally 1, but is included so that you can make it 2 to learn two identical cases at once, or -1 to "unlearn" a case, etc.

Within the probability vector, the probability for the node state that is consistent with the case is changed from prob_c to prob_c' as follows:

$$\text{prob}_c' = (\text{prob}_c * \text{exper} + \mathtt{degree}) / \text{exper}'$$

The other probabilities in that vector are changed by:

$$\text{prob}_i' = (\text{prob}_i * \text{exper}) / \text{exper}'$$

which will keep the vector normalized (exper' and exper act as the new and old normalization factors).

6.4 How To Do Counting-Learning

There are two ways to do counting-learning from cases: singly (one-by-one) or in batch mode.

Here is how you learn from a single case. If the case is not already in the Bayes net, you enter it into the net as findings (see the "Findings and Cases" chapter). Then `Net.reviseCPTsByFindings()` is called with a list of nodes. Nodes not present in the list passed will not have their probabilities revised, so normally it will be a list of all the nodes in the net. Nodes in the list for which the case provides sufficient data will have their probabilities revised a small amount to account for the case, and their experience levels increased slightly as well.

The batch mode way of revising probabilities does exactly the same thing as the one-by-one way, but for a whole file of cases at once. You call `Net.reviseCPTsByCaseFile()` with the file and the same list of nodes as before, and it does the same thing as the one-by-one method for each of the cases in the file, only much more efficiently than if you were to read in the cases one-by-one and call `Net.reviseCPTsByFindings()` each time. See the "Findings and Cases" chapter for more information on creating a file of cases.

If the case file has a node value that doesn't correspond to any state of that node in the net (e.g. the states of net node 'color' are 'red' and 'green', and the value for color in the case file is 'blue'), then an error will be generated. An exception to this is if one of the states of the net node is called "other". Then the case will be read without error, and the finding for the node will be 'other'.

6.5 Example of Counting-Learning

The program below, LearnCPTs.java, will demonstrate learning from cases. This program can be found in the **examples/** directory of your Netica-J distribution. The program operates by first reading from file a very simple example net (the net that was constructed in the "Building and Saving Nets" chapter), and then duplicates it by making a new net and duplicating all the nodes into it. Next it removes the probabilities and experience from the duplicated nodes with `Node.deleteTables()`. The idea is to relearn approximations of those probabilities by using the case file “ChestClinic.cas” that we created in the last chapter, “Findings and Cases”. In effect, we start with a net that has the structure of ChestClinic.dne, but no probabilities and experience (since they were deleted), and then using a set of cases that match the probability distribution of that net, we will learn a net that should have a similar probability distribution. Of course, the more samples that are in the case file, the better the approximation to the original net.

The program reads all the cases with a single instruction:

```
        reviseCPTsByCaseFile (casefile, learned_nodes, 1.0);
```

If instead we wanted to examine each case, say to exclude outliers, perform calculations on them, or otherwise modify them, we could have looped through the case file, entering each as a finding, and used the instruction

```
        reviseCPTsByFindings (learned_nodes, 1.0);
```

to incrementally adjust the CPTs. The comment section at the bottom of LearnCPTs.java shows you how to use this alternate approach.

Finally, the program concludes by saving the new net to file, so that we can compare it with the old. It will be similar, but the probabilities won't be quite the same. The more cases we put in the case file, the more similar the learned net will be to the original. Of course, in a real application there would be no point in relearning a net which already existed; you would use a case file that had real cases in it. But this demonstration is good to show that the new net comes out similar to the old.

```
/*
 *  LearnCPTs.java
 *
 *  Example use of Netica-J for learning the CPTs of a Bayes net from a file of cases.
 */
import java.io.File;
import norsys.netica.*;

public class LearnCPTs {

  public static void main (String[] args){
      try {
              Environ env = new Environ (null);

              // Read in the net created by the BuildNet.java example program.
              Net net = new Net (new Streamer ("Data Files/ChestClinicBuilt.dne"));
              NodeList nodes = net.getNodes();
              int numNodes = nodes.size();

              // Remove CPTables of nodes in net, so new ones can be learned.
              for (int n = 0;  n < numNodes;  n++){
                      Node node = nodes.getNode (n);
                      node.deleteTables();
              }
              // Read in the case file created by the SimulateCases.java
              // example program, and learn new CPTables.

              Streamer caseFile = new Streamer ("Data Files/ChestClinic.cas");
              net.reviseCPTsByCaseFile (caseFile, nodes, 1.0);

              net.write (new Streamer ("Data Files/Learned_ChestClinic.dne"));


              net.finalize();
      }
      catch (Exception e){
              e.printStackTrace();
      }
  }
}
              /*========================================
               * This alternate way can replace the net.reviseCPTsByCaseFile
               * line above, if you need to filter or adjust individual cases.
               */
              long[ ] casePosn = new long[1];
              casePosn [0] = Net.FIRST_CASE;
              while (true) {
                      net.retractFindings();
                      net.readFindings (casePosn, caseFile, nodes, null, null);
                      if (casePosn[0] == Net.NO_MORE_CASES)  break;

                      net.reviseCPTsByFindings (nodes, 1.0);
                      casePosn[0] = Net.NEXT_CASE;
              }
```

6.6 EM and Gradient Descent Learning

As described in the "Algorithms" section above, counting learning should be done when possible, because it is much faster and simpler, but in cases where there is a substantial amount of uncertain findings, missing data or even variables for which there are no observations (!), EM or gradient descent learning can do amazing things. If you are unfamiliar with the nature of these learning algorithms, you may first want to experiment with them on your data a little using Netica Application, and read its onscreen help about EM learning. The below method can be used to do any of Netica's learning algorithms.

First you create a `Learner` by calling

```
    Learner (int method, String info, Environ env);
```

passing for `method` the algorithm you wish to use (one of `Learner.COUNTING_LEARNING`, `Learner.EM_LEARNING`, or `Learner.GRADIENT_DESCENT_LEARNING`).

If you are doing EM learning or gradient descent learning, then if you wish you can adjust the stopping conditions with:

```
    void setMaxIterations (int maxIterations);
    void setMaxTolerance (double logLikelihoodTolerance);
```

Finally, you perform the learning with:

void learnCPTs(NodeList nodeList, Caseset caseset, double degree)by passing in the nodes whose CPTs you wish to modify, the data as a `Caseset` (see the previous chapter for instructions on creating a `Caseset`), and the degree, which is a multiplier for the frequency of the cases (e.g. `degree = 3` means act as if every case in the Caseset appeared 3 times).

When you are done with the `Learner`, you may reduce the resources required by calling:

```
    void finalize();
```

Here is a small code example: (for another, see "Connecting with a Database" in the previous chapter)

```
    Streamer      netfile   = new Streamer ("ParameterlessNet.dne");
```

```
Streamer        datafile = new Streamer ("Data.cas");
Net       net          = new Net (netfile, env, "no_visual");
NodeList        nodes    = net.getNodes();
Caseset         cases    = new Caseset();
Learner learner  = new Learner (Learner.EM_LEARNING);
learner.setMaxTolerance (1e-5);
cases.addCases (datafile, 1.0, null);
learner.learnCPTs (nodes, cases, 1.0);
learner.finalize();
cases.finalize();
```

6.7 Fading

When a Bayes net is supposed to capture relationships between variables in a world which is constantly changing, it is useful to treat more recent cases with a higher weight than older ones. An example might be an adaptive Bayes net which is constantly receiving new cases and doing inferences while it slowly changes to match a changing world.

Netica achieves this partial forgetting of the past by using *fading*. Every so often you call `Node.fadeCPTable()`, passing it a `degree` between 0 and 1, and it will reduce the experience and smooth the probabilities of the node by an amount dictated by the degree. A degree of 0 has no effect, while a degree of 1 does complete forgetting, resulting in uniform distributions with no experience. Calling `fadeCPTable()` once with `degree = 1-a`, and again with `degree = 1-b`, is equivalent to a single call with `degree = 1-ab`.

During fading, each of the probabilities in the node's conditional probability table is modified as follows (where prob and exper are the old values of probability and experience, and prob' and exper' are the new values):

prob' = normalize (prob * exper * (1 - *degree*) + *degree* * BaseExper)

where BaseExper is normally 1 (see section 7.1). exper' is obtained as the normalization factor from above (remember that there is one experience number per vector of probabilities). So:

prob' * exper' = prob * exper * (1 - *degree*) + *degree* * BaseExper

When learning in a changing environment, you would normally call `fadeCPTable()` every once in a while so that what has been recently learned is more strongly weighted than what was learned long ago. If an occurrence time for each case is known, and the cases are learned sequentially through time, then the amount of fading to be done is: *degree* $= 1 - r^{\Delta t}$ where Δt is the amount of time since the last fading was done, and r is a number less than, but close to, 1 and depends on the units of time and how quickly the environment is changing. Different nodes may require different values of r. See the example in the description of `fadeCPTable()` in the "Function Reference" chapter.

6.8 Performance Testing a Net using Real-World Data

After you have built a Bayes net, either by hand based on the judgments of an expert, or automatically by learning it from data, you may want to test how effective it is. That can be done by using a set of cases gathered from the real-world or from the environment in which the net will be used. You should use a different data set than was used to build the Bayes net, otherwise your net may score too high, since it will probably test slightly better on the training set than other sets. A common approach when learning a Bayes net from data, is at the beginning to set aside a certain percentage of the (well shuffled) cases to be used for later testing. These are known as the *test cases* (or “test data”), as opposed to the *training cases* (or “training data”).

The first step is to identify the variables (i.e. nodes) that Netica won’t know the value of during actual usage of the net. For example, if the net is to be used as a classifier, then during usage Netica won’t know the value of the class variable. If the net is to be used for prediction, then Netica won’t know the values of the variables that are yet to occur in time. If the net is to be used for diagnosis, Netica won’t know what the actual faults or internal states are during the diagnosis. The variables (i.e. nodes) that will not be known during usage are called the *unobserved nodes*.

The next step is to choose which of the unobserved nodes you want to test the Bayes net’s ability on. These are the nodes that statistics will be generated for, and are called the *test nodes*.

In the code, you first call `new NetTester()`, passing in a list of the test nodes. If there are some unobserved nodes that aren’t already in the test nodes, you pass in a list of them as the `unobsv_nodes` argument (which can also include any of the test nodes if you want – it makes no difference since Netica will take as the unobserved nodes the union of the two lists).

Then you call `Tester.testWithCaseset()`, passing in the case file containing the real-world data. Netica will go through the case file, processing the cases one-by-one. Netica first reads in a case, except for findings for the unobserved nodes. It then does belief updating to generate beliefs for each of the test nodes, and checks those beliefs against the true value for those nodes as supplied by the case file (if they are supplied for that case). It accumulates all the comparisons into summary statistics. If you want, you can call `testWithCaseset()` several times with different files to generate statistics for the combined data set.

Finally, you call functions to retrieve the actual performance statistics you desire. You can obtain the error rate with `NetTester.getErrorRate()`, the logarithmic loss with `NetTester.getLogLoss()`, the quadratic loss with `NetTester.getQuadraticLoss()` and the whole confusion matrix with `NetTester.getConfusion()`. Be sure to see the function documentation for each of these functions, and `NetTester()` and `NetTester.testWithCaseset()`, for more details on the whole process. Also, you can contact Norsys for a document with more information on what the various measures mean.

Here is some example program that rates the toy Bayes net "ChestClinic", to test the "Cancer" node diagnosis assuming that the other disease nodes (Tuberculosis, Bronchitis, TbOrCa) are also unobserved nodes:

```
/*
 *  TestNet.java
 *
 *  Example use of Netica-J for testing the performance of
 *  a learned net with the net-tester tool.
 */
import java.io.File;
import norsys.netica.*;

public class TestNet {

  public static void main (String[] args){
  try {
      Environ env      = new Environ (null);
      Net      net     = new Net (new Streamer ("Data Files/ChestClinic.dne"));

      Node tuberculosis        = net.getNode ("Tuberculosis");
      Node cancer              = net.getNode ("Cancer");
      Node tbOrCa              = net.getNode ("TbOrCa");
      Node bronchitis          = net.getNode ("Bronchitis");

      //  The observed nodes are typically the factors known during diagnosis:
      NodeList testNodes = new NodeList (net);
      testNodes.add (cancer);

      //  The unobserved nodes are typically the factors not known during diagnosis:
      NodeList unobsvNodes = new NodeList (net);
      unobsvNodes.add (bronchitis);
      unobsvNodes.add (tuberculosis);
```

```
        unobsvNodes.add (tbOrCa);
        net.retractFindings();    // IMPORTANT: Otherwise any findings will be part of tests !!
        net.compile();

        NetTester tester          = new NetTester (testNodes, unobsvNodes, -1);
        Streamer  inStream        = new Streamer ("Data Files/ChestClinic.cas");
        Caseset  testCases        = new Caseset();

        testCases.addCases (inStream, 1.0, null);
        tester.testWithCaseset (testCases);

        printConfusionMatrix (tester, cancer);
        System.out.println ("Error rate for "          + cancer.getName() + " = " + tester.getErrorRate
(cancer));
        System.out.println ("Logarithmic loss for " + cancer.getName() + " = " + tester.getLogLoss
(cancer));

        // the following are not strictly necessary, but a good habit
        testCases.finalize();
        tester.finalize();
        net.finalize();
    }

    catch (Exception e){
        e.printStackTrace();
    }
  }


    /*
     *  Print a confusion matrix table
     */
    public static void printConfusionMatrix (NetTester nt, Node node) throws NeticaException {
        int numStates = node.getNumStates();
        System.out.println ("\nConfusion matrix for " + node.getName() + ":");

        for (int i = 0;  i < numStates;  ++i)
                System.out.print ("\t" + node.state(i).getName());
        System.out.println ("\tActual");

        for (int a = 0;  a < numStates;  ++a){
                for (int p = 0;  p < numStates;  ++p)
```

```
                    System.out.print ("\t" + (int) (nt.getConfusion (node, p, a)));
                System.out.println ("\t" + node.state(a).getName());
    }
  }
}
```

And this is the output it produces:

```
Confusion matrix for Cancer:
        present absent  Actual
        9       2       present
        4       185     absent
Error rate for Cancer = 0.03
Logarithmic loss for Cancer = 0.08219048904200114
```

7 Modifying Nets

A common scenario is that you've built a Bayes net using Netica Application (or Netica API, as described in the "Building and Saving Nets" chapter) and saved the file. Now your program uses Netica API to read the net file and use it to solve problems. Each of the problems is a little bit different, and it's not enough to just enter different findings, you need to modify the net itself. Perhaps it's a small change like altering the CPT tables, adding new states to a node, changing utilities or converting decision nodes to nature nodes. Or maybe it is a major operation like taking several net fragments from different nets and stitching them together to make a new net for the particular problem at hand. This chapter discusses some ways to modify a net in place, and then in the section "Node Libraries" it discusses how to create "libraries" of nodes or network fragments, and then stitch them together on the fly to create models. Finally it discusses transforms that may be done on a Bayes net to remove nodes or reverse the direction of links while maintaining the overall probabilistic relationship between the remaining nodes.

7.1 Common Modifications

Most of the functions introduced previously for building a Bayes net can also be used to modify it. For instance, `new Node()` and `Node.addLink()` can introduce new variables or dependencies, and `Node.delete()` and `Node.deleteLink()` can remove them.

Almost every property of nets and nodes can be altered. Even decision nodes can be converted to nature nodes (`Node.setKind()`), or vice versa, without losing their CPT tables or other properties. That can be useful to model situations with multiple agents, where the nodes that are the decisions of one agent, are nature nodes to the other agents. First the optimal decisions are found for the first agent, and then those decision nodes are converted to nature nodes when finding the optimal decisions for the next agent.

When adapting a net to a new environment, states can be added (`Node.addStates()`), removed (`Node.state().delete()`), or the order of the states may be changed (`Node.reorderStates()`). In each case the tables of the nodes being changed, and the tables of their children, will be appropriately modified.

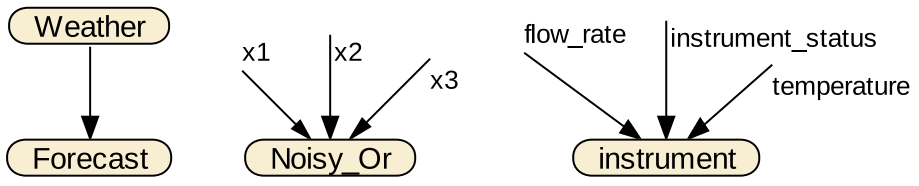
The node tables themselves may be modified. Perhaps CPTs need to be changed based on frequency data that is calculated externally. Or perhaps the utility tables of utility nodes are modified based on preference information about a particular end-user, and then new optimal decisions found. The most common change to CPT tables is to adjust them to take into account case data from the world, and that is covered in detail in the "Learning From Case Data" chapter. Tables may be changed with: `Node.setCPTable()`, `Node.setStateFuncTable()`, `Node.setRealFuncTable()`, `Node.equationToTable()` and `Node.deleteTables()`.

An advanced program may wish to lay out the visual positions of all the nodes, so that when the Bayes net file is read by Netica Application, they will be displayed in the desired layout. Or perhaps choose in which style to display each node (e.g. Belief Bars, Labeled Box or Hidden). The functions to use are: `Node.visual().setPosition()` and `Node.visual().setStyle()`.

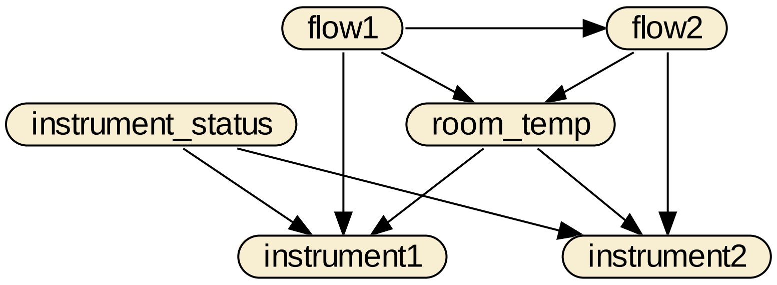
7.1 Node Libraries

Often the probabilistic relation between a node and its parents represents a small piece of local knowledge which may be applicable in a number of different nets to be used in different situations. That relation may have been learned from data, or entered by an expert. Each new net it is placed in captures the global relations between such local pieces of knowledge, and belief updating combines the local and global knowledge with the details of some particular case.

For example, suppose that you made a simple net consisting of a node called Weather connected to a node called Forecast. The link between them could go either way, since we can't really capture causation (they are both caused by other variables, like the previous weather), but say you put the link from weather to forecast because often it's better to put links from more immutable to less immutable variables. Each day you revised its probabilities so that eventually it accurately captured the probabilistic relationship between the morning weather forecast and the weather for that day. Then you could put it in a library to later graft into nets for inference involving the weather and its forecast, such as the decision problem discussed in the "Decision Nets" chapter.



As another example, suppose you have a device for measuring the flow rate in a pipe. This sensor will produce biased readings depending on the ambient temperature, and it can break in a few different ways, each of them producing wrong or inaccurate readings. You can model the sensor with a 4 node net, 1 node for the reading on the sensor, and 3 parent nodes corresponding to: actual flow rate, ambient temperature, and sensor status (okay, broken_1, broken_2, etc.). You enter the probabilistic relationship, and then you disconnect the node from its parents and place it in a library (so it appears as in the above diagram; disconnection and grafting are explained below). Later, if you have a net to model a situation in which you have made two measurements with the device, you just duplicate the device characteristics node from the library twice into the new net, and graft it to the appropriate nodes in that net (see diagram below). Note that if the ambient temperature could be different between the two measurements, then the room_temp node would appear as two connected nodes, similar to the flow nodes, and the same goes for the instrument_status node if the device may have broken between measurements. Automating the process of net construction for new situations is an area of active research, with dynamic Bayes nets, templates and graph grammars being some of the methods used.



Netica makes it easy to maintain libraries of disconnected nodes and subnets. To make a new library, just use `new Net()`. Nodes and subnets can be copied to it using `Net.copyNodes()`, which can transfer material from one net to another, and also copies all the links between nodes in a subnet. When a node is being duplicated, but one of its parents isn't, then `Net.copyNodes()` will give the duplicated node a *disconnected link* where that parent was. This is a link which only has a place-holder for a parent, and is meant to be reconnected to another node before being used for inference. In this way the conditional probability relationship that the node had with its parents is not lost. The disconnected link is given the name of the parent it once had if the link is not already named. If you ever want to check whether a link is disconnected, use `Node.getKind()`.

When you want to use something in the library, you call `Net.copyNodes()` again, this time to duplicate from the library into the new net. Then you connect up any disconnected links with `Node.switchParent()`, which will switch out the parent place-holder, and switch in the new parent.

Below is a code example for the flow measuring instrument described earlier:

```
Net net          = new Net();
Node flow        = new Node ("flow_rate",          0, net);
Node temp        = new Node ("temperature",        0, net);
Node broken      = new Node ("instrument_status",          5, net);
Node instrument        = new Node ("instrument",          0, net);
```

```
instrument.addLink (flow);
instrument.addLink (temp);
instrument.addLink (broken);

// ...
// … Put here: Build probabilistic relation for node 'instrument',
// … either by learning from cases, or entry by an expert.
// ...

// The below will put a copy of the 'instrument' node,
// disconnected from its parents, into the library.
// Its disconnected link names will be those of the old parents.

Net libnet = new Net();
duplicate (instrument, libnet);       // see definition below
libnet.write (new Streamer ("Library.dnet"));

net.finalize();
libnet.finalize();


// This is a static variant of NodeEx.duplicate(), used above


public static Node duplicate (Node oldNode, Net newNet) throws NeticaException {
    NodeList nodes = new NodeList (oldNode.getNet());
    nodes.add (oldNode);
    NodeList newNodes = newNet.copyNodes (nodes);
    return newNodes.getNode (0);
}
```

Now the library is constructed and saved to file, with *instrument* as the only node in it.

At a later session, we use the library to construct *appnet*, an application net in which the instrument is used to measure *flow1* and *flow2*, which are in the same room at the same temperature:

```
Net appnet     = new Net();
Node flow1     = new Node ("flow1",      0, appnet);
Node flow2     = new Node ("flow2",      0, appnet);
Node rtemp     = new Node ("room_temp",        0, appnet);
Node status    = new Node ("instrument_status",          0, appnet);


// ...
// … Put here: Build rest of application net.
// … Connect up nodes flow1, flow2, rtemp, and status.
// … Add probabilistic relations for flow1, flow2, rtemp, and status.
// ...

// The below will get 2 copies of the instrument node from the library,
// and put them in the application net.

libnet = new Net (new Streamer ("Library.dnet"));
Node instrument1 = duplicate (libnet.getNode ("instrument"), appnet);
Node instrument2 = duplicate (libnet.getNode ("instrument"), appnet);

// The below will graft them to the other nodes in the application net.

instrument1.switchParent (instrument1.getInputIndex ("flow_rate"), flow1);
instrument1.switchParent (instrument1.getInputIndex ("temperature"), rtemp);
instrument1.switchParent (instrument1.getInputIndex ("instrument_status"), status);
instrument2.switchParent (instrument2.getInputIndex ("flow_rate"), flow2);
instrument2.switchParent (instrument2.getInputIndex ("temperature"), rtemp);
instrument2.switchParent (instrument2.getInputIndex ("instrument_status"), status);
```

Now the application net *appnet* is ready for probabilistic inference. Perhaps we have positive findings for the instrument node (i.e. what we read from its dial), and we use them to determine flows and their uncertainties in a way that properly accounts for random (uncorrelated) and systematic (correlated) errors, as well as all the background knowledge about the situation.

7.2 Net Reduction

Suppose you have a large net that has been constructed over time by a combination of expert assistance and probability learning. It shows the relationships between hundreds of variables, and contains much valuable information that could be used in a number of different applications.

Now you want to use it in an application where only 10 of the variables are of interest to you. In every query of the new application, four of them will always have the same value. For instance, one of the nodes in the original net might by Gender, and in the restricted application the net will only be used for females, so we would like to enter a permanent finding of 'female' for the node Gender. These nodes are called *context nodes*. In each of the queries, you will be receiving new findings for 4 other nodes, and then you want the resulting beliefs of the remaining 2. The nodes that will have new findings are called *findings nodes*, and those whose beliefs you will want are called *target nodes*. The hundreds of other nodes in the net might be involved in intermediate calculations, but you don't care about their values explicitly.

You can simplify the large net down to one with just 6 nodes using `Net.absorbNodes()`. First enter the permanent findings for the context nodes. Then make a list of all the nodes except the findings nodes and the target nodes, and pass it to `Net.absorbNodes()`. The resulting 6 node net will give the same inference results as the original large one, for the restricted queries you will be making. If you are guaranteed that there will always be findings for every findings node, then you can then further simplify things by removing any links that go from findings node P to findings node C, providing C does not have a target node as an ancestor. This means that if you use `Node.reverseLink()` to make all the findings nodes ancestors of all the target nodes, then you can remove all the links between the findings nodes. Any findings node that is left completely disconnected by this operation is irrelevant to the query. And now you can examine the conditional probability relations of the target nodes to see directly how they depend on the findings. You may just be able to look up the desired probabilities without doing belief updating at all!

There is a danger to keep in mind. Even though the reduced net has fewer nodes than the original, it may actually be more complex, if many links were added by `Net.absorbNodes()` or `Node.reverseLink()` (remember that the size of a node's conditional probability table can be exponential in its number of parents). Generally speaking, absorbing out context nodes (i.e. nodes with findings entered) which have many ancestor nodes results in the worst increase in complexity. The next worst is absorbing out non-context nodes (i.e. nodes with no findings) which have many descendant nodes. Absorbing out context nodes with no ancestors, or non-context nodes with no descendants, will not add any links. Of course, if the number of target and findings nodes is *very* small, the resulting net must be simpler, although the transformations to generate it might temporarily require a lot of memory.

7.3 Probabilistic Inference by Node Absorption

From the previous section you may have realized it is possible to do probabilistic inference using node absorption, by entering all the findings, and then absorbing all the nodes except for a single target node. The resulting probability distribution for that node can be obtained with `Node.getCPTable()`, and it will be a single belief vector (because the node won't have any parents), that is the same as the belief vector that would be obtained by compiling the Bayes net, and obtaining the beliefs via belief updating with `Node.getBeliefs()`.

The question is: which method is faster? If you need the beliefs for all the nodes, then you would have to repeat the absorbing-node method for each of the nodes (duplicating the net each time, since it is destroyed in the process), and so it will usually be far slower. But if you only need the beliefs of one node, for one set of findings, and there are many nodes in the net that are irrelevant to the particular query, then the node absorption method can be much faster (providing a good “elimination order” for absorbing the nodes is used).

It should be mentioned that node absorption will also work with decision nets (see the "Decision Nets" chapter) to find optimal decisions. When a decision node is absorbed it is not removed from the net; instead it is completely disconnected and its decision table set to the optimal decision function.

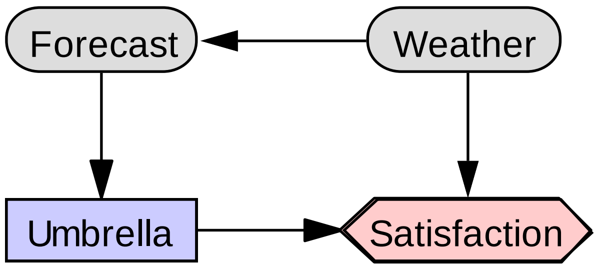
When using `Net.absorbNodes()` for decision nets, the decision nodes must have no-forgetting links, and if the list of nodes to absorb does not include all the nodes in the net, it must consist of a descendant subnet (see Shachter86, Shachter88 and Shachter89 for definitions and details of the algorithm used). If there are missing no-forgetting links or missing descendants in the list of nodes to absorb, then `Net.absorbNodes()` will absorb as many nodes as possible, then generate an error explaining exactly why it was impossible to proceed.

8 Decision Nets

Chapter 3 was about probabilistic inference using a Bayes net, where the purpose was to determine new beliefs (in the form of probabilities) as observations were made or facts gathered. A Bayes net is composed only of *nature nodes* (which may be “chance” nodes or “deterministic” nodes). By adding *decision nodes* and *utility nodes* (also known as “value” nodes) to a Bayes net, we obtain a *decision net* (also known as an “influence diagram”). Decision nets can be used to find the optimal decisions which will maximize expected utility.

First, we give a small warning. You may find it overly challenging if your first usage of Netica API is to build a large decision net with multiple decisions, and you haven’t had related experience. People usually start by building Bayes nets, then nets with just one decision, and after they have some experience, nets with a few decisions. Also, they usually have some experience working with nets using Netica Application, or a similar program, before using Netica API for complex decision nets.

As an example decision net, let's consider a very tiny one from Ross Shachter known as "Umbrella". It has 2 *nature nodes* representing the weather Forecast in the morning (sunny, cloudy or rainy), and what the Weather actually turns out to be during the day (sunshine or rain), a *decision node* of whether or not to take an Umbrella, and a *utility node* that measures our level of Satisfaction. There is a link from Weather to Forecast capturing the believed correlation between the two (perhaps based on previous observations).



There is a link from Forecast to Umbrella indicating that we will know the forecast when we make the decision. It is always the case that links entering a decision node indicate what variables will be known at the time of the decision. What we wish to find in solving the decision problem is a function providing the value of the decision node for each possible setting of its parent nodes, which maximizes the expected value of the utility nodes. In other words, we find a contingent plan that tells which decision to make for each possible set of observations that will be made when it is time to act on the decision. There is no link from Weather to Umbrella; if we knew for certain what the weather was going to be, it would be easy to decide whether or not to take the umbrella.

There are links from Weather and Umbrella to Satisfaction, capturing the idea that I am most happy when it is sunny and I don't take my umbrella (utility = 100), next most when it is raining and I take my umbrella (utility = 70). I hate carrying my umbrella on a sunny day (utility = 20), but am most unhappy if it is raining and I don't have one (utility = 0).

8.1 Programming Example

Below is a listing of the program, MakeDecision.java, which build this decision net in memory, and then solves it (i.e., finds the optimal decisions). This program can be found in the **examples/** directory of your Netica-C distribution. Much of it is very similar to building a Bayes net (see the chapter "Building and Saving Nets" for explanations of those parts). We will discuss the things new to this example.

When a node is first created with `new Node()`, it starts off as a nature node. Here we change Umbrella into a decision node, and Satisfaction into a utility node using `Node.setKind()`. `Node()` is passed the number of states of the node, and in this example, as well as having 2-state nodes, there is also a 3-state node, and a continuous node (indicated by passing 0 for number of states). Utility nodes are always continuous deterministic nodes. We use `Node.setRealFuncTable()` to build up the relations of a deterministic node instead of `Node.setCPTable()`, but it works in a similar fashion.

```
/*
 *  MakeDecision.java
 *
 *  Example use of Netica-J to build a decision net and choose an optimal decision with it.
 */
import norsys.netica.*;
import norsys.neticaEx.aliases.Node;

public class MakeDecision {

  public static void main (String[] args){
    try {
        Node.setConstructorClass ("norsys.neticaEx.aliases.Node");
        Environ env = new Environ (null);
        Net net = new Net();
```

```
Node weather          = new Node ("Weather","sunshine,rain", net);
Node forecast         = new Node ("Forecast",        "sunny,cloudy,rainy", net);
Node umbrella         = new Node ("Umbrella""take_umbrella, dont_take_umbrella", net);
Node satisfaction     = new Node ("Satisfaction",      0, net);     // 0 for continuous node

umbrella.setKind (Node.DECISION_NODE);
satisfaction.setKind (Node.UTILITY_NODE);

forecast.addLink (weather);
umbrella.addLink (forecast);
satisfaction.addLink (weather);
satisfaction.addLink (umbrella);

weather.setCPTable (0.7, 0.3);

//                       forecast
//                       weather   |sunny   cloudy  rainy
forecast.setCPTable ("sunshine",          0.7,     0.2,     0.1);
forecast.setCPTable ("rain",      0.15,   0.25,    0.6);
//                                  weather  umbrella        utility
satisfaction.setRealFuncTable ("sunshine,        take_umbrella", 20.0);
satisfaction.setRealFuncTable ("sunshine,        dont_take_umbrella",    100.0);
satisfaction.setRealFuncTable ("rain,      take_umbrella", 70.0);
satisfaction.setRealFuncTable ("rain,      dont_take_umbrella",    0.0);

net.compile();

//-----   1st type of usage:  To get the expected utilities, given the current findings

forecast.finding().enterState ("sunny");

 float[] utils = umbrella.getExpectedUtils();  // returns expected utilities, given current findings

System.out.print   ("If the forecast is sunny,  ");
System.out.println ("the expected utility of "       + umbrella.state(0) + " is " + utils[0] +
                    ", of "            + umbrella.state(1) + " is " + utils[1]);

net.retractFindings();
forecast.finding().enterState ("cloudy");
utils = umbrella.getExpectedUtils();
```

```
            System.out.print   ("If the forecast is cloudy, ");
            System.out.println ("the expected utility of "        + umbrella.state(0) + " is " + utils[0] +
                                ", of "                            + umbrella.state(1) + " is " + utils[1] + "\n");

            //-----   2nd type of usage:  To get the optimal decision table

            net.retractFindings();
            umbrella.getExpectedUtils();       // causes Netica to recompute decision tables,
                        // given current findings (which in this case are no findings)
            for (int fs = 0;  fs < forecast.getNumStates();  ++fs){
                        int[] parStates = new int[1];
                        parStates[0] = fs;          // forecast is the parent of umbrella
                        int decision = umbrella.getStateFuncTable (parStates, null) [0];
                        System.out.println ("If the forecast is "     + forecast.state (fs) +
                                            ",\tthe best decision is "        + umbrella.state (decision));
            }
            net.finalize();      // free resources immediately and safely; not necessary, but a good habit
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Once the net is built, the program calls `Net.compile()`, and then `Node.getExpectedUtils()` to force a belief updating, which will build a new deterministic table for each decision node. Each deterministic table represents a function which provides a value for the node for each possible configuration of parent values. Since the links into a decision node indicate what the decision maker will know when he is about to make the decision, this function provides a decision for each possible information state. The decision functions Netica finds are the ones that provide the highest expected value of the utility node (or the sum of the utility nodes if there are more than one). The above program uses `Node.getStateFuncTable()` to access this decision function, and prints out the following:

```
If the forecast is sunny,  the expected utility of take_umbrella is 24.205606,
of dont_take_umbrella is 91.58878
If the forecast is cloudy, the expected utility of take_umbrella is 37.44186,
of dont_take_umbrella is 65.11628

If the forecast is sunny,    the best decision is dont_take_umbrella
If the forecast is cloudy,   the best decision is dont_take_umbrella
If the forecast is rainy,    the best decision is take_umbrella
```

Note that `Node.getExpectedUtils()` or `Node.getBeliefs()` must be called before `Node.getStateFuncTable()` to have Netica build the decision table (and again after entering findings if you want it optimized for the new findings).

For more information on decision nets in general, and using Netica to work with them, see the onscreen help system of Netica Application (and there is also some information in the tutorial at the Norsys website).

9 Drawing Nodes and Nets

Netica-J includes several Java-SWING components for displaying Netica nets and nodes. The color, layout, and general appearance of the displayed components are very similar to the style used in the Netica Application program (see http://www.norsys.com/netica.html for details, to purchase, or to download a size-restricted free version.)

With the exception of norsys.netica.VisualNode, all of the classes relevant to graphical display can be found in the **norsys.netica.gui** package. The two most important classes in this package are NodePanel and NetPanel. Each of these is a javax.swing.JPanel that displays assorted graphical components (e.g., JLabels) within itself. You typically have full access to these subcomponents, and so can change colors, fonts, and borders, attach event listeners, set visibility, etc., just as you would with any AWT/SWING component.

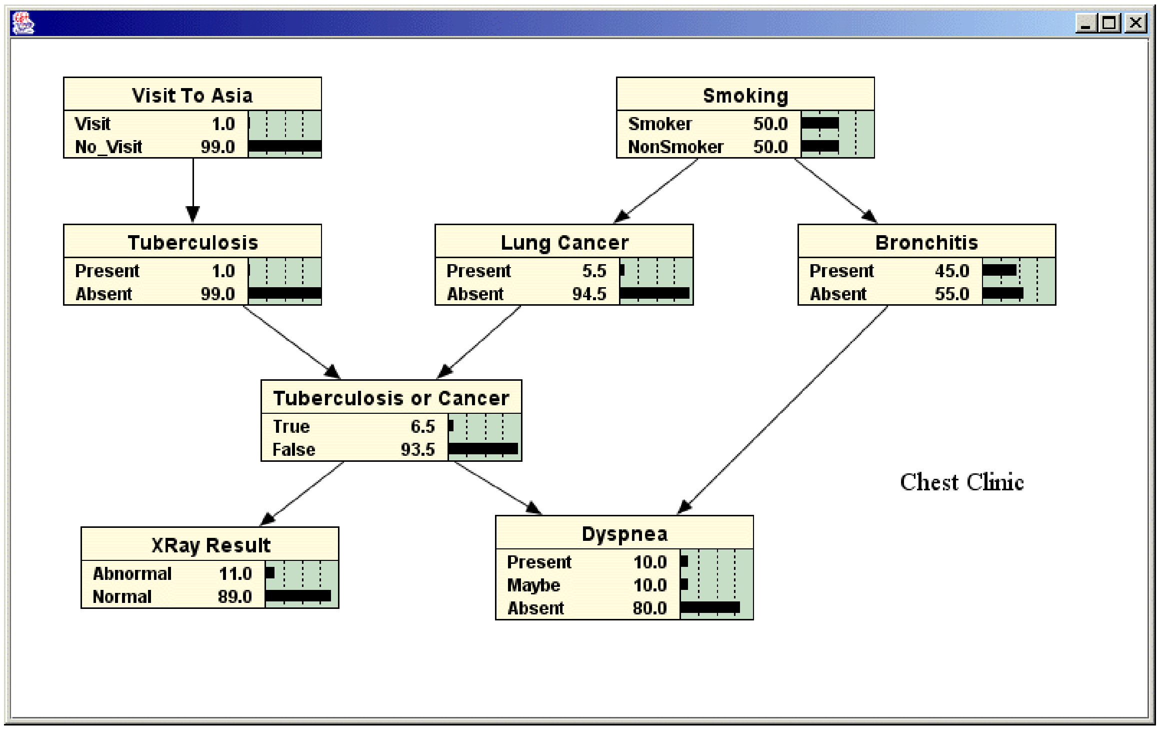
The philosophy behind the development of the **gui** package is to enable you to very easily and rapidly add graphical displays to your Netica-J programs. For instance, the following tiny program is all that is needed to display a net:

```
import norsys.netica.*;
import norsys.netica.gui.*;
import javax.swing.*;

class DrawNet extends JFrame {

  public DrawNet (String netName) throws Exception  {
      Net net = new Net (new Streamer (netName));
      net.compile();    // optional
      NetPanel netPanel = new NetPanel (net, NodePanel.NODE_STYLE_AUTO_SELECT);
      getContentPane().add (new JScrollPane (netPanel));   // adds the NetPanel to ourself
      setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
      setSize (800, 500);   // or supply getPreferredSize();
```

```
        show();
    }

    public static void main (String[] args){
        try {
                Environ env = new Environ (null);
                DrawNet dn = new DrawNet (args[0]);
        }
        catch (Exception e){
                e.printStackTrace();
        }
    }
}
```

You call the program with: `java DrawNet SomeNet.dne` (or .neta) and it will draw the net in a fashion similar to the way that Netica Application does, with the nature nodes drawn using the popular "belief-bar" style. If you'd prefer a different style, then simply replace NodePanel.NODE_STYLE_AUTO_SELECT in the NetPanel constructor call with the style of your choice. Here is the window that the command

`java DrawNet "Data Files/ChestClinic_WithVisuals.dne"` creates:



9.1 Netica Net Visual Properties and the gui Package

Netica files (.dne and .neta files) may optionally contain visual information as to the size, position, and visual style of the nodes they contain. This visual information is used both by the Netica Application and the Netica-J gui package in order to help decide where and how to display the net components. The general policy of a Netica graphical application is to attempt to make sense of and employ the visual information that is present, but if it is not able to do so, it will ignore that information.

9.2 Node Position

If a Netica file has been saved without visual information, then all of the Nodes are by default given a position of (0,0). To add position information, you may use the Netica Application to display the net and then position the Nodes as desired, or from within the Java API you may use Node.visual().setPosition().

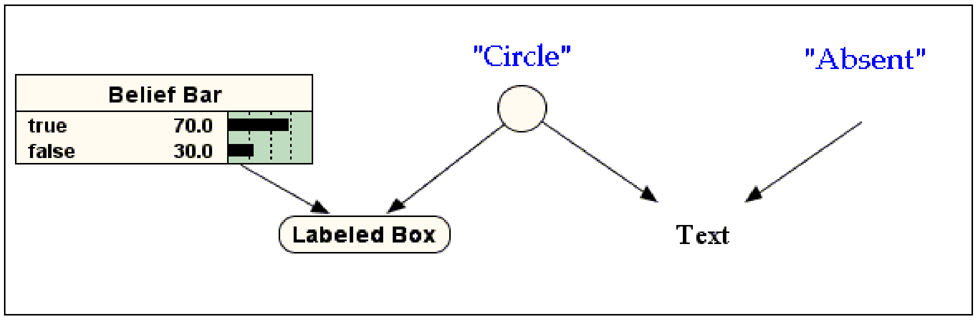
Once a NodePanel has been created, it is a Java component that can be moved anywhere (e.g., using java.awt.Component.setLocation()) without affecting the Node.visual() position data. If you want to keep the displayed position in sync with the Netica visual() position, you must either manage this yourself, or else confine yourself to moving the component by calling NodePanel.moveBy().

9.3 Node Style

If a Node does not contain any style information (introduced using Netica Application or by calling Node.visual().setStyle(), then Netica-J will apply certain default styles when creating NodePanels (NodePanel.createNodePanel()) for that node. Conversely, if the node does contain style information, that information will be treated as the preferred style when Netica-J is given an option in deciding what variety of NodePanel to create.

9.4 Drawing Nodes

You do not require the NetPanel class to draw nodes. You may draw the nodes directly using any of several NodePanels that are supplied:



The NodePanel class itself is an abstract base class that manages a number of common functions of all NodePanels, such as hooking up to the Netica Node that is represented and discovering it's title (or name, if it lacks a title), managing event listeners, and managing display modes (hi-lighted, normal, or grayed).

9.5 Event Handling

Besides being standard JComponents and hence being able to partake of all the standard Java AWT/SWING events, NetPanel and NodePanel objects are also norsys.netica.gui.RecursingEventListeners. The RecursingEventListener interface provides two very convenient methods: **addListenerToAllComponents** (java.util.EventListener eventListener) and **removeListenerFromAllComponents** (java.util.EventListener eventListener). These will recursively run through all the subcomponents of the NetPanel or NodePanel and attach the EventListener to those subcomponents. Thus, with a single command, you can add an event listener to all the components within a NetPanel.

9.6 NetViewer

Included in the **examples/** directory of this distribution is a reasonably sophisticated program, NetViewer.java. This program allows you to select a net from a list, whereupon it draws the selected net, and then allows you to edit the net and to enter finding information as well. It illustrates how to attach mouse events to nodes, to parts of nodes (e.g., the belief-bars rows), and even to links. If you need to build a more sophisticated graphical application for selecting nodes, entering findings, and such, you may wish to use this program as a starting point.

9.7 Miscellaneous Useful Features

The NetPanel class supports the concept of a selection set, which is just a NodeList of nodes in the "hilited state" (see NodePanel.getDisplayMode). Also, you may choose to display a subset of the nodes in a net using the setSubnet method.

9.8 Feedback Wanted

We would appreciate hearing whether you find the gui package useful and how you might like to see it evolve.

10 Special Topics

10.1 Node Lists and Node-sets

Many operations in intelligent computing require working with lists of variables, and when using Bayes nets that means working with lists of nodes, so it is not surprising that many Netica functions take node lists as arguments. At first it might not be clear why Netica has two classes to work with sets of nodes, but their purposes are very different. Netica's workhorse node list class is `NodeList`, whose main purpose is to pass an ordered list of nodes to Netica-J, or retrieve such a list from Netica-J. It extends java.util.Vector, but requires that all of its elements be `Nodes`, and that they all come from the same `Net`.

The other class is `Nodeset`, which is just meant to access node-sets defined in Netica Application. Netica Application has a convenient way of using the GUI to define sets of nodes, give them a color, and do operations on a whole set at once. It displays each node with the color of its node-set (since a node may be a member of several node-sets, they have a priority order for determining color), and all the node-set information is saved in the file with the net. Since Netica-J works with the net files, node-sets provide a convenient way to pass sets of nodes back and forth between Netica Application and Netica-J. Depending on their name, some of the sets have special meaning to Netica; others have meaning only to the developer.

`NodeList`s have the following constructor, copy constructor, access function and finalizer:

NodeList.NodeList (Net parentNet)
Constructs an empty NodeList (initial capacity is 100).

NodeList (NodeList nodeList)
Copy constructs a new NodeList from an existing NodeList. The list is duplicated, but not the nodes themselves.

Node NodeList.getNode (int index)
Returns the Nth node of a list (the first node is numbered 0).

void NodeList.finalize ()
Frees the memory used by a list of nodes. Not necessary to call, since Java will do it automatically.

As well they inherit the usual functions from java's Vector:

void NodeList.add (int index, Node element)
Node NodeList.remove(int index)
Node NodeList.set (int index, Node element)
int NodeList.indexOf (Node elem, int index)
int NodeList.size ()
void NodeList.clear ()

Here are some of the most basic functions using `NodeList`s:

NodeList Net.getNodes ( )
Returns a list of all the nodes in the net

NodeList Node.getParents ( )
Returns a list of the parents of a node

NodeList Node.getChildren ( )
Returns a list of the children of a node

void Node.getRelatedNodes (NodeList relatedNodes, String relation)
Finds all the nodes that bear a given relationship (such as D-connected, Markov blanket, ancestors, children, etc.) with a given node.

void Net.getRelatedNodes (NodeList relatedNodes, String relation, NodeList nodeList)
Finds the nodes that bear a given relationship with a given set of nodes.

Static int NodeList.mapStateList (int[] srcStates, NodeList srcNodes, NodeList destNodes)
Returns an array of the same states as srcStates (which is in same order as srcNodes), but in the order of destNodes.

And here are all the Nodeset functions:

void Node.addToNodeset (String nodeset)
Adds this node to the node-set of the given name

void Node.removeFromNodeset (String nodeset)
Removes this node from the node-set of the given name

boolean Node.isInNodeset (String nodeset)
Returns whether this node is a member of the given node-set

String Net.getAllNodesets (boolean includeSystem)
Returns a list of all node-sets defined for this net, separated by commas, in priority order. The argument indicates whether to include Netica's built-in nodes-sets, otherwise it only puts user defined ones.

void Net.reorderNodesets (String nodesetOrder)
Re-orders the priority of the node-sets as requested (priority is used to choose display color). Any node-sets contained in the comma-separated string nodesetOrder will become the highest priority, with the nodes earlier in that list being higher priority. The priority of nodes not mentioned in nodesetOrder will not be modified.

java.awt.Color Net.getNodesetColor (String nodeset)

void Net.setNodesetColor (String nodeset, java.awt.Color color)

10.2 Graph Algorithms

The nodes and links of a Bayes net form a "graph", as defined in graph theory. Graph theory provides algorithms to efficiently find all the descendents of a node, or all its ancestors, connected nodes, Markov blanket, etc. Netica very efficiently implements these algorithms, and makes them available with the `Node` method:

```
void getRelatedNodes (NodeList relatedNodes, String relation);
```

To use it, you pass the relation you desire as a string, and a node list to be filled. Then the function puts all of the related nodes into the list. For example, to find the Markov blanket of `node_A`, you could use:

```
NodeList mb = new NodeList (net);
node_A.getRelatedNodes (mb, "markov_blanket");
```

After execution, the list `mb` will contain all the nodes in the Markov blanket of `node_A`.

The allowed relation strings are: `"parents"`, `"children"`, `"ancestors"`, `"descendents"`, `"connected"`, `"markov_blanket"`, and `"d_connected"` (the singular version of each of these words is also acceptable, and does the same thing). You can add certain modifiers (in any order) to the string containing the relation. The allowed modifiers are:

"`append`" means to add to the list that is passed in (otherwise that list is first emptied).

"`union`" means to add to the list that is passed in and remove all duplicates.

"`intersection`" means to reduce the passed-in list to only the nodes that are in both the original passed-in list and the relation.

"`subtract`" means to take the nodes that are in the relation away from the passed-in list.

"`exclude_self`" is only relevant for: `"ancestors"`, `"descendents"`, `"connected"`, and "`d_connected`". Without it the relation list will also include the original node (generation 0).

"`include_evidence_nodes`" is only relevant for "`markov_blanket`" and "`d_connected`". Without it the relation list will not contain any nodes with findings.

For example, to create a list of all the nodes that are both ancestors of `node_A`, and descendents of `node_B`, you could use:

```
NodeList ad = new NodeList (net);
node_A.getRelatedNodes (ad, "ancestors");
node_B.getRelatedNodes (ad, "intersection, descendents");
```

If you want to find all the nodes that are related to a whole *group* of nodes, you use `Net.getRelatedNodes()`. It works the same as `Node.getRelatedNodes()`, except that it takes a list of nodes as an extra parameter:

```
void getRelatedNodes (NodeList relatedNodes, String relation, NodeList ofNodes);
```

Sometimes you don't need a list of all the nodes bearing some relation to a certain node, you just want to know if that relation holds between two nodes. For example, you may want to know if node A is an ancestor of node B. You could use the function described above to generate the whole list of ancestors of B, and then check if A is a member, but that would be wasteful. Instead, you call `Node.isRelated()`, like this:

```
if (node_A.isRelated ("ancestor", node_B)) ...
```

10.3 User-defined Data

Sometimes it is very useful to be able to attach your own data to Netica objects. Netica doesn't do anything with that data; it just holds the data until you ask for it back. The Netica objects that you can attach data to are: nets (`Net`) and nodes (`Node`).

There are two different ways of attaching data. One is to attach to the Netica object a single Java object. That object can be whatever you wish, perhaps a large collection. When the Netica object is duplicated or saved to file, the reference you have attached will be ignored. Only one such arbitrary Java object can be attached to each Netica object. The relevant methods for attaching and retrieving data in this way are: `Node/Net/Environ.user().setReference()`and
`Node/Net/Environ.user().getReference()`:

```
void setReference (Object obj);
Object getReference ();
```

The other way of attaching data to Netica objects is by "user fields", with which you can attach as many data items as you wish to an object, each under its own name (i.e., "attribute-value"). Your data will be duplicated if the node is duplicated, and when you save your net to file, Netica will include your data in the file. Representative prototypes are:

```
void    setNumber       (String fieldName, double fieldValue);
void    setString       (String fieldName, String fieldValue);
void    setObject       (String fieldName, java.io.Serializable fieldValue);
void    setBytes        (String fieldName, byte[] bytes);
void    removeField      (String fieldName);
double  getNumber        (String fieldName);
String  getString        (String fieldName);
Object  getObject        (String fieldName);
byte[]  getBytes         (String fieldName);
String  getNthFieldName         (int index);
```

To set a user field you pass a name for the field and a reference to your data. When you later call the method to recover your data, you pass in the name you gave it, and Netica will return you a newly constructed object identical to the original (except for its hashcode).

For example:

```
myNode.user().setString (“author”, “Sarah”);
…
String author = myNode.user().getString(“author”);
```

User fields can be very conveniently viewed or modified using Netica Application, which provides a good way of transferring information between an end-user and your Netica-J program (as are node-sets).

If you wish to find all the user fields defined for some node or net, you can iterate through them with `User.getNthFieldName()`.

10.4 Sensitivity

Of significant importance in Bayes net work is a measure of the independence between various nodes of the net. Using just the link structure and d-separation rules, you can determine which nodes are completely independent of which other ones (see the “Graph Algorithms” section above), and how that changes as findings arrive. However, dependence is a matter of degree, and using Netica’s sensitivity functions, you can efficiently determine how much an as yet unknown finding at one node will likely change the beliefs at another node.

There are many varied uses for the sensitivity measure. During diagnosis, it can specify which nodes will be the most informative in crystallizing the beliefs of the most probable fault nodes. As findings arrive it will adjust to account for the new findings, always identifying where further information would be useful to complete the diagnosis, in a most intelligent manner. In a net built for classification, you can determine which features are the most valuable for performing the classification (i.e. “feature selection”). In an information gathering environment, you can identify which are the most important questions to ask at each point (to provide information on the variables of interest), based on the answers to questions already received, so as to avoid asking unnecessary or irrelevant questions. When building a model of the world, such as an environmental model, you can determine which parts of the model most affect the variables of interest; thereby identifying which parts should be made the most carefully and accurately.

Say you are interested in the beliefs of a particular node, which we call the *target node*. Then there are a set of other nodes (called the *varying nodes*), for which it may be possible to have findings, and you want to know how much those findings are likely to influence the beliefs of the target node.

To use Netica's sensitivity functions, you first create a `Sensitivity` object using the constructor:

```
Sensitivity (Node targetNode, NodeList varyingNodes, int whatFind);
```

You pass it the target node `targetNode`, and a list of the varying nodes `varyingNodes`. You will be able to use the `Sensitivity` object returned to find the sensitivity of `targetNode` to each of the nodes in `varyingNodes`. You also pass `whatFind` to indicate what type of sensitivity calculations you wish it to be able to perform, which should be `VARIANCE_OF_REAL_SENSV` if you wish to be able to call `getVarianceOfReal`, `ENTROPY_SENSV` if you wish to be able to call getMutualInfo, or their bitwise-or to be able to use both. Finally, you obtain the actual sensitivity numbers by calling one of these methods on the `Sensitivity` object:

```
double getMutualInfo (Node varyingNode);
double getVarianceOfReal (Node varyingNode);
```

If the target node is discrete with no real number levels associated with the states, then the mutual information is the only function that can be used. If the target node is a discretized continuous node, or a discrete node with a real number associated with each state, then the variance-of-real measure is the recommended measure, although you may wish to use mutual information in some situations. The mutual information is the reduction in entropy of the target node belief distribution, due to a finding at the varying node (over each possible finding, weighted by the probability of obtaining that finding).

When you call one of the two functions, it will return the sensitivity of the original `targetNode` (used in the construction of the `Sensitivity`) with respect to the `varyingNode` passed in. The first time it is called, it takes longer to return, since it is calculating the results for all the `varyingNodes` that were used in the construction of the `Sensitivity` (because it can save time doing them all at once), but it remembers the results so subsequent calls are very fast (unless a finding or something else in the net changes, in which case it must re-calculate).

Mutual information is symmetric (i.e., it has the same value when the target node and varying node are reversed), so you can use `getMutualInfo()` to efficiently determine how much obtaining a finding at one node will likely effect the beliefs of all the rest of the nodes in the net.

When you are finished using a `Sensitivity` object, you can safely free its resources using `finalize()`, or you can leave that to the JVM.

Currently Netica's sensitivity analysis works only on Bayes nets, and not decision nets. You can also use Netica Application to do sensitivity analysis by choosing **Network → Sensitivity to Findings** from the menu. For more information on Netica's calculation of sensitivity, contact support@norsys.com, and ask for the "Sensitivity" document.

10.5 Stochastic Simulation

Netica can be used to generate random cases (aka "synthetic data"), which are cases whose values follow the distribution represented by the Bayes net, including any findings that it has.

This synthetic data may be browsed by people to get a feel for the type of cases to expect, or used to test people on their predictive or diagnostic ability. It can be used to learn other Bayes nets, or other machine learning representations, such as neural nets, decision trees or decision rules.

Perhaps its most valuable use is when the Bayes net is a physical model of a real-world situation, and the synthetic data provides stochastic simulations. The output of those simulations can then be analyzed by other programs. For example, the Bayes net may model a warehouse and distribution scheme, which can be tested under various conditions to check its performance. In a similar vein the Bayes net may model a control system, economic system, political environment, computer network, etc.

To generate a synthetic case, the method of Net to use is:

```
int Net.generateRandomCase (NodeList nodeList, int method, double timeout);
```

where the `method` argument determines which algorithm Netica uses (for example, forward sampling with rejection or by junction tree). For an example of a small program using it, see the SimulateCases.java example program in the "Findings and Cases" chapter.

11 Equations

The relation between a node and its parent nodes can be defined using an equation if desired. This eliminates the burden of building conditional probability tables (CPTs) manually. It is possible to use an equation for continuous or discrete nodes, and for probabilistic or deterministic relations.

Equations are a kind of "short-hand" form of expressing a CPT. Since Netica's Bayesian inference usually requires that CPTs be available, equations must be converted to tables (by calling `Node.equationToTable()`) before compiling a net, or doing certain net transforms like absorbing nodes or reversing links. Netica then uses the tables in the same way as if they had been entered directly.

Sometimes Netica uses an equation directly, without the need for a table. If findings are entered for all the parents of a node, and that node has a deterministic equation, then the node is given the exact value computed from the equation (which can then propagate to its children) during a *deterministic propagation* phase that is the first step of belief updating (see `Node.calcValue()` and `Node.CalcState()`). Having this phase increases both accuracy and speed, and can be useful for "preprocessing" input data. Another time Netica uses an equation directly is during stochastic simulation (calling `Node.generateRandomCase()` with `method=FORWARD_SAMPLING`).

11.1 Simple Examples

Here are some examples of using equations in Netica:

Suppose X is a continuous variable representing the position of a moving object, and is dependent on its parent nodes: Velocity, Time, and Start position. This equation could compactly express their relationship:

```
X (Velocity, Time, Start) = Start + Velocity * Time
```

Now suppose that the start position is zero, but that there is some uncertainty about the end position, given by the normal distribution with standard deviation S:

```
      p (X | Velocity, Time, S) = NormalDist (X, Velocity * Time, S)
```

Here is an example of a discrete node Color with states red, blue and green. As a parent, it has the discrete node Taste with states sour, salty and sweet. The below is a deterministic equation giving Color as a function of Taste, which demonstrates the use of the conditional operator ?:

```
      Color (Taste) =
           Taste==sour?  blue:    Taste==sweet? red:
           Taste==salty? green:   gray
```

Finally, consider a discrete node Color, which is indicator taking on the values red or blue depending on whether the parent node Taste is sweet or not, but that works imperfectly:

```
      p (Color | Taste) =
           (Taste==sweet) ? (Color==red ? 0.9 : 0.1):  0.5
```

For more examples, see the “Specialized Examples” section below.

11.2 Equation Syntax

Netica equations follow most of the usual standards for mathematical equations, and are similar to programming in Java, C or C++. The usual mathematical operators (+, -, *, /, etc.), and the usual functions (min, abs, sin, etc.) can be used, parenthesis are used for grouping, and numeric constants are in their usual form (e.g. 3, -4.2, 5.3e-12).

Left-Hand Side: For a deterministic node, the part of an equation to the left-hand side of the equals symbol consists of the name of the node, an open parenthesis, a list of the names of the parents separated by commas, and a close parenthesis (if you have defined link names, you must use those instead of parent names). For instance, if the equation is for node Position, and the parents of Position are Velocity, Time and Mode, the left hand side could be:

```
      Position (Velocity, Time, Mode) = ...
```

Note that the spaces are not required, there may be more spaces if desired, and the parents can be in any order.

For probabilistic nodes (i.e. "chance nodes"), the left-hand side consists of a lower case "p", an open parenthesis, the name of the node, a vertical bar, a list of the names of the parents (or link names) separated by commas, and a close parenthesis. If the node mentioned above had been a probabilistic node, the left hand side of its equation could be:

```
      p (Position | Velocity, Time, Mode) = ...
```

Right-Hand Side: The right-hand side of an equation may consist of numbers, state names, conditionals, variables (i.e. parent nodes), constant nodes, and built-in functions, constants or operators. Probabilistic equations will normally also contain the node the equation is for on the right-hand side (possibly in several places).

Nodes Allowed: The only nodes which may be mentioned in an equation are: the node the equation describes, its parents, and any constant node.

Whitespace: As many spaces or line breaks as desired may be placed between any two symbols.

Comments: Comments may be embedded in equations, and they will be ignored by Netica. Everything between /* and */ will be interpreted as a comment, as will everything between // and the end of the line.

All Values: If the equation is for a probabilistic node, its right-hand side must provide a probability for all the node's possible values (so the name of the node must appear there at least once). For example, if node **`Color`** (with states red, orange, yellow) has parent **`Temp`** (with states low, med, high), its equation could be:

```
p (Color | Temp) =
Temp == high      ? (Color==yellow ? 1.0 : 0.0) :
Temp == med       ? (Color==orange ? 1.0 : 0.0) :
Temp == low       ? (Color==orange ? 0.2 : Color==red ? 0.8 : 0.0) : 0
```

If you use the built-in distributions (such as NormalDist), the above rule is automatically taken care of.

One exception to the above rule is if a node is boolean. Then only the probability for the true state need be given. For example, if node **It_Falls** is boolean, its equation could be:

```
p (It_Falls | Weight, Size) =
Weight/Size > 10 ? 0.10 :
Weight/Size > 5  ? 0.03 :
                   0.01
```

Differences between standard Java (or C/C++) equation syntax: The Netica equation syntax is the same as in the Java (and C and C++) programming languages, except the part to the left of the assignment operator (=) is different, and no semicolon is required at the end of the equation.

Furthermore, the Java/C/C++ bitwise operators (such as &, |, ~, ^) are not available in Netica, but the logical operators &&, ||, ! are. In addition, Netica has a logical 'xor' function. A final difference is that the bitwise xor operator ^ of Java/C/C++ is instead used as the power operator by Netica (thus 2^3=8).

All of the C Standard Library math functions (sin, log, sqrt, floor, etc.) are available and use the same names.

11.3 Equation Conditionals

Suppose continuous node X has the parents Y and B. If you wanted to give P(X|Y) a different equation involving X and Y for different values of B, you could write a conditional statement using the ? and : operators, like this:

```
p(X|Y,B) =
   (B < 2) ? NormalDist (X, 3 + Y, 1) :
   (B < 6) ? NormalDist (X, 2 + Y, 3) :
             UniformDist (X, 0, 10)
```

The conditions are evaluated in order, so the first covers all cases where B < 2, the second covers cases 2 ≤ B < 6, and the last covers the remaining cases (i.e. B ≥ 6). So, if B is less than 2, X is distributed normally with mean 3+Y; if it is between 2 and 6 then the mean is 2+Y; and if it is over 6 then X is distributed uniformly.

If there are more parents, this sort of construct can be nested to provide a tree structure of possible contingencies.

Here are a couple more examples. They show a way to condition over the states of a discrete node:

```
p(X|Y,B) =
(B == yellow) ? NormalDist (X, 2, sqrt (Y)) :
(B == orange) ? NormalDist (X, 4, Y) :
(B == red)    ? NormalDist (X, 6, Y ^ 2) : 0

p(X|B) =
member (B, CA, TX, FL) ? NormalDist (X, 3, 1) :
member (B, MA, WA)     ? NormalDist (X, 5, 1) :
member (B, NY, UT, VA) ? NormalDist (X, 7, 2) :
                         UniformDist (X, 0, 10)
```

Notice that the "fall through" case of the first example above is simply a 0. This indicates that the designer is counting on B to be one of *yellow*, *orange* or *red*. If B ever has another state, then when Netica is converting the equation to a table it will give a warning message that "for n/N conditions, no nonzero probability was discovered by sampling" (providing no sampling uncertainty is being added).

In the last example, the fall through case gives a uniform distribution. If extra states are later added to B, then they will just fall through and use the uniform distribution.

11.4 Converting an Equation to a Table

As mentioned earlier, all equations must be converted to tables before compiling a net or doing net transforms like absorbing nodes or reversing links. The procedure is done by the following three steps:

1. If the node, or any of its parents, is a continuous node that has not yet been discretized, then call **`Node.setLevels()`** to discretize it. The finer the discretization, the more accurate, but the bigger the tables will be.

2. If the node doesn't already have its equation, call **`Node.setEquation()`**, passing in the equation string.

3. Finally, call **`Node.equationToTable()`**. Note that if you later change the equation for the node, or the discretization of the node or of any of its parents, or the finding of a constant node referred to by the equation, you must repeat this step before the changes will take effect. With the parameters passed to this function you can control the number of samples in any Monte Carlo integration that is required, whether the final CPT will include uncertainty due to the sampling process, and you can blend tables with those produced by learning from data, other equations, or manual CPT entry into Netica Application.

If Netica reports errors in the above steps, it is often helpful to debug the equation using Netica Application. If there is a problem with the syntax of an equation, when you enter it into Netica Application's node property dialog box, the cursor will be placed on the problem while the error message is displayed. From Netica Application's menu, you can choose "Equation To Table" to check if there is going to be any problem with the equation, and conveniently view the resulting CPT to see if it is what you expect.

11.5 Equations and Table Size

The size of the table generated is the product of the number of states of the node with the numbers of states of each of its parent nodes. So if a node has many states, or many parents, then the tables may be very large, and Netica may report that it doesn't have enough memory for the operation. You can alleviate the problem by eliminating unnecessary parents, introducing intermediate variables, or using more course discretizations (perhaps have more than one node for the same variable, with different discretizations depending on which node it is a parent for). If Netica creates extremely large tables, it may starve other processes of memory, or result in very slow virtual memory hard disk activity, so you might want Netica to instead just report that it doesn't have enough memory. In that case, you can limit the amount of memory available to Netica with `Environ.setMemoryUsageLimit()`.

11.6 Link Names

In the simplest way of writing equations, the names of the parent nodes appear in the equation. However, you might want a more modular representation, so that you can disconnect some of the parent nodes and hook the node up to new parents without having to change all the parent names within the equation.

Or perhaps you duplicate the node to use with new parents. Or you put the node in a network library without any parents. Or you want to copy the equation from one node to another, without changing all the node names.

The way to do that is to use *input names*, sometimes called *link names*. They provide an argument name for each link entering the node (and therefore a proxy for each parent node). You can set them with `Node.setInputName()`. You refer to them in your equation in exactly the same way you would the corresponding parent name. When a parent is disconnected, the link name will remain.

Note. If link names are defined for a node, they **must** be used instead of the parent names.

11.7 Referring to States of Discrete Nodes

To refer to the states of a discrete or discretized node, You can use the state names of a discrete node as constants in an equation. For example, if node *Color* has states *red*, *green*, *blue* and *yellow*, and node *Temperature* has states *cool* and *warm*, you could write:

```
Temperature (Color)  =  member (Color, red, yellow) ? warm : cool
```

Each state name only has meaning relative to the node it's for. Usually when you use a state name, Netica can identify that node from context. However, if Netica doesn't know which node a state name refers to (e.g. it gives an unknown value error message), you can indicate which node by following the state name with a double-dash and then the name of the node. Continuing with the above example, if a new node *Switch* could take on the values 0, 1 and 2, you could write:

```
Color (Switch) = select0 (Switch, red--Color, yellow, blue)
```

The "--Color" was not required on "yellow" and "blue", because the context was carried over from "red--Color", but it could be put there as well.

If a discrete node has a numeric value associated with each state (see `Node.setLevels()`), that numeric value can be used in an equation instead of the state name.

Alternatively, you can use the state index (numbering starts at 0) preceded by a hash # character. However, it is recommended to use the names or values, because they are more readable, less error-prone and more robust to future changes to the node, such as the adding or re-ordering of states.

11.8 Constant Nodes as Adjustable Parameters

Sometimes it is useful to have an equation parameter that normally acts as a fixed constant, but which you can change from time to time. That is the purpose of a *constant node*.

You create a constant node by addinga nature node to the network, and then converting it to a constant node by calling `Node.setKind()`. You can also set other characteristics of a constant node in the same way as any other node, such as giving it state names. To set or change the value of a constant node, enter the value in the same way as you would enter a finding.

You can refer to the value of a constant node anywhere in any node's equationby using the constant node's name. It should not appear in the argument list on the left hand side of the = symbol. No link is required.

When you convert the equation to a table, the value of any constant nodes it references will be used. If you change the value of a constant node, you must rebuild the table for the change to take effect.

11.9 Tips on Using Equations

- It is often helpful to debug equations using Netica Application. If there is a problem with the syntax of an equation, it leaves the cursor on the problem while it displays an error message. You can choose "Equation To Table" from the menu to check that, and easily view the resulting CPT to see if it is what you expected.

- The tables generated by equations may result in large files (and therefore slow reading), so you may want remove the nodes' tables with `Node.deleteTables()`, before saving it to file. Later, when you restore the net from file, you call `Node.equationToTable()` to fully restore them.

- If you need to define intermediate variables to simplify the equations, implement them as new (intermediate) nodes.

11.10 Specialized Examples

State Comparisons: Suppose the states of node Source are CA, TX, FL, BC and NY. The states of node Dest are TX, NY, MA and UT. We want to know if cross-border travel is required to transport from Source to Dest, and that is indicated by the boolean node Travel. The equation below works even though nodes Source and Dest have different sets of states, and in a different order.

```
Travel (Source, Dest) = (Source != Dest)
```

Additive Noise: Say you want to represent something like:
x1 = x2 + gauss (0, 0.2) which could indicate that x1 is the same as x2, but with the addition of gaussian noise having mean 0 and s = 0.2. You could do this by defining a new node x3, and setting the equations of x1 and x3 as:

```
X1 (X2, X3) = X2 + X3

p(X3) = NormalDist (X3, 0, 0.2)
```

Multiple Discretizations: Sometimes it is beneficial to use more than one node to represent a single continuous variable, but with each discretized differently. For example, the more course one may be a parent for another node whose CPT would be too big with a finer discretization, while the finer one would serve as a parent for nodes requiring more accuracy. Put a link from the finer node to the courser, and give the courser node an equation like:

```
X5 (X20) = X20
```

Noisy-Or: To create a noisy-or node, just create a regular boolean nature node, put links to it from the possible causes, give it a noisy-or equation, and use that to build its CPT.

For example, if C1, C2 and C3 are boolean nodes representing causes of boolean node E, and there are links from each Ci to E, then E could have the noisy-or equation:

```
p (E | C1, C2, C3) =
NoisyOrDist (E, 0, C1, 0.5, C2, 0.3, C3, 0.1)
```

For its meaning, see the NoisyOrDist description. The causes, and even the link parameters, can be more complex expressions. For example:

```
p (Bond | Temperature, BackTemp, Pressure, Switch, Eff)=
NoisyOrDist (Bond, 0.001,
Temperature > BackTemp, 0.5,
Pressure == high, 0.3,
Switch, 0.9 * Eff)
```

For more information on using Netica’s Noisy-Or, Noisy-And, Noisy-Max and Noisy-Sum functions, contact Norsys for the “Noisy Or, Max, Sum” document.

11.11 Equation Constants, Operators, and Functions

A: Built-in Constants

The following constants may be used in equations:

```
pi      = 3.141592654

deg     = radian per degree = pi / 180
```

If you wish to have the constant **e** (= 2.7182818) in your equation, use `exp(1)`.

B: Built-in Operators

Both the functional and the operator notations shown below are accepted.

Functional Notation	Operator Notation
neg (x)	- x
not (b)	! b
equal (x, y)	x == y
not_equal (x, y)	x != y
approx_eq (x, y)	x ~= y
less (x, y)	x < y
greater (x, y)	x > y
less_eq (x, y)	x <= y
greater_eq (x, y)	x >= y
plus (x_1, x_2, ... x_n)	x_1 + x_2 + ... + x_n
minus (x, y)	x - y
mult (x_1, x_2, ... x_n)	x_1 * x_2 * ... * x_n
div (x, y)	x / y
mod (x, base)	x % base
power (x, y)	x ^ y
and (b_1, b_2, ... b_n)	b_1 && b_2 && ... && b_n
or (b_1, b_2, ... b_n)	b_1 || b_2 || ... || b_n
if (test, tval, fval)	test ? tval : fval

C: Built-in Functions

Netica contains an extensive library of built-in functions which you can use in your equations.

The probability distribution functions all have a name that ends with "Dist" (e.g. NormalDist). Their first argument is always the node for which the distribution is for. So if node X has parent m, you could write:

```
P (X | m) = NormalDist (X, m, 0.2)
```

to indicate that X has a normal (Gaussian) distribution with mean given by parent m, and a standard deviation of 0.2.

Common Math

abs (x)	absolute value
sqrt (x)	square root (positive)
exp (x)	exponential (e ^ x)
log (x)	logarithm base e
log2 (x)	logarithm base 2
log10 (x)	logarithm base 10
sin (x)	sine (x is in radians)
cos (x)	cosine
tan (x)	tangent
asin (x)	arc sine (result is in radians)
acos (x)	arc cosine
atan (x)	arc tangent
atan2 (y, x)	atan(y/x) but considers quadrant
sinh (x)	hyperbolic sine
cosh (x)	hyperbolic cosine
tanh (x)	hyperbolic tangent
floor (x)	floor (highest integer ≤ x)
ceil (x)	ceiling (lowest integer ≥ x)
integer (x)	integer part of number (same sign)
frac (x)	fraction part of number (same sign)

Special Math

```
round (x)
roundto (dx, x)
approx_eq (x, y)
eqnear (reldiff, x, y)
clip (min, max, x)
sign (x)
xor ($b_1$, $b_2$, ... $b_n$)
increasing ($x_1$, $x_2$, ... $x_n$)
increasing_eq ($x_1$, $x_2$, ... $x_n$)
min ($x_1$, $x_2$, ... $x_n$)
max ($x_1$, $x_2$, ... $x_n$)
argmin0/1 ($x_0$, $x_1$, ... $x_n$)
argmax0/1 ($x_0$, $x_1$, ... $x_n$)
nearest0/1 (val, $c_0$, $c_1$, ... $c_n$)
select0/1 (index, $c_0$, $c_1$, ... $c_n$)
member (elem, $s_1$, $s_2$, ... $s_n$)
factorial (n)
logfactorial (n)
gamma (x)
loggamma (x)
beta (z, w)
erf (x)
erfc (x)
binomial (n, k)
multinomial ($n_1$, $n_2$, ... $n_n$)
```

Continuous Probability Distributions

```
UniformDist (x, a, b)
TriangularDist (x, m, w)
Triangular3Dist (x, m, $w_1$, $w_2$)
TriangularEnd3Dist (x, m, a, b)
NormalDist (x, $\mu$, $\sigma$)
LognormalDist (x, $\eta$, $\phi$)
ExponentialDist (x, $\lambda$)
GammaDist (x, $\alpha$, $\beta$)
WeibullDist (x, $\alpha$, $\beta$)
BetaDist (x, $\alpha$, $\beta$)
Beta4Dist (x, $\alpha$, $\beta$, c, d)
CauchyDist (x, $\mu$, $\sigma$)
LaplaceDist (x, $\mu$, $\beta$)
ExtremeValueDist (x, $\mu$, $\sigma$)
ParetoDist (x, a, b)
ChiSquareDist (x, $\nu$)
StudentTDist (x, $\nu$)
FDist (x, $\nu_1$, $\nu_2$)
```

Discrete Probability Distributions

SingleDist (k, c)
DiscUniformDist (k, a, b)
BernoulliDist (b, p)
BinomialDist (k, n, p)
PoissonDist (k, m)
HypergeometricDist (k, n, s, N)
NegBinomialDist (k, n, p)
GeometricDist (k, p)
LogarithmicDist (k, p)
MultinomialDist (bc, n, k_1, p_1, k_2, p_2, ... k_m, p_m)
NoisyOrDist (e, leak, b_1, p_1, b_2, p_2, ... b_n, p_n)
NoisyAndDist (e, inh, b_1, p_1, b_2, p_2, ... b_n, p_n)
NoisyMaxTableDist (...)
NoisySumTableDist (...)

11.12 Special Math and Distribution Functions Reference

Legend: 📊 **= Discrete Probability Distribution**

(the first argument is a discrete variable that the distribution is over)

▲ **= Continuous Probability Distribution**

(the first argument is a continuous variable that the distribution is over)

`approx_eq (x, y)` **`x ~= y`** = eqnear (2e-5, x, y)

where x and y are unrestricted real numbers

Returns TRUE iff **`x`** is equal to **`y`**, within a small relative tolerance.

Usually the operator form of this function is most convenient: **`x`** `~=` **`y`**

It is meant for comparing computed real number values that might not be *exactly* equal due to slight numerical inaccuracies.

To have control of the tolerance, use `eqnear`.

---

argmax0 (x_0, x_1, ... x_n) = i s.t. ($x_i \geq x_j$) for all j
argmax1 (x_1, x_2, ... x_n)

where x_i are unrestricted real numbers

Returns the index (position in list) of the argument with the highest value. If there are several with the same highest value, then the index of the first occurrence will be returned. The first argument has index 0 if `argmax0` is used, or index 1 if `argmax1` is used. At least one argument must be passed. See also `max`, `argmin`, `select`.

Example:
```
argmax0 (1, -6.6, 3.4, 1.26, 3.4)   returns 2
argmax1 (1, -6.6, 3.4, 1.26, 3.4)   returns 3
```

argmin0 (x_0, x_1, ... x_n)
argmin1 (x_1, x_2, ... x_n)

= i s.t. ($x_i \leq x_j$) for all j

where x_i are unrestricted real numbers

Returns the index (position in list) of the argument with the lowest value. If there are several with the same lowest value, then the index of the first occurrence will be returned. The first argument has index 0 if `argmin0` is used, or index 1 if `argmin1` is used. At least one argument must be passed. See also `min`, `argmax`, `select`.

Example:

```
argmin0 (10, 6.6, 3.4, 126, 3.4)   returns 2
argmin1 (10, 6.6, 3.4, 126, 3.4)   returns 3
```

BernoulliDist (b, p)

= b ? p : 1 - p

Required: $0 \leq p \leq 1$ `b boolean`

This is the distribution for a single "Bernoulli trial", in which **p** is the probability of an outcome labeled "success" occurring. **b** is a boolean that is true if the “success” occurs. An example is flipping a coin and checking for the event of heads appearing.

_BernoulliDist

This is a distribution that Netica uses internally to represent the Bernoulli distribution (`BernoulliDist`). If you get an error message saying there was an error evaluating _Bernoulli (k, p), where k and p are numbers, then your equation is supplying illegal values, even if you never explicitly used _Bernoulli in your equation.

For instance, if your equation for boolean B is P(B|x) = x / 10 and values of `x` can go up to 11, then _Bernoulli (1, 1.1) will be illegal, since you are supplying 1.1 as a probability (and Netica can’t normalize it, since no probability for B being false is given).

beta (z, w)

= gamma (z) gamma (w) / gamma (z + w)

where: $z > 0$ $w > 0$

Returns the beta function of **z** and **w**. `BetaDist` is the beta probability distribution, which is based on the beta function.

BetaDist (x, α, β)

= $x^{\alpha-1} (1-x)^{\beta-1}$/ beta (α, β)

Required: $\alpha > 0$ $\beta > 0$

The beta distribution over x. Almost any reasonably smooth unimodal distribution on [0,1] can approximated to some degree by a beta distribution (if its not on [0,1], see `Beta4Dist`).

Beta4Dist (x, α, β, c, d)

= BetaDist ((x - c) / (d - c), α, β)

Required: $0 \leq x \leq 1$ $\alpha > 0$ $\beta > 0$

Also known as the “Generalized Beta Distribution”, this is a beta distribution that has been shifted and scaled, so that the pdf has nonzero values from **x** = **c** to **x** = **d**, instead of from **x**=`0` to **x**=`1`. This distribution has great flexibility to roughly fit almost any smooth, unimodal distribution with no tails (i.e., only nonzero over a finite range).

binomial (n, k)

= n! / (k! * (n-k)!)

Where: $0 \leq k \leq n$ `n and k are integers`

Returns the binomial coefficient (**n k**). That is the number of different **k**-sized groups that can be drawn from a set of **n** distinct elements. See also the `multinomial` function.

`BinomialDist` is the binomial probability distribution, which is based on the binomial coefficient..

BinomialDist (k, n, p)

= binomial (n, k) $p^{k} (1-p)^{n-k}$

Required: `k and n are integers,` $0 \leq k \leq n$, and $0 \leq p \leq 1$

A "binomial experiment" is a series of **n** independent trials, each with two possible outcomes (often labeled "success" and "failure"), with a constant probability, **p**, of success. The total number of successes, **k**, is given by the binomial distribution.

If there are more than two possible outcomes, use the multinomial distribution (`MultinomialDist`). If the sampling is without replacement, use the hypergeometric distribution (`HypergeometricDist`)

For large **n**, and **p** not too close to 0 or 1, the binomial distribution can be approximated by a normal distribution (`NormalDist`) with mean m = **n p**, and variance = **n p** (1-**p**). For large **n**, and **p** close to 0, it can be approximated by a Poisson distribution (`PoissonDist`) with parameter λ = **n p**. As **n** $\rightarrow \infty$ these are the limiting distributions (providing **p**=constant in the normal case, and **p** $\rightarrow$ 0, **np**=constant in the Poisson case).

CauchyDist (x, μ, σ)

$= 1 / (\pi\, \sigma\, (1 + ((x-\mu)/\sigma)^2))$

Required: $\sigma > 0$

Although real-world data rarely follows a Cauchy distribution, it is useful because of its unusualness. For example, although it is symmetric about **μ** (which is therefore its median and mode), it doesn't have a mean (or variance, etc.) because the appropriate integrals don't converge. The C(0,1) distribution is also Student's t distribution with degrees of freedom = 1.

ChiSquareDist (x, ν)

$= x^{(\nu/2-1)} / [\exp(x/2)\; 2^{(\nu/2)}\; \mathrm{gamma}(\nu/2)]$

Required: $x \geq 0$ $\nu > 0$ ν is an integer

This is the distribution of $Z_1^2 + Z_2^2 + \ldots Z_\nu^2$ where Z_i are independent standard normal (`NormalDist`) variates.
ν is usually called the “degrees of freedom” of the distribution.

---

clip (min, max, x)

= (x < min) ? min : (x > max) ? max : x

where `min ≤ max`

Returns **`x`**, unless it is less than **`min`** (in which case it returns **`min`**), or more than **`max`** (in which case it returns **`max`**).
See also the functions: `min`, `max`.

DiscUniformDist (k, a, b)

$= 1 / (b - a + 1)$

Required: `a ≤ b` `k, a, b are integers`

This distribution represents the situation where **`k`** has an equal probability of taking on any of the integer values from **`a`** to **`b`** inclusive (where **`a`** and **`b`** are integers). If **`k`** were continuous, then it would be a continuous uniform distribution.

---

eqnear (reldiff, x, y)

$= (\,|X - Y| / \max(|X|, |Y|) \leq \mathrm{reldiff}\,)$

where `reldiff ≥ 0`

Returns TRUE iff **`x`** is equal to **`y`**, within `reldiff`. To use a tiny built-in value for `reldiff`, suitable for numerical floating point inaccuracy, use `approx_eq`.

---

erf (x)

$$= \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2)\, dt$$

where x is an unrestricted real

This returns the error function of **`x`**. It is useful for calculating integrals of the normal distribution function (`NormalDist`).
If **`x`** is large, you can obtain better accuracy with `erfc`.

---

erfc (x)

$= 1 - \mathrm{erf}(x)$

where x is an unrestricted real

This returns the complementary error function of **`x`**. It is useful for calculating an integral of a tail of a normal distribution function (`NormalDist`). It would be easy enough to just use `1-erf(`**`x`**`)`, but this provides better numerical accuracy when **`x`** is large (so `erf(`**`x`**`)` is very close to 1).

ExponentialDist (x, λ)

$= \lambda \exp(-\lambda x)$

Required: $\lambda > 0$

If events occur by a Poisson process, then the time between successive events is described by the exponential distribution (where λ is the average number of events per unit time).

ExtremeValueDist (x, α, β)

$= \exp(-\exp(-(x-\alpha)/\beta) - (x-\alpha)/\beta) / \beta$

Required: $\beta > 0$

This distribution is the limiting distribution for the smallest or largest values in large samples drawn from a variety of distributions, including the normal distribution Also known as the "Fisher-Tippet distribution", "Fisher-Tippet Type I distribution" or the "log-Weibull distribution".

FDist (x, ν₁, ν₂)

Required: $\nu > 0$ $\nu_2 > 0$

The ratio of two chi-squared variates X_1 and X_2, each divided by their degrees of freedom: $(X_1/\nu_1)/(X_2/\nu_2)$ follows an F-distribution. Also known as "Snedecor's F distribution", "Fisher-Snedecor distribution", "F-ratio distribution" and " variance-ratio distribution ".

factorial (n)

$= n (n – 1) (n – 2) \ldots 1$

where $n \geq 0$ `n is an integer`

Returns the factorial of **n**, which is the product of the first **n** integers.

`factorial(n)` is often written as **n**!

`factorial(0) = 1`

Even fairly small values of **n** (around 170) can cause `factorial` to overflow. For that reason calculations with the factorial function are often done using the logarithm of the results, for which you can use `logfactorial`.

If **n** is not an integer you may want to use the `gamma` function, which for integer values is related to factorial by: `factorial` (**n**) = `gamma` (**n** + 1) but which is also defined for non-integer values.

gamma (x)

where $x \geq 0$

Returns the gamma function of **x**.

The gamma function is normally defined for negative values of **x** as well, but Netica cannot compute these.

Don't confuse this function with `GammaDist`, the gamma probability distribution.

Even fairly small values of **x** (around 170) can cause `gamma` to overflow. For that reason calculations with the gamma function are often done using the logarithm of the results, for which you can use `loggamma`.

For integer values of **x**, the gamma function is related to the `factorial` function by: `factorial` (n) = `gamma` (n + 1).

GammaDist (x, α, β)

$= x^{\alpha-1} e^{-x/\beta} / (\text{gamma}(\alpha)\, \beta^{\alpha})$

Required: $\alpha > 0$ $\beta > 0$

If events occur by a Poisson process, then the time required for the occurrence of α events is described by the gamma distribution (where β is the average time between events).

For $\alpha = 1$, this is the exponential distribution (`ExponentialDist`) with $\lambda = 1 / \beta$. For $\beta = 2$, this is the chi-square distribution (`ChiSquareDist`) with degrees of freedom $\nu = 2\alpha$.

GeometricDist (k, p)

$= p (1-p)^k$

Required: $0 < p \leq 1$ `k is an integer`

This distribution describes the number of Bernoulli trials (independent trials, with outcomes labeled "success" or "failure", and constant probability **p** of success) before the first success occurs (i.e., includes only the failure trials). An example would be the number of coin flips resulting in tails before the first head is seen.

Situations where Bernoulli trials are repeated until the nth success are called "negative binomial experiments", and the geometric distribution is a special case of the negative binomial distribution (`NegBinomialDist`) with **n** = 1.

`HypergeometricDist (k, n, s, N)` = binomial (s,k) binomial (N-s, n-k) / binomial (N,n)

Required: `N ≥ 0   0 ≤ n ≤ N   0 ≤ s ≤ N   k, N, n and s are integers`

This provides the probability that there are **k** "successes" in a random sample of size **n**, selected (without replacement) from **N** items of which **s** are labeled "success" and **N-s** labeled "failure".

It is used in place of the binomial distribution (`BinomialDist`) for situations which sample without replacement.

`increasing (`x_1`, `x_2`, ... `x_n`)` = $(x_1 < x_2)$ && $(x_2 < x_3)$ && ... && $(x_{n-1} < x_n)$

where x_i are unrestricted real numbers

Returns TRUE iff each $\mathbf{x_i}$ is greater than the previous one. If you wish the test to be "greater than or equals", use `increasing_eq`.

`increasing_eq (`x_1`, `x_2`, ... `x_n`)` = $(x_1 \le x_2)$ && $(x_2 \le x_3)$ && ... && $(x_{n-1} \le x_n)$

where x_i are unrestricted real numbers

Returns TRUE iff each $\mathbf{x_i}$ is greater than the previous one. If you wish the test to be just "greater than", use `increasing`.

`LaplaceDist (x, μ, β)` = (1/(2β)) exp (- |x-μ|/β)

Required: $\beta > 0$

Its pdf is two exponential distributions spliced together back-to-back. The difference between two iid exponential distribution random variables follows a Laplace distribution. Also known as the "double exponential" distribution.

`LogarithmicDist (k, p)` = -(p^k)/ (k log (1-p))

Required: `0 < p < 1    k is an integer`

Also known as the "logarithmic series distribution".

`logfactorial (n)` = log (n (n – 1) (n – 2) ... 1)

where $n \ge 0$ `n is an integer`

Returns the natural logarithm of the factorial of **n**, that is: log (**n**!).

You could also use the `factorial` function, but this helps to avoid overflow when n is large (>170).

If **n** is not an integer you may want to use the `loggamma` function, which for integer values is related to `logfactorial` by: `logfactorial (n) = loggamma (n + 1)` but which is also defined for non-integer values.

`loggamma (x)` = log (gamma (x))

where $x \ge 0$

Returns the natural logarithm of the `gamma` function of **x**.

It may be used to avoid overflow when **x** is large. The gamma function is normally defined for negative values of **x** as well, but Netica cannot compute these.

`LognormalDist (x, ξ, ϕ)` = N (log (x), ξ, ϕ) / x, where N is the "normal distribution"

$= (1 / [x\ \phi\ \mathrm{sqrt}(2\pi)])\ \exp(-[(\log(x) - \xi) / \phi]^2 / 2)$

Required: `ϕ > 0`

The lognormal distribution results when the logarithm of the random variable is described by a normal distribution (`NormalDist`). This is often the case for a variable which is the product of a number of random variables (by the central limit theorem). Notice that the 'n' of Lognormal is not capitalized, indicating that this is not the same as the logarithm of the normal distribution.

max $(x_1, x_2, \ldots x_n)$

$= x_i$ s.t. $(x_i \geq x_j)$ for all j

where x_i are unrestricted real numbers

Returns the maximum of $\mathbf{x_1}$, $\mathbf{x_2}$, … $\mathbf{x_n}$.

At least one argument must be passed. If you just want the index of the maximum (i.e. its position in the list), use `argmax`. See also `min`.

Example: `max (-10, 6.6, 3.4, -126, 3.4)   returns 6.6`

member (elem, s_1, s_2, ... s_n)

= (elem == s_1) || (elem == s_2) || ... || (elem == s_n)

where `elem` and all s_i must be the same type

Returns TRUE iff one of the $\mathbf{s_i}$ arguments has the same value as **elem**.. See also: `nearest`, `select`

Examples:
```
member (1, -6, 3, 1, 3)  returns TRUE
member (C, blue, red)  and C = red  returns TRUE
```

min $(x_1, x_2, \ldots x_n)$

$= x_i$ s.t. $(x_i \leq x_j)$ for all j

where x_i are unrestricted real numbers

Returns the minimum of $\mathbf{x_1}$, $\mathbf{x_2}$, … $\mathbf{x_n}$.

At least one argument must be passed.

If you just want the index of the minimum (i.e. its position in the list), use `argmin`. See also `max`.

Example: `min (10, 6.6, 3.4, 126, 3.4)   returns 3.4`

multinomial $(n_1, n_2, \ldots n_n)$

$= (n_1 + n_2 + \ldots n_n)! / (n_1! * n_2! * \ldots n_n!)$

where $n_i \geq 0$ $\quad$ n_i are integers

Returns the number of ways an $(\mathbf{n_1+n_2+\ldots n_n})$ sized set of distinct elements can be partitioned into sets of size $\mathbf{n_1}$, $\mathbf{n_2}$, … $\mathbf{n_n}$. If partitioning into only two sets, this is the same as `binomial`.

MultinomialDist (bc, n, k_1, p_1, k_2, p_2, ... k_m, p_m)

Required: $n >= 0$ $\quad k_i >= 0$ $\quad 0 <= p_i <= 1$ $\quad$ sum p_i != 0 $\quad$ bc boolean $\quad$ n, k_i integer

The multinomial distribution is a generalization of the binomial distribution to the situation where there are not just two outcomes (usually labeled "success" and "fail"), but rather **m** outcomes, each having probability $\mathbf{p_i}$ (i=1..m), and we are interested in the number of occurrences of each outcome ($\mathbf{k_i}$), given that a total of n trials are performed.

To create a multinomial distribution between the $\mathbf{k_i}$ and **n** nodes, first add to the net a new boolean node, in this example called **bc**. Then add links from the nodes of all the non-fixed parameters (usually **n** and all $\mathbf{k_i}$) to node **bc**. At node **bc**, put an equation with MultinomialDist, and convert the equation to a table. Finally, give node **bc** a finding of true.

Normally the sum of $\mathbf{p_i}$ is one, but Netica will just normalize the $\mathbf{p_i}$ if that is not the case.

If m is 2, then k_2 is deterministically determined by k_1 (i.e., $k_2 = n - k_1$), and k_1 is distributed by BinomialDist.

Each of the $\mathbf{k_i}$ separately has a binomial distribution with parameters n and pi, and because of the constraint that the sum of the $\mathbf{k_i}$'s is **n**, they are negatively correlated.

The Dirichlet distribution is the conjugate prior of the multinomial in Bayesian statistics.

For assistance on using this function, contact Norsys (support@norsys.com).

nearest0 (val, x_0, x_1, ... x_n)
nearest1 (val, x_1, x_2, ... x_n)

= i **s.t.** $(|val - x_i| \leq |val - x_j|)$ $(x_i \geq x_j)$ **for all** j

where val and x_i are unrestricted real numbers

Returns the index (position in list) of the argument with the value closest to **val** (as measured by the absolute value of the difference). If there are several with the same smallest difference, then the index of the first occurrence will be returned. The first **x** argument has index 0 if `nearest0` is used, or index 1 if `nearest1` is used.

Must be passed at least 2 arguments (**val** and an **x**). See also: `member`

Example:
```
nearest0 (1, 1, 3.4, 1, 3.4)      returns 0
nearest1 (5e3, -6.6, -3.4, 126)   returns 3
```

NegBinomialDist (k, n, p)

= binomial (n+k-1, k) p^n $(1-p)^k$

Required: `0 ≤ n    0 < p ≤ 1    k and n are integers`

The negative binomial distribution is the distribution of the number of failures that occur in a sequence of trials before **n** successes have occurred, in a Bernoulli process (independent trials, with outcomes labeled "success" or "failure", and constant probability **p** of success).

The limit of a negative binomial distribution as $n \to \infty$, $(1-p) \to 0$, $n(1-p) \to \lambda$, is a Poisson distribution with parameter λ.

If **n** = 1, then this distribution is just the geometric distribution.

NoisyAndDist(e,inh,b$_1$,p$_1$,... b$_n$,p$_n$)

= P(e) = (1-inh) **product i=1 to n** (b_i? 1: (1-p_i))

Required: $0 \le p_i \le 1$ $0 \le inh \le 1$ e, b_i boolean

Use this distribution when there are several possible requirements for an event, and each has a probability that it will actually be necessary. Each of the necessary requirements must pass for the event to occur. Even then there is a probability (given by **inh**) that the event may not occur (make **inh** zero to eliminate this).

Each $\mathbf{b_i}$ is a booleanvariable, which when TRUE indicates a requirement passed. **e** is also a boolean, which indicates whether the event occurs. Each of the $\mathbf{p_i}$ are the probability that $\mathbf{b_i}$ will be required to cause **e**.

If **inh** is zero, and only one possible requirement is FALSE, say $\mathbf{b_k}$, then the probability for **e** is $1- \mathbf{p_k}$. If more possible requirements are FALSE, the probability will be lower. And if **inh** is nonzero, the probability will be lower. Reducing a $\mathbf{p_i}$ always results in the same or higher P(**e**).

$\mathbf{p_i}$ can be considered the “strength” of the relation between **e** and $\mathbf{b_i}$, with zero indicating independence (link could be removed), and 1 indicating maximum effect. See also `NoisyOrDist`.

NoisyMaxDist(...)
NoisySumDist(...)

For documentation, contact Norsys to obtain the document titled “Noisy Or, Max, Sum”.

NoisyOrDist(e,leak,b$_1$,p$_1$,... b$_n$,p$_n$)

= P(e) = 1–[(1-leak) **product i=1 to n** (b_i? (1-p_i): 1)]

Required: $0 \le p_i \le 1$ $0 \le leak \le 1$ e, b_i boolean

Use this distribution when there are several possible causes for an event, any of which can cause the event by itself, but only with a certain probability. Also, the event can occur spontaneously (without any of the known causes being true), with probability **leak** (make this zero if it can’t occur spontaneously).

Each $\mathbf{b_i}$ is a booleanvariable, which may cause the event when its TRUE. **e** is also a boolean, which indicates whether the event occurs. Each of the $\mathbf{p_i}$ are the probability that **e** will occur if $\mathbf{b_i}$ is TRUE in isolation.

If **leak** is zero, and only one possible cause is TRUE, say $\mathbf{b_k}$, then the probability for **e** is $\mathbf{p_k}$. If more possible causes are TRUE, P(**e**) will be greater. And if **leak** is nonzero, P(**e**) will be greater. Reducing a $\mathbf{p_i}$ always results in the same or lower P(**e**).

$\mathbf{p_i}$ can be considered the “strength” of the relation between **e** and $\mathbf{b_i}$, with zero indicating independence (link could be removed), and 1 indicating maximum effect. See Pearl88, page 184 for more information (his $q_i = 1 - p_i$). See also `NoisyAndDist`.

Example: `P (Effect | Cause1, Cause2) = NoisyOrDist (Effect, 0.1, Cause1, 0.2, Cause2, 0.4)`

NormalDist (x, μ, σ)

$= [1/(\sigma \text{ sqrt}(2\pi))] \exp(-[(x-\mu)/\sigma]^2 / 2)$

Required: `σ > 0`

The normal (Gaussian) distribution of mean **μ** and standard deviation **σ**.

The normal distribution, or approximations of it, arise frequently in nature (this is partly explained by the central limit theorem). Since it also has many convenient mathematical properties it is the most commonly used continuous distribution.

For this distribution, 68.2% of the probability is within 1 standard deviation of the mean, 95.4% is within 2 standard deviations, and 99.74% is within 3 standard deviations.

If **μ** = 0 and **σ** = 1, it is known as a "standard normal" distribution.

ParetoDist (x, a, b)

$= (a/b)\ (b/x) \wedge (a+1)$

Required: `a > 0` `b > 0`

The Pareto distribution is a power law probability distribution found in a large number of real-world situations, such as the distribution of wealth among individuals, frequencies of words, size of particles, size of towns/cities, areas burnt in forest fires, size of some fractal features etc. These are situations where there are many that are small and a few that are large (like the Pareto principle, in which 20% of the population owns 80% of the wealth).

For any value of a, the distribution is "scale-free", which means that no matter what range of x one looks at, the proportion of small to large events is the same (i.e., the slope of the curve on any section of the log-log plot is the same).

PoissonDist (k, μ)

$$= \frac{\mu^k}{k!}\ e^{-\mu}$$

Required: `k ≥ 0` `μ > 0` `k is an integer`

If events occur by a Poisson process, then the number of events that occur in a fixed time interval is described by the Poisson distribution (where **μ** is the average number of events per unit time).

round (x)

= floor (x + 1/2)

where `x` is an unrestricted real

Rounds **x** to the nearest integer. To round off to other quantities, use `roundto`.

roundto (dx, x)

= dx * floor ((x + dx/2) / dx)

where `dx > 0`

Rounds **x** to the nearest **dx**, which may be less than or greater than 1.

For example, `roundto(10,17)` rounds 17 to the nearest 10, and so it returns 20.

If **dx** `= 1`, then this is the same as the `round` function.

select0 (index, x_0, x_1, ... x_n)
select1 (index, x_1, x_2, ... x_n)

$= x_i$ **s.t.** i == index

where `index is integer,` x_i `are all the same type`
`select0: 0 ≤ index < n`
`select1: 1 ≤ index ≤ n`

Returns the value of the **x** argument at position **index**: $\mathbf{x_{index}}$

The first **x** argument is at index 0 if `select0` is used, and at index 1 if `select1` is used.

Must be passed at least 2 arguments (**index** and an **x**). See also: `member`

Example:
```
select0 (1, -6.6, 3.4, 1.26, 3.4)   returns 3.4
select1 (1, -6.6, 3.4, 1.26)        returns -6.6
```

`sign (x)` $= (x > 0)\ ?\ 1 : (x < 0)\ ?\ -1 : 0$

where `x` is an unrestricted real

Returns 1 if **x** is positive, -1 if **x** is negative, and 0 if **x** is zero. See also: `abs`

`SingleDist (k, c)` $= (k == c)\ ?\ 1 : 0$

Required: `k and c are integers`

The single point distribution indicates that **k** = **c**. The probability that **k** is any other value is 0. This is the discrete version of a Dirac delta.

`StudentTDist (x, ν)` $= \Gamma((\nu+1)/2) / [\mathrm{sqrt}(\nu\ \mathrm{pi})\ \ \Gamma(\nu/2)\ (1+x^2/\nu)^{((\nu+1)/2)}$

Required: `ν > 0`

The t-distribution or Student's t-distribution arises in the problem of estimating the mean of a normally distributed population when the sample size is small.

`TriangularDist (x, m, w)` $= (|x - a| > w)\ ?\ 0:\ \ (w - |x - a|) / w^2$

Required: `w > 0`

The graph of this distribution has a triangular shape, with the highest point at **x** = **a**, and nonzero values only from **a - w** to **a + w**.

`Triangular3Dist (x, m, w₁, w₂)` — **Triangular3Dist** (x, m, w_1, w_2)

Required: $w_1 >= 0$ $w_2 >= 0$ w_1 `&` w_2 `can't both be 0`

The pdf has a triangular shape, with the highest point at **x = m**, and nonzero value from **m -** $\mathbf{w_1}$ to **m +** $\mathbf{w_2}$.

`TriangularEnd3Dist (x, m, a, b)`

Required: a <= m b >= m b > a

The pdf has a triangular shape, with the highest point at **x = m**, and nonzero value from **a** to **b**.

`UniformDist (x, a, b)` $= 1 / (b - a)$

Required: `a < b`

This is the distribution to use when the minimum and maximum possible values for a variable are known, but within that range there is no knowledge of which value is more likely than another. It has a constant value from **x** = **a** to **x** = **b**, and zero value outside this range.

`WeibullDist (x, α, β)` $= (\alpha/\beta)\ (x/\beta)^{\alpha-1}\ \exp(-(x/\beta)^{\alpha})$

Required: `α > 0` `β > 0`

The Weibull distribution is often used for reliability models, since if the failure rate of an item (i.e., percent of the remaining ones which fail, as a function of time) is given as: $Z(t) = r\, t^{\alpha-1}$, then the distribution of item lifetimes is given by the Weibull distribution with $r = \alpha / \beta^{\alpha}$.

xor $(b_1, b_2, \ldots b_n)$ $=$ odd (NumberTrue $(b_1, b_2, \ldots b_n)$)

where b_i `are boolean`

Returns the exclusive-or of $\mathbf{b_1}, \mathbf{b_2} \ldots \mathbf{b_n}$.

This is also known as the parity function, and will return true iff an odd number of $\mathbf{b_i}$ evaluate to true. See also: `and`, `or`, `not`.

12 Bibliography

Russell, Stuart and Peter Norvig (1995) Artificial Intelligence: A Modern Approach, Prentice Hall.

Pearl, Judea (1988) Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference, Morgan Kaufmann, San Mateo, CA.

Lauritzen, Steffen L. and David J. Spiegelhalter (1988) "Local computations with probabilities on graphical structures and their application to expert systems" in J. Royal Statistics Society B, **50**(2),157-194.

Cowell, Robert G., Dawid, A.P. et al (1999) Probabilistic Networks and Expert Systems, Springer-Verlag.

Spiegelhalter, David J., Philip Dawid, et all (1993) “Bayesian analysis in expert systems” in Statistical Science, **8**(3), 219-283.

Neapolitan, Richard E. (2004) Learning Bayesian Networks, Pearson Education, Inc./Prentice Hall.

Korb, Kevin and Ann E. Nicholson (2004) Bayesian Artificial Intelligence, Chapman & Hall/CRC.

Shachter, Ross D. (1986) “Evaluating influence diagrams” in Operations Research, **34**(6), 871-882.

Shachter, Ross D. (1988) "DAVID: Influence diagram processing system for the Macintosh" in Uncertainty in Artificial Intelligence 2, John F. Lemmer and L. N. Kanal (Eds.), North-Holland, Amsterdam.

Shachter, Ross D. (1989) “Evidence absorption and propagation through evidence reversals” in Uncertainty in Artificial Intelligence 1989, 303-308.

13 Functions by Category

System

Environ	constructor	Initializes the Netica system
Environ	finalize	Signals an end to using Netica system, and frees all possible resources (e.g. memory, close any open files)
Environ	g/setArgumentChecking	Adjusts the amount that Netica functions check their arguments
Environ	getVersion/getVersionString	Gets the software version of Netica currently running
Environ	g/setMemoryUsageLimit	Adjusts the amount of memory that Netica can allocate for tables
Environ	g/setCaseFileDelimChar	The symbol to separate data fields in case files created by Netica
Environ	g/setMissingDataChar	The symbol indicating missing data in case files created by Netica

Error Handling

NeticaError	getMessage	Returns an error message for the given error report
NeticaError	isInCategory	Indicates the nature of the error (out of memory, aborted, etc.)
NeticaError	getSeverity	Returns the severity level of the given error report
NeticaError	getIdNumber	Returns the error number of the given error report
Environ	g/setArgumentChecking	Adjusts the amount that Netica functions check their arguments

File Operations

Streamer	constructor	Creates a stream for the file with the given name
Streamer	constructor	Creates a stream for reading and writing to buffers in memory
Streamer	finalize	Closes files, frees resources and deletes either type of stream
Streamer	setPassword	Sets a password to read or write encrypted files
Environ	g/setCaseFileDelimChar	The symbol to separate data fields in case files created by Netica
Environ	g/setMissingDataChar	The symbol indicating missing data in case files created by Netica
Net	write	Saves a net to a file
Net	constructor	Reads a net from a file
Net	writeFindings	Saves a net's current set of findings to a file
Net	readFindings	Reads findings from a file, and enters into a net
Caseset	writeCases	Writes all the cases to a file in CSV or UVF format
Caseset	addCases	Makes the case-set object consist of the cases located in the file
Net	reviseCPTsByCaseFile	Reads a file of cases to revise probabilities
Net	getFileName	Name of file (with full path) that net was last written to or read from

Findings (Evidence)

Node	finding().enterState	Enters a discrete finding that a node is in a given state
Node	finding().enterStateNot	Enters a discrete finding that a node is not in a given state
Node	finding().enterReal	Enters a real number finding for a continuous node
Node	finding().enterLikelihood	Enters a likelihood finding for a node (i.e. a soft finding)
Node	finding().enterGaussian	Enters a finding given by a Gaussian (normal) distribution
Node	finding().enterInterval	Enters a finding uniform over an interval, zero outside
Node	finding().setState	Enters a discrete finding, overriding any previous entry
Node	finding().setReal	Enters a real number finding, overriding any previous entry
Node	finding().getState	Returns the finding for a node, if there is one
Node	finding().getReal	Returns the real number finding entered for a continuous node
Node	finding().getLikelihood	Returns the accumulated findings for a node, as a likelihood vector
Node	finding().getKind	Returns what kind of finding was entered
Node	finding().clear	Retracts all findings for a single node
Net	retractFindings	Retracts all findings (i.e. the current case) from a net
Net	getFindingsProbability	Returns the joint probability of the findings entered so far

Compiling

Net	compile	Compiles a net for fast belief updating
Net	uncompile	Releases the resources (e.g., memory) used by a compiled net
Net	sizeCompiled	The size and speed of the compiled net (i.e. of the junction tree)
Net	reportJunctionTree	Returns a string describing the internal compiled junction tree
Net	g/setElimOrder	The node order used to guide compilation
Net	g/setAutoUpdate	Automatically propagate beliefs when findings are entered
Node	equationToTable	Builds the CPT for a node based on the equation given to it

Belief Updating and Inference

Node	getBeliefs	Returns a node's current beliefs, doing belief updating if necessary
Node	getExpectedValue	Expected value (and std dev) of a continuous or numeric-valued node
Node	getExpectedUtils	Returns the expected utility of each choice in a decision node
Node	isBeliefUpdated	Returns whether a node's beliefs have already been calculated to account for current findings
Net	g/setAutoUpdate	Automatically propagate beliefs when findings are entered
Net	getJointProbability	Returns a specified joint probability, given the findings entered
Net	getFindingsProbability	Returns the joint probability of the findings entered so far
Net	getMostProbableConfig	Finds the state for each node which results in the most probable explanation (MPE)
Net	generateRandomCase	Creates a case sampled from the net, given the current findings
Net	absorbNodes	Removes the given nodes while maintaining the joint distribution of the remaining nodes
Sensitivity	getMutualInfo	Measures the mutual information between two nodes
Sensitivity	getVarianceOfReal	Measures how much a finding at one node is expected to reduce the variance of another node
Node	calcState	Returns the state of a node calculated from its neighbors, if that can be done deterministically
Node	calcValue	Returns the numeric value of a node calculated from its neighbors, if that can be done deterministically

Learning From Data

Net	reviseCPTsByCaseFile	Reads a file of cases to revise each node's probabilities
Net	reviseCPTsByFindings	Uses the current case to revise probabilities
Learner	constructor	Creates a new object for use in learning CPTs from case data
Learner	finalize	Deletes a learning object (learner)
Learner	learnCPTs	Learn CPTs from case data, with choice of algorithm
Learner	g/setMaxIterations	The maximum number of learning-step iterations (i.e., complete passes through the data) which will be done when the learner is used
Learner	g/setMaxTolerance	The minimum change in data log likelihood between consecutive passes through the data, as a termination condition
Node	fadeCPTable	Adjusts a node's probabilities for a changing world
Node	getCPTable	Returns the results of learning
Node	getExperTable	Determines how much experience was involved in the learning
Node	setCPTable	Directly sets the probabilities (or starts them off)
Node	setExperTable	Manually sets the amount of experience (or starts it off)

Decision Nets

Node	getExpectedUtils	Returns the expected utility of each choice in a decision node
Node	setKind	Used to create decision nodes and utility nodes

Node Lists

NodeList	constructor	Creates a new (empty) list of nodes
NodeList	add	Inserts a node at the given position of a list, making it one longer
NodeList	remove	Removes the node at the given index of a list, making it one shorter
NodeList	set	Sets the Nth node of a list to a given node without changing length
NodeList	getNode	Returns the Nth node of a list (the first node is numbered 0)
NodeList	indexOf	Returns the position (index) of a node in a list, or -1 if it is not present
NodeList	size	Returns the number of nodes in a list
NodeList	copy constructor	Duplicates a list of nodes
NodeList	clear	Empties a node list without releasing the memory it uses
NodeList	finalize	Frees the memory used by a list of nodes
Net	getNode	Returns the node with the given name
Net	getNodes	Returns a list of all the nodes in the net
Node	getParents	Returns a list of the parents of a node
Node	getChildren	Returns a list of the children of a node
Node	getRelatedNodes	Finds all the nodes that bear a given relationship (such as D-connected, Markov blanket, ancestors, children, etc.) with a given node
Net	getRelatedNodes	Finds the nodes that bear a given relationship with a given set of nodes
NodeList	mapStateList	Change the order of a list of states to match a given node list

Cases (Sets of Findings)

(see also "Findings")		To enter a case into a net, and to read it out
Net	writeFindings	Saves a net's current set of findings to a file
Net	readFindings	Reads findings from a file, and enters into a net
Net	retractFindings	Retracts all findings (i.e. the current case) from a net
Net	getFindingsProbability	Returns the joint probability of the findings entered so far
Net	reviseCPTsByFindings	The current case is used to revise each node's probabilities

Net	reviseCPTsByCaseFile	Reads a file of cases to revise probabilities
Learner	learnCPTs	Learn CPTs from case data, with choice of algorithm
Net	generateRandomCase	Generates a random case in a net, according to the net's distribution
Caseset	constructor	Creates a new case-set object, initially with no cases
Caseset	finalize	Deletes and frees all resources used by a case-set object
Caseset	addCases	Searches the given database, adding cases to a case-set object
Caseset	addCases	Adds the cases located in the given case file
Caseset	writeCases	Writes all the cases in the given case-set to a file stream
NetTester	testWithCaseset	Performance tests a Bayes net with a set of cases
NodeList	mapStateList	Change the order of a list of states to match a given node list

Sensitivity to Findings (Utility-Free Value of Information)

Sensitivity	constructor	Creates an object to measure sensitivity
Sensitivity	finalize	Deletes the sensitivity measuring object
Sensitivity	getVarianceOfReal	Measure the expected reduction in variance due to a finding
Sensitivity	getMutualInfo	Measure the mutual information (entropy reduction)

Performance Testing a Net

NetTester	constructor	Creates a new tester object, for given tests on given nodes
NetTester	finalize	Deletes a tester object
NetTester	testWithCaseset	Reads the cases one-by-one, and for each it does inference and grades the Netica net, gathering statistics
NetTester	getConfusion	Returns a confusion matrix result of the testing
NetTester	getErrorRate	Returns the error rate result of the testing
NetTester	getLogLoss	Returns the logarithmic loss result of the testing
NetTester	getQuadraticLoss	Returns the quadratic loss result of the testing

Database Connectivity

DatabaseManager	constructor	Creates a new database manager object for a given database
DatabaseManager	finalize	Closes connection and deletes a database manager object
DatabaseManager	insertFindings	Adds current findings within the net into the database as a new record
Caseset	addCases	Adds the cases (or a subset) in the database to a case-set object
DatabaseManager	executeSql	Executes arbitrary SQL commands on the database
DatabaseManager	addNodes	Adds to the given net nodes that match the variables in the database

High-Level Net Modification

See also “Learning from Data”

Node	reverseLink	Reverses a single link while maintaining joint probability
Net	absorbNodes	Absorbs out (sum or max) some net nodes
Node	equationToTable	Builds a node’s CPT or function table based on its equation
Node	switchParent	Switches a link that comes from some node to come from a different node, without changing the child node or its tables
Net	duplicateNodes	Duplicates each node in a list, putting them in the same or a new net
Net	copy constructor	Duplicates a whole net (with options to skip tables, etc.)
Net	undoLastOperation	Undoes the last operation done to a net
Net	redoOperation	Call this to re-do an operation that was undone

Low-Level Net Modification

See also “Equations”, “Tables”, “Node-Sets” , “Visual Display” , and “User Data Fields”

Net	constructor	Creates a new empty net
Net	finalize	Frees all memory used by a net and all its substructures
Net	setName	Changes the name of the net
Net	setAutoUpdate	Changes whether a node does belief updating immediately
Net	setElimOrder	Provides the elimination order to be used for the next compilation
Net	setTitle	Sets the string used to title a net
Net	setComment	Attaches a comment string to the net
Net	addListener	Attaches a callback for when nodes get created, removed, etc.
Node	constructor	Creates a new node for a given net
Node	delete	Removes a node from its net, and frees the memory it required
Net	duplicateNodes	Duplicates each node in a list, putting them in same or new net
DatabaseManager	addNodes	Adds to a given net nodes that match the variables in the database
Node	setName	Changes the name of a node
Node	setTitle	Sets the string used to title a node
Node	setComment	Attaches a comment string to the node
Node	setKind	Changes whether the node is a nature, decision, utility, etc. node
Node	setStateNames	Names all the states of a node at once with a comma-delimited string
Node	state().setName	Provides a name for a state of the node
Node	state().setTitle	Sets the title of a state of the node
Node	state().setComment	Attaches a comment to the state of a node
Node	state().setNumeric	Sets a real number for a state of a discrete node
Node	setLevels	Sets threshold numbers for continuous / discrete conversion
Node	addStates	Inserts one or more states into a node’s list of states
Node	state().delete	Remove a state from a node
Node	reorderStates	Changes the order of a node’s states
Node	addLink	Adds a link from one node to another
Node	deleteLink	Removes a link from one node to another
Node	switchParent	Switches a link that comes from some node to come from a different node, without changing the child node or its tables
Node	setInputName	Sets the link’s name (to be used by the child node in its equation)
Node	addListener	Attaches a callback for when nodes get created, removed, etc.

Retrieving Net Information

See also “Equations”, “Tables”, “Node-Sets” “Visual Display” , and “User Data Fields”

Net	getName	Returns the name of the net
Net	getTitle	Returns the string which is the net's title
Net	getComment	Returns the comment associated with the net
Net	getNodes	Returns a list of all the nodes in a net
Net	getNode	Returns the node having the given name from the net
Net	getFileName	Name of file (with full path) that net was last written to or read from
Net	getAutoUpdate	Returns whether the net does belief updating immediately
Net	getElimOrder	Returns a list of the elimination order used for compiling (triangulation)
Environ	getNthNet	Can be used to return all the nets in the Netica Environ, one-by-one
Node	getNet	Returns the net containing the given node
Node	getName	Returns the name of the given node
Node	getTitle	Returns the string titling the node

Node	getComment	Returns a comment string for the node
Node	getType	Returns whether the node is for a discrete or continuous variable
Node	getKind	Returns whether the node is a nature, decision, utility, etc. node
Node	getNumStates	Returns the number of states node can take on
Node	state().getName	Returns the name of the given state
Node	state().getIndex	Returns the state number of the state with the given name
Node	state().getTitle	Returns the title of the given state
Node	state().getComment	Returns the comment of the given state
Node	state().getNumeric	Returns the real number associated with a state of a discrete node
Node	getLevels	Returns threshold numbers for continuous / discrete conversion
Node	getInputIndex	Returns the parent index of the link with the given name
Node	getParents	Returns a node list of the parents of the node
Node	getChildren	Returns a node list of the children of the node
Node	isRelated	Checks if a node has a given graphical relationship (such as D-connected, Markov blanket, ancestors, children, etc.) with another node
Node	getRelatedNodes	Finds all the nodes that bear a given relationship with a given node
Net	getRelatedNodes	Finds the nodes that bear a given relationship with a given **set** of nodes

Equations

Node	g/setEquation	Set a node's equation (expressing the node's value or CPT as a function of its parent nodes)
Node	equationToTable	Builds the node's function or CPT table from its equation
Node	g/setInputName	Defines a name for a link (to be used by the node's equation instead of the parent node's name)
Node	calcState	Calculates, if possible, the state of a node, based on its deterministic equation or table, and findings at its neighbor nodes
Node	calcValue	Calculates, if possible, the numerical value of a node, based on its deterministic equation or table, and findings at its neighbor nodes

Tables

Node	g/setCPTable	The conditional probability of the node given its parent's values
Node	g/setExperTable	Experience quantities indicating how much data was used to learn each row of the CPTable
Node	g/setStateFuncTable	Function table of a discrete deterministic node
Node	g/setRealFuncTable	Function table of a continuous deterministic node
Node	deleteTables	Removes a node's function, probability, and experience tables
Node	hasTable	Whether the node has a CPT table or function table
Node	isDeterministic	Discovers if the node is a deterministic function of its parents
NodeList	mapStateList	Useful for getting states in correct order to access a table
Node	equationToTable	Builds table from equation
Learner	learnCPTs	Performs learning of CPT tables from data
Net	reviseCPTsByFindings	Modify CPTs by learning from a single case
Net	reviseCPTsByCaseFile	Modify CPTs by learning from cases
Node	fadeCPTable	Increase uncertainty in CPT table to account for passage of time

Node-Sets

Node	addToNodeset	Adds the given node to the node-set of the given name
Node	removeFromNodeset	Removes the given node from the node-set of the given name

Node	isInNodeset	Returns whether the given node is a member of the given node-set
Net	getAllNodesets	Returns a list of all node-sets defined for this net, in priority order
Net	reorderNodesets	Re-orders the node-sets as requested, for priority during display
Net	g/setNodesetColor	Gets or sets the color used to display nodes of a given node-set

Visual Display

See also the NetPanel and NodePanel classes.

Node	visual().g/setStyle	The style to draw the node in Netica Application
Node	visual().g/setPosition	The coordinates of the center of the node in the Netica Application

User Data Fields

These are also all repeated for the Net object:

Node	user().g/setNumber	Attaches a named-field number to the node, that gets saved to file
Node	user().g/setString	Attaches a named-field string to the node, that gets saved to file
Node	user().g/setObject	Attaches a named-field Serializable object to the node, that gets saved to file
Node	user().g/setBytes	Attaches a named-field blob to the node, that gets saved to file
Node	user().removeField	Removes one of the named fields of the node
Node	user().getNthFieldName	Retrieves field-by-field info from the node by index
Node	user().g/setReference	Attaches a single arbitrary data object to the node (not saved to file)

14 Index