Netica™ API

Programmer's Library

C LANGUAGE VERSION

Reference Manual

Version 3.25 and Higher

Netica API Programmer's Library

Reference Manual

Version 3.25

October 4, 2007

Published by:

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

Contents

1 Introduction

This reference manual is for the Netica C Programmer's Library, also known as the "Netica-C API" (Application Program Interface), which is a module designed for C or C++ programmers to embed in their programs. It 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).

Besides C, special versions of the Netica API exist for the Java (also usable by Matlab), C#, Visual Basic and C++, 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.

This manual assumes that you are familiar with the C or C++ programming language. It also assumes familiarity with Bayesian belief 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-c-api@norsys.com.

1.1 Netica-C API

The Netica-C API is a complete library of C-callable functions 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-C 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 throw exceptions or take an unpredictable amount of time or memory before returning. Netica-C is threadsafe in multi-threaded environments. It may be used in conjunction with other C or C++ libraries and it won't interfere with them. It does not require any other library except the Standard C library.

Versions of the Netica API are available for MS Windows, Linux, Macintosh, some embedded systems, etc. (contact us for others), 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 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. Products such as BoundsChecker and Purify are used to make sure there are no memory leaks or other memory faults. This level of QA, combined with a careful initial design and 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-C 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. Cases may be incomplete, and may have an associated ID number and multiplicity.
- **Simulation:** Can do sampling (i.e. simulation) to generate random cases with a probability distribution matching the Bayes net. Can use a junction tree algorithm for speed, or 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. It won't throw exceptions (C++ version does).
- **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 (about 500 KB to 3 MB depending on platform/usage, 1 MB typical).
- **C Language Interface:** Usable by programs in written in any language that can call C functions, such as: C, C++, Java, Python, Perl, Visual Basic, Delphi Pascal, Lisp, Fortran or Cobol.

- **Many Platforms:** Is available for a wide range of platforms including MS Windows (95/NT to Vista), Linux, Macintosh (OS X and Classic), embedded platforms. Contact Norsys for other platforms.
- **Other Libraries:** Global symbols in Netica end with special suffixes to avoid namespace conflicts. The only other library required to use Netica API is the Standard C Library (and sometimes the standard pthreads library).
- **Object Encapsulation:** Only function calls and constants (as enumerated types) are exposed, no internal structures or variables. This makes the system much more secure and safe to use, and assists forward/backward compatibility when new versions of Netica API are released. It also makes it easier to map to object-oriented languages.
- **Memory Limiting:** You can set a bound on how much total heap space Netica API is allowed to allocate for large tables, thereby preventing virtual memory thrashing or the memory-starving of other parts of your application.
- **Memory Management Independence:** Netica will never de-allocate any array, string or structure you pass it, which was not originally created by Netica. You are responsible for de-allocating the things you create, and Netica has functions for de-allocating the things it creates.
- **More Features:** A more extensive list of features is available from: http://www.norsys.com/netica_api.html and for those features specific to the C version: http://www.norsys.com/netica_c_api.htm

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 **LicAgree.txt.**

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 `NewNeticaEnviron_ns` function. For example:

```
environ_ns *env = NewNeticaEnviron_ns ("your unique license", NULL, NULL);
```

If you do not have a license password, then you can simply supply `NULL` in place of it, in which case Netica 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 Java version (Netica-J), the C# or Visual Basic version, or the C++ version. Simply supply that license string to the appropriate Environment constructor objects 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 when you call the `InitNetica2_bn` function, it will put a warning

message in the string it returns, 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 Installation

The recommended installation steps are:

1. Obtain the file NeticaAPI_Win.zip (or the version for your OS/platform) from the CD-ROM sent to you, or from the Norsys website: http://www.norsys.com/download_api.html.

2. Unzip it on your hard drive, and it will form a directory called **NeticaAPI_C_325** (or the current version number).

3. Read the "README.txt" file that resides in the **NeticaAPI_C_306** directory**.** It contains installation information specific to your operating system and computer platform, as well as other news and notices regarding that version.

4. Click on the file **doc/webdocs/index.html** so that it opens in your web browser. It is an entry point to excellent onscreen documentation of every Netica-C function. Add a favorites bookmark to it so that you will always have it available while you are working with Netica.

5. Test your installation with the "Demo" application provided. For IDE based systems, that generally means double-clicking the Demo project (e.g. "Demo.sln" or "Demo.dsp") in the examples_c directory, then choosing the build and run commands from the menu. For command line systems it generally means changing to the **examples_c/** directory, typing `compile.sh` or `compile.bat`, and then typing: `run.sh` or `run.bat`. If the Demo program displays a welcome message and the results of some simple probabilistic inference, without declaring any errors, then your installation is probably successful.

6. The Demo project is a good starting point for developing your own applications. You may wish to duplicate it and then add your own code to it, or to "copy-and-paste" from it into your own project. Similar examples showing how to build a net from scratch, do inference, generate cases, and learn from cases are also provided in the **examples_c/** directory. If you copy from these example programs, don't forget to replace the first `NULL` in "`NewNeticaEnviron_ns (NULL, NULL, NULL)`" with your own license password, to have the full functionality of Netica.

1.4 Files Included

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

Directory	File	Description
/	• README.txt ----------	Release notes
doc	• NeticaAPIMan_C.pdf ---	This manual
	• webdocs/ ----------	The directory for Netica-C's HTML documentation tool
	• LicAgree.txt ----------	A legal document relating to the use of Netica-C
lib	• Netica.dll ----------	Netica-C dynamic link library (Windows only)
	• Netica.lib ----------	Netica-C link-step import library (Windows only)
	• libnetica.so ----------	Netica-C dynamically linked library (Linux/Unix only)
	• libnetica.a ----------	Netica-C statically linked library (Linux/Unix/Mac OS X only)
src	• Netica.h ----------	The header file to #include in your source code in order to use the Netica API A listing of this file appears as a section near the end of this manual.
	• NeticaEx.c ----------	C language source code to use in your program See the "NeticaEx.c File" section below for more information.
	• NeticaEx.h ----------	The header file to #include in your source code in order to use NeticaEx utilities
examples_c	• Demo.c ----------	A sample application to test your Netica-C installation
	• Demo.* ----------	Depending on development environment, project or IDE files to support Demo.c See the "Installation" section for details on using Demo.c to get started.
	• BuildNet.c ----------	Demonstrates building a Bayes net by function calls
	• DoInference.c ----------	Demonstrates doing probabilistic inference
	• MakeDecision.c ----------	Demonstrates building a decision net and finding an optimal decision with it
	• SimulateCases.c ----------	Demonstrates creating case instances that statistically derive from a given net
	• LearnCPTs.c ----------	Demonstrates learning from cases
	• LearnLatent.c ----------	Demonstrates learning a hidden (latent) variable from cases
	• ClassifyData.c ----------	Demonstrates naïve Bayes classification
	• TestNet.c ----------	Demonstrates testing a learned net against real world data
	• Data Files/ ----------	A directory of nets and data sets for the examples software
	• * Project/ ----------	Directories with Visual Studio projects of the examples (Windows only)
	• compile.bat (.sh) ---------	A sample batch file for compiling all of the programs in this directory (.bat for Windows, .sh for Unix/Linux/MacOSX)
	• run.bat (.sh) ----------	A sample batch file for running any of the programs in this directory

Directories:

doc/ contains manuals, onscreen (HTML) documentation, license agreements, and any other documentation.

lib/ contains the Netica-C runtime software, without which Netica-C will not function.

src/ contains header files and source software that is distributed with Netica-C. You are free to copy from these source files for your own software. We suggest that you leave the original files unmodified. These functions may change in future version of Netica.

examples_c/ contains sample programs that may be compiled and run after installation. You are free to copy from these source files for your own software. See the "Installation" section of this manual for details on using Demo.c to test your installation and get started.

NeticaEx.c File

NeticaEx.c is a file of example source code that you are free to copy and include in your programs. The "Ex" stands for "Extra", "Example", "External", "Experimental", and "Excellent!"

The "Ex" functions are a good place to look for coding examples. Indeed, many of the coding examples found in this manual and webdocs appear in NeticaEx.c.

Nothing in NeticaEx.c is required by the Netica-C API; it is all optional. You can modify the functions in this file to precisely suit your needs, and place them in your program. If you do modify them, we recommend that you document this, so that when future releases of Netica come out, and you copy new, more advanced versions of these functions from the new NeticaEx.c file, you will be able to reproduce the appropriate modifications.

All functions of Netica-C end in `_ns`, `_cs` or `_bn`, but functions defined in NeticaEx.c don't. If you put them in your own program, you may want to give them some suffix (such as `_nx`) to remind you of where they came from.

The "Ex" functions are supported by the entire community of Netica-C users, so please feel welcome to submit additional functions that you have found useful, or to suggest improvements to the ones already there.

1.5 Special Platform Considerations

Using Netica-C with Microsoft Visual Studio (V.6 or higher)

The easiest way to get started with Netica under Visual Studio is to use the "Demo" project included in the Netica-C distribution. It has a project file specifically for Visual Studio, called Demo.sln or Demo.dsw. That file is deliberately for an older version of Visual Studio, so that it can be used with any version of Visual Studio released in the last few years (when started a message may appear saying that Visual Studio is translating it to a modern format). Open that project file, compile and run. If it works successfully, then you can replace the code in Demo.c with your own code.

If instead you want to add Netica to an existing project, you simply add the appropriate files to your project, for example by choosing **Project → Add Existing Item**. You will need to add Netica.dll, Netica.lib, Netica.h, and you will probably also want to add NeticaEx.c and NeticaEx.h.

Netica will work in single threaded or multi-threaded projects, and can be used to develop applications, other DLLs, console projects, etc.

Using Netica-C with gcc under Linux/Unix/MacOS-X:

The following command, issued at the command line in the examples_c/ directory of the Netica API distribution, will invoke gcc to compile the Demo.c program found in that directory:

gcc Demo.c -o Demo -I../src -L. -L../lib -lm -lnetica –lpthread

Explanation:

- `-o Demo` means name the resulting executable `"Demo"` (as opposed to `a.out`, the default)
- `-I../src` means look in the distribution's `src/` directory for header files to #include. This is needed because `Netica.h` and `NeticaEx.h` are located there.
- `-L.` means look for libraries and object modules in the current directory. In the case of Demo.c, it is not required, but we have added it because it is commonly useful.
- `-L../lib` means look for libraries in the distribution's `lib/` directory. This is needed because `libnetica.a` and `NeticaEx.o` are found there.
- `-lm -lnetica –lpthread` means load the Standard C math library, netica library, and pthread library, all of which are required.

If you are using a dynamically loaded version of the Netica library (`libnetica.so` or `libnetica.dyn` rather than `libnetica.a`), then you must include the path to the distribution `lib/` directory on your dynamic library path. On most Unix and Linux platforms this is LD_LIB_PATH, so something like the following should be issued within your shell before attempting to run your executable.

C-shell:
```
setenv LD_LIBRARY_PATH $LD_ LIBRARY_PATH$:/home/user/Netica_C_API/lib
```

Bourne-shell:
```
LD_LIBRARY_PATH=$LD_ LIBRARY_PATH$:/home/john/Netica_C_API/lib
export LD_LIBRARY_PATH
```

Using Netica-C with Borland C++ Builder 6 under Windows:

1. Borland C++ doesn't like code for functions in its header files when building precompiled headers, so either turn off precompiled headers, or remove the code in Netica.h (which you don't need for new projects). To do that, put the following line before #include "Netica.h":

   ```
   #define NO_DEPRECATED_NETICA_FUNCS 1
   #include "Netica.h"
   ```

2. There appears to be a compiler bug in Borland C++, that thinks a const object is being modified when it isn't, so just comment out the definition of the MultiDimnIndex function in NeticaEx.c

3. The file Netica.lib is in a format (COFF) which is different from that used by Borland BCB (OMF). To create a file Netica_bcb.lib which will serve the same purpose for BCB, use the IMPLIB utility supplied by Borland (in the Borland bin folder) like this:

```
IMPLIB -c C:\netica\netica_bcb.lib c:\netica\netica.dll
```

assuming the files are stored in C:\netica. Note the case sensitivity flag -c is required.

Using Netica-C with Other Systems under Windows:

We have tried to make Netica.dll portable across many development environments. All function calls in Netica.dll use the **stdcall** calling convention (i.e., not **cdecl**). Every function name appears in two formats:

decorated e.g. _InitNetica2_bn@8

upper case e.g. INITNETICA2_BN (to be compatible with Delphi, VB, etc.)

Only function calls are exposed, no structures or variables.

Using Netica-C with Other Languages:

Many development systems using other programming languages have a C language interface (sometimes called "foreign language interface"), so programs in those languages can call Netica functions. Norsys does not directly support using Netica in those environments, bus since some members of the Netica community may have experience with such usage, we may be able to pass along to you some of their wisdom, and some helpful files. So contact us if you need assistance, and if you are successful in using Netica from such an environment, we would be very grateful to hear from you.

A few people have used Netica from **Delphi Pascal**, and we have an interface file for that. Others have used Netica from **Prolog**, **Fortran** and some have built a **Python** wrapper for Netica (which we don't currently have). The open source **Lisp** project (C-Lisp) has built an extensive Lisp interface to Netica; for more information, see http://www.norsys.com/netica_clisp_api_popup.htm. For more information on using Netica from **Matlab**, see http://www.norsys.com/netica_matlab_api_popup.htm. Of course, Netica is available natively for **Java, C#** and **Visual Basic**, as described at the beginning of the Introduction.

1.6 Function Reference Documentation

Very thorough documentation is supplied for every function in Netica-C API. Two formats are supplied:

1. The last half of this manual presents all the API functions in alphabetical order, meant for viewing in printed form.

2. The onscreen (HTML based) webdocs documentation provides very complete documentation of every function in a pleasant and productive browsing environment. To access it, simply point your browser at the **index.html** file which is found in the **doc/webdocs/** directory of this distribution.

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 (also known as DNE) files.

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

1.8 Upgrades, Support and Mailing List

New versions of Netica API are usually released every 3-9 months, and are available for download from the Norsys website (from the "Downloads" menu at www.norsys.com, choose "Netica-C API"). If you are using a license password, it will work with any new 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-C, please visit https://www.norsys.com/mailing_list.html?interests=C-API 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-C 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-c-api@norsys.com

2 Programming with Netica-C

2.1 Getting Started

First, ensure that you have correctly installed the distribution package, as was outlined in the "Installation" section.

Second, be **sure** to find the on-line HTML documentation system (click doc/webdocs/index.html), and bookmark it in your browser. You may also want to have a printed version of this manual available for reference. If it is too long to print, you may want to leave out the Function Reference chapter. Or perhaps you just want to print the "Functions by Category" pages.

Now you are ready to begin programming Netica.

If you are using Visual Studio, and you want to jump right into things and try them out before doing further reading, just go to the examples_c folder, open one of the project folders, double-click the *.sln file, build, run and then experiment with making changes to the source code. It would be a good idea to save a copy of the examples_c folder first.

Below is one of the smallest programs which properly uses Netica (sort of a "Hello World" example). We will start by understanding it, and getting it to work.

```
#include "Netica.h"
environ_ns* env;

int main (void){
   char mesg[MESG_LEN_ns];
   int res;

   //    Region A ...

   env = NewNeticaEnviron_ns (NULL, NULL, NULL);
   // replace first NULL above with your license string if desired

   res = InitNetica2_bn (env, mesg);
   printf ("%s\n", mesg);
   if (res < 0)  exit (-1);

   //    Region B ...

   res = CloseNetica_bn (env, mesg);
   printf ("%s\n", mesg);
   if (res < 0)  exit (-1);

   //    Region C ...
}
```

To get this example working, you can first put the above program in a file called "myapp.c" or "myapp.cpp" (there is a copy of it called main_ex in NeticaEx.c for copying and pasting). If you have purchased a Netica API license, you should change the first argument passed to `NewNeticaEnviron_ns` from `NULL` to the password (called the "license number" on the invoice) provided to you, so the line looks something like:

```
env = NewNeticaEnviron_ns ("+SmithJ/UCS/310-3/12345", 0, 0);
```

You can leave it as `NULL`, but then you will not obtain the full functionality of Netica.

Compile myapp.c, and link it with the Netica API library and the C Standard library. If your system has several versions of the Standard Library, you will have to use one that includes file I/O and floating point math.

Compiling and Linking under Unix

If you are using a Unix command line, you might use commands like:

```
cc -c -I../src myapp.c

cc -o myapp myapp.o -L. -L../lib -lnetica -lm
```

The first line compiles myapp.c and puts the resulting object code in myapp.o. The second line links myapp.o with libnetica.a and libm.a (the math library), and puts the resulting executable in myapp.

Compiling and Linking under Windows

If you are using a Windows command line, you might use commands like:

```
CL.EXE  /c  /I..\src  myapp.c

LINK.EXE  /LIBPATH:..\lib  Netica.lib  myapp.obj  /OUT:myapp.exe
```

The first line compiles myapp.c and puts the resulting object code in myapp.o. The second line links myapp.obj with Netica.lib and puts the resulting executable in myapp.exe.

When you run the executable (in the above example, by typing "myapp"), it should print out something similar to this:

```
Netica (AF) 3.25 Linux, (C) 1990-2007 Norsys Software Corp.

Leaving Netica.
```

Whenever you use the Netica API, the structure of your program should be similar to the example (except, of course, you may not want to call "exit" when a serious error occurs, you may not want to print out the messages, and you probably want to use subroutines instead of putting things directly in main). Regions A, B, and C in the example can contain whatever code you wish, provided regions A and C do not call any Netica function, or use any data structure originally obtained from a Netica function. Also, with the current version of Netica, Region B must not call NewNeticaEnviron_ns or InitNetica2_bn. There are a few Netica functions (which are documented as such in the "Function Reference" chapter), which can be called between the call to NewNeticaEnviron_ns and InitNetica2_bn. It is not necessary to call CloseNetica_bn if you don't want to free up the resources (e.g. memory) that Netica is using. The whole structure above can be repeated several times if desired (i.e., start up Netica, close it down, start it up again, close it down again, etc.).

2.2 Problems and Debugging

Whenever you encounter problems it is always a good idea to check if Netica has registered a descriptive error message, by calling GetError_ns. In conjunction, you may want to turn argument checking to its maximum level by calling ArgumentChecking_ns with COMPLETE_CHECK. Don't forget to check the message returned by InitNetica2_bn.

To reach Norsys technical support, email support@norsys.com. Make sure you indicate which version of Netica API you are using (see GetNeticaVersion_bn), and which platform (e.g. operating system and compiler version) you are using.

2.3 Naming Conflicts

Whenever a C library is used, naming conflicts must be considered. Two global symbols (function names or global variables) in your program cannot have the same name, whether they appear in code you have written, in the Netica library, or in some other library being linked in. For that reason libraries often restrict the beginning or ending of every symbol they declare.

All Netica global symbols end in "_ns", "_bn" or "_cs" The "_ns" symbols are those used in all Norsys products, while the "_bn" symbols are those particular to Bayes nets or decision nets, and the "_cs" are functions dealing with cases, case-sets and databases. Your program, or other libraries it includes, should not define any global symbols that end with these letters (if this is a problem, contact Norsys).

Name clashes with Netica type names or enumeration constants are more easily dealt with. If you have such a conflict, say with NEXT_CASE, you can make a new header called MyNetica.h which contains:

```
#define NEXT_CASE  NEXT_CASE_ns
#include "Netica.h"
#undef NEXT_CASE
```

and include MyNetica.h in your source code instead of Netica.h, *before* your definition of NEXT_CASE. But then be *very* careful to use NEXT_CASE_ns and not NEXT_CASE when calling Netica functions.

Most C++ and some C development systems allow you to define namespaces to help avoid naming conflicts. The Netica-C API does not use namespaces, to be compatible with the most development systems, but the Netica C++ API is defined within the "netica" namespace (and therefore doesn't limit its names to those with _ns, _bn and _cs suffixes).

In your source code, use the names for the enumeration constants defined in the Netica.h file, not just the numbers that they stand for, since future versions of Netica may define them as different numbers.

2.4 Types

There are two kinds of data types involved in using Netica API: scalar types and opaque pointers to objects. The scalar types are state_bn, prob_bn, level_bn and caseposn_bn, and are defined as int, long, float or double in the Netica.h header file. There are also some enumeration scalar types: checking_ns, errseverity_ns and nodekind_bn. In your source code, always use the names for the enumeration constants defined in the Netica.h file, not just the numbers that they stand for, since future versions of Netica may define them as different numbers. For example, if variable nk is defined as a nodekind_bn, you would write "nk = NATURE_NODE" rather than "nk = 0". Also, try to use the type names to define your variables (e.g. prob_bn or nodekind_bn) instead of what they stand for (float, int, etc.).

Whenever you are working with structured objects defined by Netica, you will do so with an *opaque pointer*. These are declared in the Netica.h header file, but the header file gives no indication of their internal structure. To set or obtain the value of one of its fields you will pass it to a Netica function for the purpose. Examples of opaque pointer types are: `net_bn`, `node_bn`, `nodelist_bn`, `environ_ns`, `report_ns` and `stream_ns`. The purpose of hiding the internal structure of these types is for object encapsulation (for example, so that you don't have to change your software to work with future versions of Netica).

2.5 Memory Management

Netica will never free any array, string or structure you pass it, that was not originally created (allocated) by Netica, and you should never use the C++ ‘delete’ or C Standard Library function 'free' to free any array, string, or structure that Netica originally created. In other words, you are responsible for freeing the things you create, and Netica is responsible for freeing the things it creates. Of course, you can control when Netica frees the structures it has created, by using functions like `DeleteNet_bn` and `DeleteNode_bn`.

Whenever this manual says that Netica returns a "non-modifiable" structure (including an object, array, or character string), it is really returning a pointer to some structure that is being maintained within the Netica system, and not a duplicated copy of the structure. If you intend to use the returned structure over an extended period of time, then you should make a duplicated copy of it, because Netica may erase the original during the course of its operation. For example, if you request the name of a node with `GetNodeName_bn`, then free the node with `DeleteNode_bn`, and then try to read the string you originally obtained, its contents will likely have been destroyed. Under multi-threading, the usual considerations apply. For example, if two or more threads are just “getting” information from a net, they won’t interfere.

If Netica is running out of memory, an error report will be generated and any subsequent calls to most Netica functions will be blocked (i.e. return without doing anything except generating an error report). Generally, the functions that are not blocked are recovery-type functions, like those that start with “Write”, “Delete”, or perhaps with "Get". Netica has a way to control the amount of memory available for memory-intensive operations, to avoid virtual memory thrashing or starving other processes (see `LimitMemoryUsage_ns`).

2.6 API Changes and Compatibility Over Time

From time-to-time the Netica-C API is revised. New functions are added, and old functions will be changed to accept additional or different parameters. Norsys is committed to keeping the API stable so

that you don't have to revise your existing source software to make it compile and link with the latest version of Netica-C. This is made possible by having a "Compatibility Section" inside the file **src**/Netica.h which allows us to translate older function prototypes and other constants and structures into the latest equivalent version. However, if you want to be sure you are not using any old Netica functions in your code, put this line before #including the Netica header:

```
#define NO_DEPRECATED_NETICA_FUNCS 1
#include "Netica.h"
```

3 Probabilistic Inference

3.1 Bayes nets and Probabilistic Inference

A Bayes net (also known as a Bayesian network, belief network, BN, BBN, 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 *link*s. 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 `GenerateRandomCase_bn` many times (say 1000), with argument `method=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.

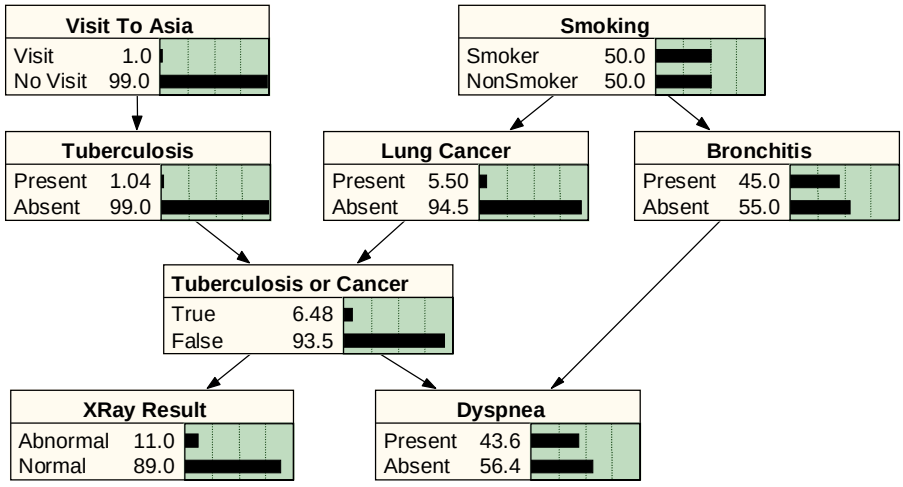
The amount of memory required by the junction tree, and the speed of belief updating are approximately proportional to each other, and are determined by the quality of the compilation. The quality of the compilation depends upon the *elimination order* used, which is a list of all the nodes in the net. 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.c**, can be found in the **examples_c/** directory of the Netica-C 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_c directory of the Netica API distribution, that will already be the case. Otherwise you should obtain the file from the "examples_c/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.c
 */

#include <stdio.h>
#include <stdlib.h>
#include "Netica.h"
#include "NeticaEx.h"

#define CHKERR  {if (GetError_ns (env, ERROR_ERR, NULL))  goto error;}

environ_ns* env;

int main (void){
   net_bn* net = NULL;
   double belief;
   char mesg[MESG_LEN_ns];
   int res;
   report_ns* err;

   env = NewNeticaEnviron_ns (NULL, NULL, NULL);
   res = InitNetica2_bn (env, mesg);
   printf ("%s\n", mesg);
   if (res < 0)  exit (-1);

   net = ReadNet_bn (NewFileStream_ns ("Data Files\\ChestClinic.dne", env, NULL),
                     NO_VISUAL_INFO);
```

```
    CHKERR

    CompileNet_bn (net);

    belief = GetNodeBelief ("Tuberculosis", "present", net);
    CHKERR

    printf ("The probability of tuberculosis is %g\n\n", belief);

    EnterFinding ("XRay", "abnormal", net);
    belief = GetNodeBelief ("Tuberculosis", "present", net);
    CHKERR

    printf ("Given an abnormal X-ray, \n\
             the probability of tuberculosis is %g\n\n", belief);

    EnterFinding ("VisitAsia", "visit", net);
    belief = GetNodeBelief ("Tuberculosis", "present", net);
    CHKERR

    printf ("Given an abnormal X-ray and a visit to Asia, \n\
             the probability of tuberculosis is %g\n\n", belief);

end:
    DeleteNet_bn (net);
    res = CloseNetica_bn (env, mesg);
    printf ("%s\n", mesg);
    return (res < 0 ? -1 : 0);

error:
    err = GetError_ns (env, ERROR_ERR, NULL);
    fprintf (stderr, "DoInference: Error %d %s\n",
             ErrorNumber_ns (err), ErrorMessage_ns (err));
    goto end;
}
```

The program starts by using NewNeticaEnviron_ns and InitNetica2_bn to initialize the system as described in the previous chapter. Next, ReadNet_bn 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 "Function Reference" chapter.

You can see that the CHKERR macro, which calls GetError_ns, is used from time to time. Any error that any Netica function detects will result in an "error report" being made and registered with the environment. You can obtain the error number or an error message from this report. It is not necessary to check for an error after every Netica function call, because the original report will not be lost, and any new errors that come along will generate new reports later in the list. You may wish to call GetError_ns directly, or in some other way than using the CHKERR macro. The C++ version of Netica API throws an exception instead.

Next, CompileNet_bn 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 GetNodeBelief is called to determine the probability tuberculosis is present:

```
belief = GetNodeBelief ("Tuberculosis", "present", net);
```

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 GetNodeBelief is then called for some other node, it would return almost immediately, because the calculated beliefs have been saved at each node. Notice that the GetNodeBelief function does not end in "_bn" or "_ns" as all Netica API functions do. That is because it is not part of the API, but is defined in the NeticaEx.c file of C source code (you can find the definitions of these functions by looking them up in the index - see the "Files Included" section of the Introduction for more information). It is defined in terms of GetNodeBeliefs_bn as a more convenient, but less efficient, way of calling that routine.

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:

```
EnterFinding ("XRay", "abnormal", net);
```

Then we use GetNodeBelief 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%.

After further testing we discover that he has lung cancer, and we enter that as a finding. The lung cancer "explains away" the abnormal X-ray, and so our probability that he has tuberculosis falls to 5.00%.

```
>DoInference.exe

  Netica (AF) 3.25 Linux, (C) 1990-2007 Norsys Software Corp.

  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

  Leaving Netica.
>
```

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 example Bayes nets, see: http://www.norsys.com/netlibrary/index.htm
- Create the file using a text editor, according to the DNET file specification.
- Write a program that creates the DNET file containing the net.
- 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 routines 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). This program, **BuildNet.c**, can be found in the **examples_c/** directory of your Netica-C installation.

```
/*
 *  BuildNet.c
 *
 *  Example use of Netica-C API to construct a Bayes net and save it to file.
 */

#include <stdio.h>
#include <stdlib.h>
#include "Netica.h"
#include "NeticaEx.h"

#define CHKERR  {if (GetError_ns (env, ERROR_ERR, NULL))  goto error;}

environ_ns* env;

int main (void){
   net_bn* net = NULL;
```

```
node_bn *VisitAsia, *Tuberculosis, *Smoking, *Cancer, *TbOrCa, *XRay;
char mesg[MESG_LEN_ns];
int res;
report_ns* err;

env = NewNeticaEnviron_ns (NULL, NULL, NULL);
res = InitNetica2_bn (env, mesg);
printf ("%s\n", mesg);
if (res < 0)  exit (-1);

net = NewNet_bn ("Built_ChestClinic", env);
CHKERR

VisitAsia =    NewNode_bn ("VisitAsia", 2, net);
Tuberculosis = NewNode_bn ("Tuberculosis", 2, net);
Smoking =      NewNode_bn ("Smoking", 2, net);
Cancer =       NewNode_bn ("Cancer", 2, net);
TbOrCa =       NewNode_bn ("TbOrCa", 2, net);
XRay =         NewNode_bn ("XRay", 2, net);
CHKERR

SetNodeStateNames_bn (VisitAsia,   "visit,   no_visit");
SetNodeStateNames_bn (Tuberculosis,"present, absent");
SetNodeStateNames_bn (Smoking,     "smoker,  nonsmoker");
SetNodeStateNames_bn (Cancer,      "present, absent");
SetNodeStateNames_bn (TbOrCa,      "true,    false");
SetNodeStateNames_bn (XRay,        "abnormal,normal");
SetNodeTitle_bn (TbOrCa, "Tuberculosis or Cancer");
SetNodeTitle_bn (Cancer, "Lung Cancer");
CHKERR

AddLink_bn (VisitAsia, Tuberculosis);
AddLink_bn (Smoking, Cancer);
AddLink_bn (Tuberculosis, TbOrCa);
AddLink_bn (Cancer, TbOrCa);
AddLink_bn (TbOrCa, XRay);
CHKERR

// WARNING: floats must be passed to SetNodeProbs, ie, 0.0 not 0

SetNodeProbs (VisitAsia, 0.01, 0.99);

SetNodeProbs (Tuberculosis, "visit",    0.05, 0.95);
SetNodeProbs (Tuberculosis, "no_visit", 0.01, 0.99);

SetNodeProbs (Smoking, 0.5, 0.5);

SetNodeProbs (Cancer, "smoker",    0.1,  0.9);
SetNodeProbs (Cancer, "nonsmoker", 0.01, 0.99);

//                    Tuberculosis Cancer
SetNodeProbs (TbOrCa, "present", "present", 1.0, 0.0);
SetNodeProbs (TbOrCa, "present", "absent",  1.0, 0.0);
SetNodeProbs (TbOrCa, "absent",  "present", 1.0, 0.0);
SetNodeProbs (TbOrCa, "absent",  "absent",  0.0, 1.0);

//                   TbOrCa  Abnormal Normal
SetNodeProbs (XRay, "true",  0.98,  0.02);
SetNodeProbs (XRay, "false", 0.05,  0.95);
```

```
    CHKERR

    WriteNet_bn (net, NewFileStream_ns ("Data Files\\Built_ChestClinic.dne", env, NULL));
    CHKERR

end:
    DeleteNet_bn (net);
    res = CloseNetica_bn (env, mesg);
    printf ("%s\n", mesg);
    return (res < 0 ? -1 : 0);

error:
    err = GetError_ns (env, ERROR_ERR, NULL);
    fprintf (stderr, "BuildNet: Error %d %s\n",
             ErrorNumber_ns (err), ErrorMessage_ns (err));
    goto end;
}
```

First, the above program creates a new net with NewNet_bn, and then adds each of the nodes with NewNode_bn. Each node represents some scalar variable of interest, either discrete or continuous. The "2" passed to NewNode_bn in the example indicates the number of states the node can take on (0 would be passed for a continuous 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 and the nodes are passed as C strings. These strings must meet the requirements of an *IDname*, which are:

- The name must be between 1 and NAME_MAX_ns (= 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 that overly restricts your expressiveness, then you can also give the object a "title" which is an unrestricted C string. Some objects can have a "comment" as well, which is also an unrestricted C string, and it would not be out of the ordinary if this were several kilobytes long. The unrestricted strings are normally in ASCII, but they may be in Unicode (UTF-16) by prefixing them with the two hex bytes 0xFEFF.

Next, the program sets the state names of the nodes using SetNodeStateNames_bn. This step is not required to do inference, but it is recommended 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. Once again the strings used for state names must conform to the requirements of an IDname. 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 AddLink_bn. A call of the form AddLink_bn (NodeP, NodeC) 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 SetNodeProbs (which is defined in NeticaEx.c as a convenient way to call SetNodeProbs_bn). The first argument in each call is the node whose probabilities we are setting. This is followed by the names of the conditioning states of its parents as C strings. Finally comes a list of numbers, being the probabilities for each of the states of the node.

For example: `SetNodeProbs (Cancer, "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, `SetNodeProbs (TbOrCa, "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 EVERY_STATE can be used with SetNodeProbs_bn; see the Function Reference for more information.

There are a couple of things to be cautious of when using SetNodeProbs. Since the function prototype uses "...", you must be very careful to pass doubles for the probabilities (e.g. passing 0 instead of 0.0 will get you in trouble). If speed is critical, and you must set large probability tables, use SetNodeProbs_bn directly instead of SetNodeProbs. For example, `SetNodeProbs (TbOrCa, "present", "absent", 1.0, 0.0);` could be accomplished by:

```
parent_states[0] = 0;   parent_states[1] = 1;      /* present absent */
probs[0] = 1.0;         probs[1] = 0.0;
SetNodeProbs_bn (TbOrCa, parent_states, probs);
```

There is an even faster way to set the whole CPT table with one function call. You call SetNodeProbs_bn, passing NULL for the array of parent states, and 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 SetNodeFuncState_bn.

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 WriteNet_bn. 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 `WriteNet_bn` to be:

```
WriteNet_bn (net,  NewFileStream_ns ("Built_ChestClinic.neta", env));
```

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.:

```
stream_ns* stream = NewFileStream_ns ("Built_ChestClinic.neta", env);
SetStreamPassword_ns (stream, "MyPassword123");
WriteNet_bn (net, 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 function documentation for `SetStreamPassword_ns`.

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, look it up in the "Function Reference" chapter.

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 *likelihood finding* (also known as "virtual evidence"). 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 SetNetAutoUpdate_bn) or if GetNodeBeliefs_bn is called between entering findings.

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

```
(a)    state_bn finding;
(b)    node_bn* node;
(c)    const prob_bn *clike, *belief;
(d)    prob_bn like[4];

(1)    like[0] = 0.6;    like[1] = 0.6;    like[2] = 1.0;   like[3] = 1.0;
(2)    EnterNodeLikelihood_bn (node, like);
(3)    EnterFindingNot_bn (node, 1);
(4)    like[0] = 0.5;    like[1] = 0.6;    like[2] = 0.0;   like[3] = 0.5;
(5)    EnterNodeLikelihood_bn (node, like);
(6)    clike = GetNodeLikelihood_bn (node);
(7)    //  EnterFinding_bn (node, 2);
(8)    belief = GetNodeBeliefs_bn (node);
(9)    finding = GetNodeFinding_bn (node);
(10)   RetractNodeFindings_bn (node);
(11)   EnterFinding_bn (node, 2);
(12)   finding = GetNodeFinding_bn (node);
(13)   clike = GetNodeLikelihood_bn (node);
```

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 there 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 GetNodeFinding_bn 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 LIKELIHOOD_FINDING.

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 GetNodeFinding_bn 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 AddDBCasesToCaseset_cs followed by WriteCaseset
- 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 C or 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.c, listed below and included in the **examples_c/** 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.c, the program which generated the above case file:

```
/*
 *  SmulateCases.c
 *
 *  Example use of Netica-C API for generating random cases that follow
 *  the probability distribution given by a Bayes net.
 */

#include <stdio.h>
#include <stdlib.h>
#include "Netica.h"
#include "NeticaEx.h"

#define CHKERR  {if (GetError_ns (env, ERROR_ERR, NULL))  goto error;}

environ_ns* env;

int main (void){
    net_bn* orig_net = NULL;
    const nodelist_bn* orig_nodes;
    const int numcases = 200;
    stream_ns* casefile = NULL;
    char mesg[MESG_LEN_ns];
    int i, res;
    report_ns* err;

    env = NewNeticaEnviron_ns (NULL, NULL, NULL);
    res = InitNetica2_bn (env, mesg);
    printf ("%s\n", mesg);
    if (res < 0)  exit (-1);

    // Read in the net created by the BuildNet.c example program
    orig_net = ReadNet_bn (NewFileStream_ns ("Data Files\\ChestClinic.dne", env, NULL),
                           NO_VISUAL_INFO);
    orig_nodes = GetNetNodes_bn (orig_net);
    SetNetAutoUpdate_bn (orig_net, 0);
    CHKERR

    remove ("Data Files\\ChestClinic.cas");
    casefile =  NewFileStream_ns ("Data Files\\ChestClinic.cas", env, NULL);
    for (i = 0;  i < numcases;  ++i){
        RetractNetFindings_bn (orig_net);
        res= GenerateRandomCase_bn (orig_nodes, 0, 20, NULL);
        if (res >= 0)
            WriteNetFindings_bn (orig_nodes, casefile, i, -1);
        CHKERR
    }

end:
```

```
    DeleteStream_ns (casefile);
    DeleteNet_bn (orig_net);
    res= CloseNetica_bn (env, mesg);
    printf ("%s\n", mesg);
    return (res < 0 ? -1 : 0);

error:
    err = GetError_ns (env, ERROR_ERR, NULL);
    fprintf (stderr, "SimulateCases: Error %d %s\n",
             ErrorNumber_ns (err), ErrorMessage_ns (err));
    goto end;
}
```

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 200 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 has a very powerful abstract class called a *case-set*. It 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:

```
caseset_cs* NewCaseset_cs (const char* name, environ_ns* env);
```

Then you add cases to the caseset. If you want them to come from a database, you use AddDBCasesToCaseset_cs, 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 stream_ns that refers to the file, using NewFileStream_ns. If you are creating the case file dynamically, it is probably much

more efficient to just create it in a memory buffer, and create the stream_ns with NewMemoryStream_ns instead. Then you add the cases within it to the caseset using:

```
void AddFileToCaseset_cs (caseset_cs* cases, const stream_ns* file, double degree, const
                          char* 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:

```
void WriteCaseset_cs (const caseset_cs* cases, stream_ns* file, const char* control);
```

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

You can use LearnCPTs_bn 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, call:

```
void DeleteCaseset_cs (caseset_cs* cases);
```

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 in it to create a caseset, then learn a Bayes net, etc. First you create a database manager (dbmgr_cs), using:

```
dbmgr_cs* NewDBManager_cs (const char* connect_str, const char* control, environ_ns* env);
```

The connection string (connect_str) 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 HTML documentation for the NewDBManager_cs function.

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

```
void ExecuteDBSql_cs (dbmgr_cs* dbmgr, const char* sql_cmnd);
```

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

```
void InsertFindingsIntoDB_bn (dbmgr_cs* dbmgr, const nodelist_bn* nodes, const char*
                              column_names, const char* tables);
```

To use the database with Netica functions such as learning from data, you create a caseset from it with:

```
AddDBCasesToCaseset_cs (caseset_cs*, dbmgr_cs* dbmgr, const char* node_names, const char*
                        column_names, const char* tables, const char* condition);
```

When you are done with the database manager, call:

```
void DeleteDBManager_cs (dbmgr_cs* dbmgr);
```

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.

```
dbmgr_cs *dbmgr =
   NewDBManager_cs ("driver=Microsoft Access Driver (*.mdb); dbq=.\\myDB.mdb; UID=dba1;",
                            "pooling", env);
caseset_cs* cases = NewCaseset_cs ("TestDBCases", env);
AddDBCasesToCaseset_cs (cases, dbmgr,
                            "Gender, Height, OwnsHouse, NumDogs"
                            "gender, height, \"Owns a house\", \"Number of dogs\"",
                            NULL,    // since the database has only one table
                            "'Owns a house' = 'yes'");
net_bn*  net = NewNet_bn ("TestDB", env);

// ... Put code to add nodes and links to net here ...
//     You could use AddNodesFromDB_bn

const nodelist_bn* nodes = GetNetNodes_bn (net);
learner_bn* learner = NewLearner_bn (COUNTING_LEARNING, env);
LearnCPTs_bn (learner, nodes, cases, 1.0);
DeleteLearner_bn (learner);
DeleteCaseset_cs (cases);
DeleteDBManager_cs (dbmgr);
```

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 ReadNetFindings_bn), it will enter them into the Bayes net as likelihood 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") likelihood 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 now be easily input into Netica (although sometimes they may mean an interval distribution, as described below).

Interval

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]`

Indicates the finding 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 numbers include the lower endpoint, but not the upper endpoint, so [0, 10] for variable X means $0 \le X < 10$. Likelihood within the interval is constant; outside the interval it is zero.

Unbounded Interval

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

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

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

Set of Possibilities

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

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

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

Indicates the finding 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. Likelihood of elements in the set is one, of those not in the set is zero.

Set of Impossibilities

Syntax: **~{s_1, s_2, … s_n}** each si is a state name, state index preceded by #, interval or unbounded interval

Examples: `~{lo}` `~{red, blue, green}` `~{#5, #7, #1}`

`~{[0, 3.5]}`

Indicates the finding 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: **{s_1 p_1, s_2 p_2, … s_n p_n}** 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 single space). The most common kind of likelihood vectors are for discrete variables, where each state is listed, followed by its probability. Any states that appear without a probability 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 the 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.

Negative Likelihood

Syntax: **~{s_1 p_1, s_2 p_2, … s_n p_n}** 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, teal .2, olive .8}`

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

The same as a set of impossibilities, but each entry is weighted with a likelihood, 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: **?** [i.e. the `syntax` is just a question mark]

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.

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 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 *query nodes*. If the links of the net correspond to a causal structure, and the query nodes are ancestors of the nodes with findings, then you could say that the net has learned to do diagnosis. If the query nodes are descendants, then the net has learned to do prediction, and if the query 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 *likelihood 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_c 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 `SetNodeExperience_bn` 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:

$$exper' = exper + degree$$

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:

$$prob_c' = (prob_c * exper + degree) / exper'$$

The other probabilities in that vector are changed by:

$$prob_i' = (prob_i * exper) / exper'$$

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 `ReviseCPTsByFindings_bn` 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 `ReviseCPTsByCaseFile_bn` 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 `ReviseCPTsByFindings_bn` 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.c, will demonstrate learning from cases. This program can be found in the **examples_c/** directory of your Netica-C 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 DeleteNodeTables_bn. 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_bn (casefile, learned_nodes, 0, 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_bn (learned_nodes, 0, 1.0);
```

to incrementally adjust the CPTs. The comment section at the bottom of LearnCPTs.c 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.c
 *
 *  Example use of Netica-C API for learning the CPTs of a Bayes net
 *  from a file of cases.
 */

#include <stdio.h>
#include <stdlib.h>
#include "Netica.h"
#include "NeticaEx.h"

#define CHKERR  {if (GetError_ns (env, ERROR_ERR, NULL))  goto error;}

environ_ns* env;
```

```
int main (void){
    net_bn *orig_net = NULL, *learned_net = NULL;
    const nodelist_bn* orig_nodes;
    nodelist_bn* learned_nodes = NULL;
    int numnodes;
    stream_ns* casefile;
    char mesg[MESG_LEN_ns];
    int i, res;
    report_ns* err;

    env = NewNeticaEnviron_ns (NULL, NULL, NULL);
    res = InitNetica2_bn (env, mesg);
    printf ("%s\n", mesg);
    if (res < 0)  exit (-1);

    /* Read in the net created by the BuildNet.c example program */
    orig_net = ReadNet_bn ( NewFileStream_ns ("Data Files\\ChestClinic.dne", env, NULL),
                            NO_VISUAL_INFO);
    orig_nodes = GetNetNodes_bn (orig_net);
    SetNetAutoUpdate_bn (orig_net, 0);
    CHKERR

    learned_net = NewNet_bn ("Learned_ChestClinic", env);
    learned_nodes = CopyNodes_bn (orig_nodes, learned_net, NULL);
    numnodes = LengthNodeList_bn (learned_nodes);

    /* Remove CPTables of nodes in learned_net, so new ones can be learned. */
    for (i = 0;  i < numnodes;  ++i)
       DeleteNodeTables_bn (NthNode_bn (learned_nodes, i));
    CHKERR

    /* Read in the case file created by the the SimulateCases.c
       example program, and learn new CPTables. */
    casefile = NewFileStream_ns ("Data Files\\ChestClinic.cas", env, NULL);
    ReviseCPTsByCaseFile_bn (casefile, learned_nodes, 0, 1.0);

    WriteNet_bn (learned_net,  NewFileStream_ns ("Data Files\\Learned_ChestClinic.dne", env,
                            NULL));
    CHKERR

end:
    DeleteNodeList_bn (learned_nodes);
    DeleteNet_bn (orig_net);
    DeleteNet_bn (learned_net);
    res= CloseNetica_bn (env, mesg);
    printf ("%s\n", mesg);
    return (res < 0 ? -1 : 0);

error:
    err = GetError_ns (env, ERROR_ERR, NULL);
    fprintf (stderr, "LearnCPTs: Error %d %s\n",
             ErrorNumber_ns (err), ErrorMessage_ns (err));
    goto end;
}

    /* ============================================================
     * This alternate way can replace the ReviseCPTsByCaseFile_bn
     * line above, if you need to filter or adjust individual cases.

    case_posn = FIRST_CASE;
    while(1){
```

```
  RetractNetFindings_bn (learned_net);
  ReadNetFindings_bn (&case_posn, casefile, learned_nodes, NULL, NULL);
  if (case_posn == NO_MORE_CASES)  break;
  ReviseCPTsByFindings_bn (learned_nodes, 0, 1.0);
  case_posn = NEXT_CASE;
  CHKERR
  }

 ============================================================ */
```

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_bn` by calling

```
learner_bn* NewLearner_bn (learn_method_bn method, const char* info, environ_ns* env);
```

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

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

```
int SetLearnerMaxIters_bn (learner_bn* algo, int max_iters);
double SetLearnerMaxTol_bn (learner_bn* algo, double log_likeli_tol);
```

Finally, you perform the learning with:

```
void LearnCPTs_bn (learner_bn* algo, const nodelist_bn* nodes, const caseset_cs*
                           cases, double degree);
```

by passing in the nodes whose CPTs you wish to modify, the data as a `caseset_cs` (see the previous chapter for instructions on creating a `caseset_cs`), 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_bn`, call:

```
void DeleteLearner_bn (learner_bn* algo);
```

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

```
stream_ns*          netfile  = NewFileStream_ns ("ParameterlessNet.dne", env, NULL);
stream_ns*          datafile = NewFileStream_ns ("Data.cas", env, NULL);
net_bn*             net      = ReadNet_bn (netfile, NO_VISUAL_INFO);
const nodelist_bn* nodes    = GetNetNodes_bn (net);
```

```
caseset_cs*         cases    = NewCaseset_cs (NULL, env);
learner_bn*         learner  = NewLearner_bn (EM_LEARNING, NULL, env);
SetLearnerMaxTol_bn (learner, 1e-5);
AddFileToCaseset_cs (cases, datafile, 1.0, NULL);
LearnCPTs_bn (learner, nodes, cases, 1.0);
DeleteLearner_bn (learner);
DeleteCaseset_cs (cases);
DeleteStream_ns (datafile);
DeleteStream_ns (netfile);
```

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 `FadeCPTable_bn`, passing it a node and 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_bn` 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. 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_bn` 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_bn` 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 `NewNetTester_bn`, 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 `TestWithCaseset_bn`, 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_bn` 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 `GetTestErrorRate_bn`, the logarithmic loss with `GetTestLogLoss_bn`, the quadratic loss with `GetTestQuadraticLoss_bn` and the whole confusion matrix with `GetTestConfusion_bn`. Be sure to see the function documentation for each of these functions, and `NewNetTester_bn` and `TestWithCaseset_bn`, 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:

```
/*
 *  NetTester.c
 *
 *  Example use of Netica-C API for testing the performance of
 *  a learned net with the net tester tool.
 */
#include <stdio.h>
#include <stdlib.h>
#include "Netica.h"
#include "NeticaEx.h"

#define CHKERR  {if (GetError_ns (env, ERROR_ERR, NULL))  goto error;}

environ_ns* env;

int main (void){
    net_bn *net = NULL;
    nodelist_bn *test_nodes = NULL, *unobsv_nodes = NULL;
    node_bn *VisitAsia, *Tuberculosis, *Smoking, *Cancer, *TbOrCa, *XRay, *Bronchitis,
                            *Dyspnea;
    tester_bn* tester = NULL;
    stream_ns* casefile = NULL;
    caseset_cs* caseset = NULL;
    char mesg[MESG_LEN_ns];
    int res;
    report_ns* err;

    env = NewNeticaEnviron_ns (NULL, NULL, NULL);
    res = InitNetica2_bn (env, mesg);
    printf ("%s\n", mesg);
    if (res < 0)  exit (-1);
    CHKERR

    net = ReadNet_bn (NewFileStream_ns ("Data Files\\ChestClinic.dne", env, NULL),
                            NO_VISUAL_INFO);
    test_nodes   = NewNodeList2_bn (0, net);
    unobsv_nodes = NewNodeList2_bn (0, net);
    VisitAsia    = NodeNamed_bn ("VisitAsia", net);
    Tuberculosis = NodeNamed_bn ("Tuberculosis", net);
    Cancer       = NodeNamed_bn ("Cancer",  net);
    Smoking      = NodeNamed_bn ("Smoking", net);
    TbOrCa       = NodeNamed_bn ("TbOrCa", net);
    XRay         = NodeNamed_bn ("XRay", net);
    Dyspnea      = NodeNamed_bn ("Dyspnea", net);
    Bronchitis   = NodeNamed_bn ("Bronchitis", net);
    CHKERR

    // The observed nodes are the factors known during diagnosis:
    AddNodeToList_bn (Cancer, test_nodes, LAST_ENTRY);

    // The unobserved nodes are the factors not known during diagnosis:
    AddNodeToList_bn (Bronchitis,   unobsv_nodes, LAST_ENTRY);
    AddNodeToList_bn (Tuberculosis, unobsv_nodes, LAST_ENTRY);
    AddNodeToList_bn (TbOrCa,       unobsv_nodes, LAST_ENTRY);

    RetractNetFindings_bn (net);  // IMPORTANT: Otherwise any findings will be part of tests
```

```
    CompileNet_bn (net);
    CHKERR
    tester = NewNetTester_bn (test_nodes, unobsv_nodes, -1);
    CHKERR

    casefile = NewFileStream_ns ("Data Files\\ChestClinic.cas", env, NULL);
    caseset = NewCaseset_cs ("ChestClinicCases", env);
    AddFileToCaseset_cs (caseset, casefile, 1.0, NULL);
    TestWithCaseset_bn (tester, caseset);
    CHKERR

    PrintConfusionMatrix (tester, Cancer);    /* defined in NeticaEx.c */
    printf ("Error rate for %s = %g %%\n\n", GetNodeName_bn (Cancer),
       GetTestErrorRate_bn (tester, Cancer) * 100.0);
    printf ("Logarithmic loss for %s = %.4g\n\n", GetNodeName_bn (Cancer),
       GetTestLogLoss_bn (tester, Cancer));
    CHKERR

end:
    DeleteCaseset_cs (caseset);
    DeleteStream_ns (casefile);
    DeleteNetTester_bn (tester);
    DeleteNodeList_bn (test_nodes);
    DeleteNodeList_bn (unobsv_nodes);
    DeleteNet_bn (net);
    CHKERR
    res= CloseNetica_bn (env, mesg);
    printf ("%s\n", mesg);
    return (res < 0 ? -1 : 0);

error:
    err = GetError_ns (env, ERROR_ERR, NULL);
    fprintf (stderr, "NetTester: Error %d %s\n",
             ErrorNumber_ns (err), ErrorMessage_ns (err));
    goto end;
}
```

And this is the output it produces:

```
Confusion matrix for Cancer:
        Present  Absent  Actual
        6        1       Present
        1        192     Absent

Error rate = 1 %

Logarithmic loss = 0.02794
```

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, `NewNode_bn` and `AddLink_bn` can introduce new variables or dependencies, and `DeleteNode_bn` and `DeleteLink_bn` can remove them.

Almost every property of nets and nodes can be altered. Even decision nodes can be converted to nature nodes (`SetNodeKind_bn`), 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 (`AddNodeStates_bn`), removed (`RemoveNodeState_bn`), or the order of the states may be changed (`ReorderNodeStates_bn`). In each case the tables of the nodes being changed, and the tables of their children, will be appropriately modified.

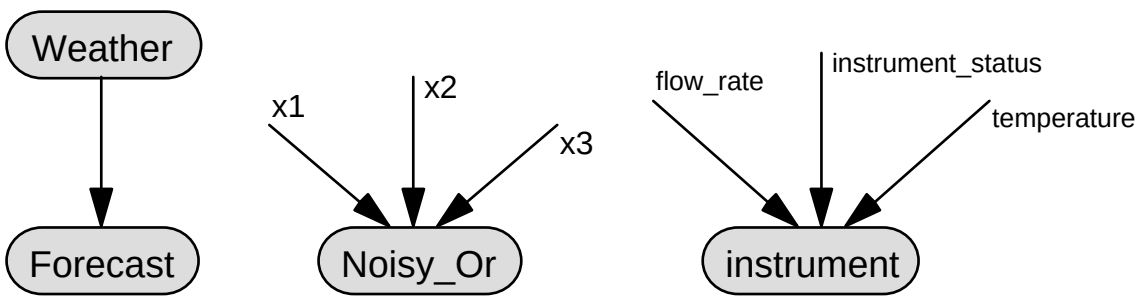
The node tables themselves may be modified. Perhaps CPTs need to be changed based on frequency data that is calculated externally. Or perhaps the utility tables of utility nodes are modified based on preference information about a particular end-user, and then new optimal decisions found. The most common 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: `SetNodeProbs_bn`, `SetNodeFuncState_bn`, `SetNodeFuncReal_bn`, `EquationToTable_bn` and `DeleteNodeTables_bn`.

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 which style to display each node in (e.g. Belief Bars, Labeled Box or Hidden). The functions to use are: `SetNodeVisPosition_bn` and `SetNodeVisStyle_bn`.

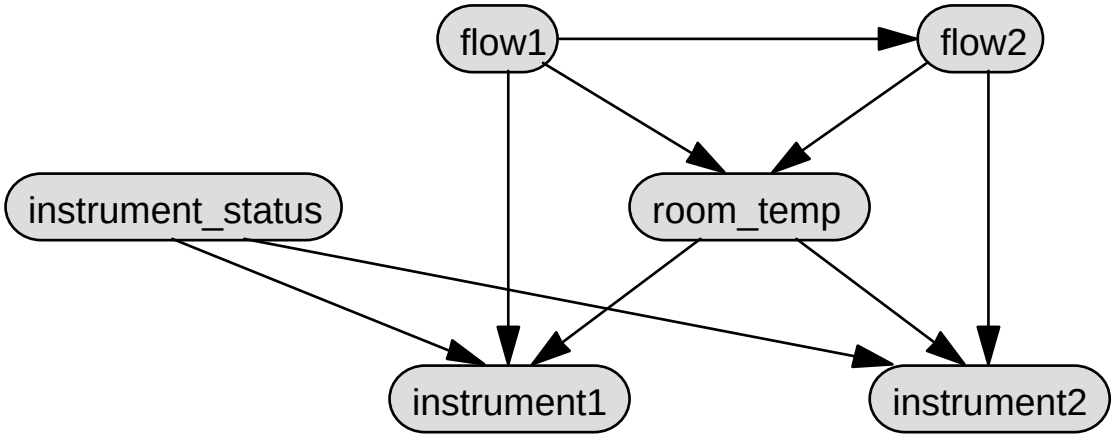
7.2 Node Libraries

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 `NewNet_bn`. Nodes and subnets can be copied to it using `CopyNodes_bn`, 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 `CopyNodes_bn` 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, see `GetNodeKind_bn` in the Function Reference for a method.

When you want to use something in the library, you call `CopyNodes_bn` again, this time to duplicate from the library into the new net. Then you connect up any disconnected links with `SwitchNodeParent_bn`, 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 = NewNet_bn ("net", env);
flow = NewNode_bn ("flow_rate", 0, net);
temp = NewNode_bn ("temperature", 0, net);
broken = NewNode_bn ("instrument_status", 5, net);
instrument = NewNode_bn ("instrument", 0, net);

AddLink_bn (flow, instrument);
AddLink_bn (temp, instrument);
AddLink_bn (broken, instrument);

// .....
// <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.

libnet = NewNet_bn ("library", env);
DuplicateNode (instrument, libnet);          // defined in NeticaEx
WriteNet_bn (libnet,  NewFileStream_ns ("Library.dnet", env, NULL));

DeleteNet_bn (net);
DeleteNet_bn (libnet);
```

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:

```
appnet = NewNet_bn ("measure_flows", env);
flow1 = NewNode_bn ("flow1", 0, appnet);
flow2 = NewNode_bn ("flow2", 0, appnet);
rtemp = NewNode_bn ("room_temp", 0, appnet);
status = NewNode_bn ("instrument_status", 5, appnet);

// .....
// <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 = ReadNet_bn ( NewFileStream_ns ("Library.dnet", env), NO_VISUAL_INFO);
instrument1 = DuplicateNode (GetNodeNamed_bn ("instrument", libnet), appnet);
instrument2 = DuplicateNode (GetNodeNamed_bn ("instrument", libnet), appnet);
```

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

SwitchNodeParent_bn (GetInputNamed_bn ("flow_rate", instrument1), instrument1, flow1);
SwitchNodeParent_bn (GetInputNamed_bn ("temperature", instrument1), instrument1, rtemp);
SwitchNodeParent_bn (GetInputNamed_bn ("instrument_status", instrument1), instrument1,
                     status);

SwitchNodeParent_bn (GetInputNamed_bn ("flow_rate", instrument2), instrument2, flow2);
SwitchNodeParent_bn (GetInputNamed_bn ("temperature", instrument2), instrument2, rtemp);
SwitchNodeParent_bn (GetInputNamed_bn ("instrument_status", instrument2), instrument2,
                     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.

7.3 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 *query 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 `AbsorbNodes_bn`. First enter the permanent findings for the context nodes. Then make a list of all the nodes except the findings nodes and the query nodes, and pass it to `AbsorbNodes_bn`. 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 query node as an ancestor. This means that if you use `ReverseLink_bn` to make all the findings nodes ancestors of all the query 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 query 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 AbsorbNodes_bn or ReverseLink_bn (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 query 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.

7.4 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 query node. The resulting probability distribution for that node can be obtained with GetNodeProbs_bn, 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 GetNodeBeliefs_bn.

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.

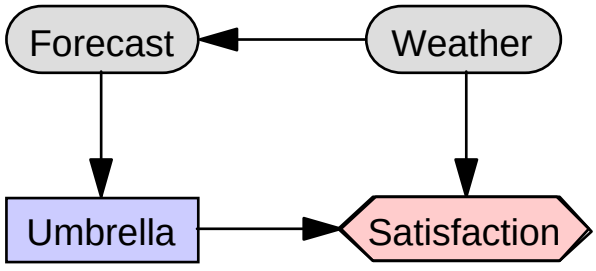
When using AbsorbNodes_bn for decision nets, the decision nodes must have no-forgetting links, and if the list of nodes to absorb does not include all the nodes in the net, it must consist of a descendant subnet (see Shachter86, Shachter88 and Shachter89 for definitions and details of the algorithm used). If there are missing no-forgetting links or missing descendants in the list of nodes to absorb, then AbsorbNodes_bn 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.c, 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_c/** 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 NewNode_bn, it starts off as a nature node. Here we change Umbrella into a decision node, and Satisfaction into a utility node using SetNodeKind_bn. NewNode_bn 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 SetNodeFuncReal to build up the relations of a deterministic node instead of SetNodeProbs, but it works in a similar fashion.

```
/*
 *  MakeDecision.c
 *
 *  Example use of Netica-C API to build a decision net and choose an
 *  optimal decision with it.
 */
#include <stdio.h>
#include <stdlib.h>
#include "Netica.h"
#include "NeticaEx.h"

#define CHKERR  {if (GetError_ns (env, ERROR_ERR, NULL))  goto error;}

environ_ns* env;

int main (void){
   net_bn* net = NULL;
   node_bn *weather, *forecast, *umbrella, *satisfaction;
   state_bn fs, decision;
   const util_bn* utils;
   char mesg[MESG_LEN_ns];
```

```
int res;
report_ns* err;

env = NewNeticaEnviron_ns (NULL, NULL, NULL);
res = InitNetica2_bn (env, mesg);
printf ("%s\n", mesg);
if (res < 0)  exit (-1);

net = NewNet_bn ("Umbrella", env);
CHKERR

weather  = NewNode_bn ("Weather",  2, net);
forecast = NewNode_bn ("Forecast", 3, net);
umbrella = NewNode_bn ("Umbrella", 2, net);
SetNodeKind_bn (umbrella, DECISION_NODE);
satisfaction = NewNode_bn ("Satisfaction", 0, net);
SetNodeKind_bn (satisfaction, UTILITY_NODE);
CHKERR

SetNodeStateNames_bn (forecast,"sunny,        cloudy,   rainy");
SetNodeStateNames_bn (weather, "sunshine,     rain");
SetNodeStateNames_bn (umbrella,"take_umbrella,dont_take_umbrella");
CHKERR

AddLink_bn (weather,  forecast);
AddLink_bn (forecast, umbrella);
AddLink_bn (weather,  satisfaction);
AddLink_bn (umbrella, satisfaction);
CHKERR

SetNodeProbs (weather, 0.7, 0.3);

//                                  forecast
//                        weather |  sunny  cloudy rainy
SetNodeProbs (forecast, "sunshine", 0.7,   0.2,   0.1);
SetNodeProbs (forecast, "rain",     0.15,  0.25,  0.6);

//                                   weather     umbrella
SetNodeFuncReal (satisfaction,  20, "sunshine", "take_umbrella");
SetNodeFuncReal (satisfaction, 100, "sunshine", "dont_take_umbrella");
SetNodeFuncReal (satisfaction,  70, "rain",     "take_umbrella");
SetNodeFuncReal (satisfaction,   0, "rain",     "dont_take_umbrella");
CHKERR

CompileNet_bn (net);

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

EnterFinding ("Forecast", "sunny", net);
utils = GetNodeExpectedUtils_bn (umbrella);  // expected utility given current findings

printf ("If the forecast is sunny, expected utility of %s is %f, of %s is %f\n",
   GetNodeStateName_bn (umbrella, 0), utils[0],
   GetNodeStateName_bn (umbrella, 1), utils[1]);
CHKERR

RetractNetFindings_bn (net);
EnterFinding ("Forecast", "cloudy", net);
utils = GetNodeExpectedUtils_bn (umbrella);
```

```
    printf ("If the forecast is cloudy, expected utility of %s is %f, of %s is %f\n\n",
        GetNodeStateName_bn (umbrella, 0), utils[0],
        GetNodeStateName_bn (umbrella, 1), utils[1]);
    CHKERR

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

    RetractNetFindings_bn (net);
    GetNodeExpectedUtils_bn (umbrella);   // causes Netica to recompute decision tables,
                            given current findings (which in this case are no findings)

    for (fs = 0;  fs < GetNodeNumberStates_bn (forecast);  ++fs){
        decision = GetNodeFuncState_bn (umbrella, &fs);
        printf ("If the forecast is '%s', the best decision is %s.\n",
                GetNodeStateName_bn (forecast, fs),
                GetNodeStateName_bn (umbrella, decision));
    }
    CHKERR

end:
    DeleteNet_bn (net);
    res= CloseNetica_bn (env, mesg);
    printf ("%s\n", mesg);
    return (res < 0 ? -1 : 0);

error:
    err = GetError_ns (env, ERROR_ERR, NULL);
    fprintf (stderr, "MakeDecision: Error %d %s\n",
             ErrorNumber_ns (err), ErrorMessage_ns (err));
    goto end;
}
```

Once the net is built, the program calls CompileNet_bn, and then GetNodeExpectedUtils_bn 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 GetNodeFuncState_bn to access this decision function, and prints out the following:

```
Netica (AF) 3.08 Win, (C) 1990-2005 Norsys Software Corp.

If the forecast is sunny, expected utility of take_umbrella is 24.205606,
of dont_take_umbrella is 91.588783
If the forecast is cloudy, expected utility of take_umbrella is 37.441860,
of dont_take_umbrella is 65.116280

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.

Leaving Netica.
```

Note that GetNodeExpectedUtils_bn or GetNodeBeliefs_bn must be called before GetNodeFuncState_bn 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 Special Topics

9.1 Node Lists

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. Netica's node list object is nodelist_bn, and there are a small set of functions specifically for building and operating on them.

Netica distinguishes between constant and modifiable node lists. Netica functions which have prototypes showing they return "const nodelist_bn* " return constant node lists (e.g., GetNetNodes_bn, GetNodeParents_bn), which you should not attempt to modify or delete (Netica will just generate an error if you try). Modifiable node lists are created by NewNodeList2_bn or DupNodeList_bn, and can be used with any Netica function requiring a node list. When done with them, they should always be deleted with DeleteNodeList_bn. To create a modifiable node list from a constant one, use DupNodeList_bn.

You can add a node to a modifiable node list with:

```
void AddNodeToList_bn (node_bn* node, nodelist_bn* nodes, int index);
```

which will insert the node into the list at position *index*, or at the end if *index* = LAST_ENTRY. The function RemoveNthNode_bn works the same way, but removes the node instead. The function:

```
void SetNthNode_bn (nodelist_bn* nodes, int index, node_bn* node);
```

replaces the node at position *index* with the specified node, and will generate an error if *index* is out-of-bounds. To make a node list empty, use ClearNodeList_bn.

For constant or modifiable node lists, you can obtain their length with LengthNodeList_bn, and obtain the node at a given index position with:

```
node_bn* NthNode_bn (const nodelist_bn* nodes, int index);
```

The inverse function, which provides the index of a given node, is:

```
int IndexOfNodeInList_bn (const node_bn* node, const nodelist_bn* nodes, int start_index);
```

where for *start_index* you usually pass zero, but you can pass a higher index to find multiple occurrences of the node if the list has multiple occurrences. To find a node by name, see GetNodeNamed_bn.

9.2 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 function:

```
void GetRelatedNodes_bn (nodelist_bn* related_nodes,
                         const char* relation,
                         const node_bn* node);
```

To use it, you pass the node, the relation you desire as a C 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 boundary of node_A, you could use:

```
nodelist_bn* mb = NewNodeList2_bn (0, net);
GetRelatedNodes_bn (mb, "markov_boundary", node_A);
```

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

The allowed relation strings are: "parents", "children", "ancestors", "descendents", "connected", "markov_boundary", 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.

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

"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).

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_bn* ad = NewNodeList2_bn (0, net);
GetRelatedNodes_bn (ad, "ancestors", node_A);
GetRelatedNodes_bn (ad, "intersection, descendents ", node_B);
```

If you want to find all the nodes that are related to a whole *group* of nodes, you use GetRelatedNodesMult_bn. It works the same as GetRelatedNodes_bn, except that instead of passing in a single node for the argument *node*, it takes a list of nodes.

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 IsNodeRelated_bn, like so:

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

9.3 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 is just held until you ask for it back. The types of Netica objects that you can attach data to are: nodes (node_bn), nets (net_bn) and the global environment (environ_ns).

There are two different ways of attaching data. One is to attach to the Netica object a single C pointer. That pointer can point to whatever you wish, perhaps a large object with many fields. When the Netica object is duplicated or saved to file, the pointer you have attached will be ignored. Only one such pointer can be attached to each object. The relevant functions for attaching and retrieving data in this way are: SetNodeUserData_bn, GetNodeUserData_bn, SetNetUserData_bn, GetNetUserData_bn, SetEnvironUserData_ns and GetEnvironUserData_ns. Representative prototypes are: (the "kind" arguments are not used):

```
void SetNodeUserData_bn (node_bn* node, int kind, void* data);
void* GetNodeUserData_bn (const node_bn* node, int kind);
```

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: (the "kind" arguments are not used):

```
void SetNodeUserField_bn (node_bn* node, const char* name, const void* data, int length, int kind);
const char* GetNodeUserField_bn (const node_bn* node, const char* name, int* length, int kind);
```

To set a user field you pass a name for the field, a pointer to your data, and the length of your data in bytes. When you later call the function to recover your data, you pass in the name you gave it, and Netica will return both a pointer to a copy of the data and how many bytes it is. Since the Netica functions just

treat your data as a blob of bytes, you can use them for any kind of data. However, NeticaEx provides some convenience functions particularly adapted to setting and getting strings and numbers:

```
void SetNodeUserString (node_bn* node, const char* fieldname, const char* str);
void SetNodeUserInt (node_bn* node, const char* fieldname, int num);
void SetNodeUserNumber (node_bn* node, const char* fieldname, double num);

const char* GetNodeUserString (node_bn* node, const char* fieldname);
long GetNodeUserInt (node_bn* node, const char* fieldname);
double GetNodeUserNumber (node_bn* node, const char* fieldname);
```

An advantage of using the convenience functions is that Netica will store your data in the file in a platform independent way. So if you are planning to create a file using one operating system or processor type, and read it back with another, then just storing the bytes of a floating point number using `SetNodeUserField_bn` may create a problem, but using `SetNodeUserNumber` will be fine.

If you wish to find all the user fields defined for some node or net, you can iterate through them with `GetNodeNthUserField_bn` or `GetNetNthUserField_bn`.

9.4 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.

During diagnosis, you may wish to know which nodes will be the most informative in crystallizing the beliefs of the most probable fault nodes. Obviously, that will change as findings arrive, so it may need to be recomputed at each stage. 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. In real-world modeling, such as environmental modeling, 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* (also known as the "query 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 sensv_bn object using the function:

```
sensv_bn* NewSensvToFinding_bn (const node_bn* Qnode, const nodelist_bn* Vnodes,
                                int what_find);
```

You pass it the target node Qnode, and a list of the varying nodes Vnodes. Later you will be able to use the sensv_bn object returned to find the sensitivity of Qnode to each of the nodes in Vnodes. You also pass what_find 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_bn, ENTROPY_SENSV if you wish to be able to call GetMutualInfo_bn, or their bitwise-or to be able to use both. Then you obtain the actual sensitivity numbers by calling one of:

```
double GetMutualInfo_bn (sensv_bn* s, const node_bn* Vnode);
double GetVarianceOfReal_bn (sensv_bn* s, const node_bn* Vnode);
```

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 Qnode (used in the construction of the sensv_bn) with respect to the Vnode passed in. The first time it is called, it takes longer to return, since it is calculating the results for all the Vnodes that were used in the construction of the sensv_bn (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_bn to efficiently find 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 sensv_bn object, delete it using DeleteSensvToFinding_bn.

Currently Netica's sensitivity analysis only works 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.

9.5 Random Case Generation

Netica can be used to generate "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 them 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 function to use is:

```
int GenerateRandomCase_bn (const nodelist_bn* nodes, int method, double num, void* gen);
```

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.c example program in the "Findings and Cases" chapter.

9.6 Listeners

Sometimes it is useful for Netica to alert your program when certain events occur, such as when a node is deleted, and that is accomplished by a "callback function". When the event occurs, Netica will call a function that you supply. The prototype of that function is:

```
int callback (const node_bn* node, eventtype_ns what, void* object, void* info);
```

When Netica calls this function, it will pass your program the `node` to which the event occurred, `what` the event was, an `object` pointer that is your reference to some object in your program that you pass to Netica when you register the callback, and some miscellaneous `info` that provides more information on the event, depending on what it was.

To prepare Netica so that it will call your callback function when the event occurs, you must register your callback using the function:

```
void AddNodeListener_bn (node_bn* node, int callback (), void* object, int filter);
```

For `node` you pass the node you want to be alerted to, or `null` if you want alerts on all nodes. For `object`, you pass the object pointer you want returned to you when the callback is made. For `filter`, pass -1; it is just for future use. Each node can have as many callbacks registered as desired; they will all be called. For each value of `object`, you can only have one callback. If you wish to remove the callback (that is "un-register" it), then call `AddNodeListener_bn` again with the same value for `object` but `NULL` for callback.

There is also a similar function to register callback functions for events on nets:

```
void AddNetListener_bn (net_bn* net, int callback (), void* object, int filter);
```

If you have a single-threaded program that completely controls what happens to nodes and nets, it would probably be better for you to just call your desired functions directly, and not use Netica's callback mechanism. It is really meant for multi-threaded situations involving separate modules, or especially when different processes operate on a single Bayes net. For example, using the new link between Netica Application and Netica API, you may want to be alerted when the GUI user has deleted a node in a Bayes net shared between your program and the GUI.

The listener functions have been recently released in Netica, and so there are not yet many events types they support. If you require assistance in using them, please contact support@norsys.com.

10 Equations

The relation between a node and its parent nodes can be entered 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 `EquationToTable_bn`) 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 `CalcNodeValue_bn` and `CalcNodeState_bn`). 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 probabilistic sampling (calling `GenerateRandomCase_bn` with `method=FORWARD_SAMPLING`).

10.1 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.

10.2 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 usually 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 C (C++/Java) 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 C/C++/Java 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 C/C++/Java is used for 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.

10.3 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 $2 \leq B < 6$, and the last covers the remaining cases (i.e. $B \geq 6$). 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.

10.4 Converting an Equation to a Table

As we noted 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 **SetNodeLevels_bn** 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 **SetNodeEquation_bn**, passing in the node and the equation string.

3. Finally, call **EquationToTable_bn**. 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, it leaves the cursor on the problem when it gives the error message. From Netica Application's menu, you can choose "Equation To Table" to check if there is going to be any problem with that function, and conveniently view the resulting CPT to see if it is what you expect.

10.5 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 LimitMemoryUsage_ns.

10.6 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 `SetNodeInputName_bn`. 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.

10.7 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 `SetNodeLevels_bn`), 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.

10.8 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 `SetNodeKind_bn`. 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.

10.9 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 when it gives 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 `DeleteNodeTables_bn`, before saving it to file. Later, when you restore the net from file, you call `EquationToTable_bn` to fully restore them.
- If you need to define intermediate variables to simplify the equations, implement them as new (intermediate) nodes.

10.10 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 useful to use more than one node to represent a continuous variable, and discretize each 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.

10.11 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
cos (x)	cosine
tan (x)	tangent
asin (x)	arc sine
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, μ, σ)
LognormalDist (x, η, ϕ)
ExponentialDist (x, λ)
GammaDist (x, α, β)
WeibullDist (x, α, β)
BetaDist (x, α, β)
Beta4Dist (x, α, β, c, d)
CauchyDist (x, μ, σ)
LaplaceDist (x, μ, β)
ExtremeValueDist (x, μ, σ)
ParetoDist (x, a, b)
ChiSquareDist (x, ν)
StudentTDist (x, ν)
FDist (x, ν_1, ν_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, k1, p1, k2, p2, ... km, pm)
NoisyOrDist (e, leak, b1, p1, b2, p2, ... bn, pn)
NoisyAndDist (e, inh, b1, p1, b2, p2, ... bn, pn)
NoisyMaxTableDist (...)
NoisySumTableDist (...)
```

10.12 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 reals

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 (x0, x1, ... xn)` = i s.t. ($x_i \geq x_j$) for all j
`argmax1 (x1, x2, ... xn)`

where x_i are unrestricted reals

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) = i s.t. ($x_i \le x_j$) for all j
argmin1 (x_1, x_2, ... x_n)

where x_i are unrestricted reals

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

Required: `α > 0` `β > 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 ≤ x ≤ 1` `α > 0` `β > 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 ≤ k ≤ 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 ≤ k ≤ n, and 0 ≤ p ≤ 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)` 

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

Required: `x ≥ 0` `ν > 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)`

$= (x < \text{min}) ? \text{min} : (x > \text{max}) ? \text{max} : x$

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)`

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

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)`

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

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, ν_1, ν_2)

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 **α** = 1, this is the exponential distribution (`ExponentialDist`) with λ = `1 / β`. For **β** = 2, this is the chi-square distribution (`ChiSquareDist`) with degrees of freedom ν = `2 α`.

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 (x1, x2, ... xn)` $= (x_1 < x_2)\ \&\&\ (x_2 < x_3)\ \&\&\ \ldots\ \&\&\ (x_{n-1} < x_n)$

where x_i are unrestricted reals

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 (x1, x2, ... xn)` $= (x_1 \le x_2)\ \&\&\ (x_2 \le x_3)\ \&\&\ \ldots\ \&\&\ (x_{n-1} \le x_n)$

where x_i are unrestricted reals

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)`  $= (1/(2\beta)) \exp(-|x-\mu|/\beta)$

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)` $= \log(n\,(n-1)\,(n-2)\ldots 1)$

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: $\phi > 0$

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, ... x_n)

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

where x_i are unrestricted reals

Returns the maximum of x_1, x_2, … 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 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, ... x_n)

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

where x_i are unrestricted reals

Returns the minimum of x_1, x_2, … 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, ... n_n)

= $(n_1 + n_2 + ... n_n)! / (n_1! * n_2! * ... n_n!)$

where $n_i \geq 0$ n_i are integers

Returns the number of ways an $(n_1+n_2+...n_n)$ sized set of distinct elements can be partitioned into sets of size n_1, n_2, … 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 k_i >= 0 0 <= p_i <= 1 sum p_i != 0 `bc boolean` `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 p_i (i=1..m), and we are interested in the number of occurances of each outcome (k_i), given that a total of n trials are performed.

To create a multinomial distribution between the 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 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 p_i is one, but Netica will just normalize the p_i if that is not the case.

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

Each of the k_i separately has a binomial distribution with parameters n and pi, and because of the constraint that the sum of the 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 reals

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** → ∞, (1-**p**) → 0, **n**(1-**p**) → λ, 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 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 p_i are the probability that b_i will be required to cause **e**.

If **inh** is zero, and only one possible requirement is FALSE, say b_k, then the probability for **e** is 1- 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 p_i always results in the same or higher P(**e**).

p_i can be considered the "strength" of the relation between **e** and 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 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 p_i are the probability that **e** will occur if b_i is TRUE in isolation.

If **leak** is zero, and only one possible cause is TRUE, say b_k, then the probability for **e** is 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 p_i always results in the same or lower P(**e**).

p_i can be considered the "strength" of the relation between **e** and 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/(σ sqrt(2π))] exp (-[(x-μ)/σ]^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) ^ (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) = x_i **s.t.** i == index
select1 (index, x_1, x_2, ... x_n)

where `index is integer, xi 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)  = Γ((ν+1)/2) / [sqrt(ν pi) Γ(ν/2) (1+x^2/ν)^((ν+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, 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)  = (α/β) $(x/\beta)^{\alpha-1}$ 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, ... b_n) = odd (NumberTrue (b_1, b_2, ... b_n))

where b_i are boolean

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

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.

11 Bibliography

Charniak, Eugene (1991) "Bayesian networks without tears" in *AI Magazine* (Winter 1991), **12**(4), 50-63.

Cover, Thomas M. and Joy A. Thomas (1991) *Elements of Information Theory*, John Wiley and Sons, Inc.

Cowell Robert G., A. Philip Dawid, Steffen L. Lauritzen and David J. Spiegelhalter (1999) *Probabilistic Networks and Expert Systems*, Springer.

Henrion, Max, John S. Breese and Eric J. Horvitz (1991) "Decision Analysis and Expert Systems" in *AI Magazine* (Winter 1991), **12**(4), 64-91.

Jensen, Finn V. (2001) *Bayesian Networks and Decision Graphs*, Springer.

Korb, Kevin B. and Ann E. Nicholson (2004) *Bayesian Artificial Intelligence*, Chapman & Hall.

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.

Matheson, James E. (1990) "Using Influence diagrams to value information and control" in *Influence Diagrams, Belief Nets and Decision Analysis*, Robert M. Oliver and J. Q. Smith (eds.), John Wiley & Sons.

Neapolitan, Richard E. (2004) *Learning Bayesian Networks*, Prentice Hall.

Pearl, Judea (1988) *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufmann, San Mateo, CA. 2nd edition 1991.

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

Shachter, Ross D. (1986) "Evaluating influence diagrams" in *Operations Research*, **34**(6), 871-882.

Shachter, Ross D. (1988) "Probabilistic inference and influence diagrams" in *Operations Research*, **36**(4), 589-604.

Shachter, Ross D. (1989) "Evidence absorption and propagation through evidence reversals" in *Proc. of the Fifth Workshop on Uncertainty in Artificial Intelligence* (Windsor, Ont.), 303-308. Later republished in: Henrion, Max (ed.) (1991) *Uncertainty in Artificial Intelligence 5*, North-Holland.

Smith, James E., Samuel Holtzman and James E. Matheson (1993) "Structuring conditional realtionships in influence diagrams" in *Operations Research*, **41**(2), 280-297.

Spiegelhalter, David J., A. Philip Dawid, Steffen L. Lauritzen and Robert G. Cowell (1993) "Bayesian analysis in expert systems" in *Statistical Science*, **8**(3), 219-283.

12 Netica.h Header File

```
/*
 * Netica.h
 *
 * Header file for Netica API, version 3.25 and greater.
 *
 * When used for Netica DLL, DYNAMIC_LINK_ns should be #defined below,
 * but when used for static linking it shouldn't.
 *
 * Copyright (C) 1992-2007 by Norsys Software Corp.
 * This file may be included as part of any software project, provided that
 * project doesn't pass NewNeticaEnviron_ns an unauthorized license string.
 */

#ifndef __NETICA_C_H
#define __NETICA_C_H

/* #define DYNAMIC_LINK_ns 1 /**/  /* Comment out, unless this file is used for Netica DLL */

#ifdef __cplusplus
extern "C" {
#endif

#ifdef DYNAMIC_LINK_ns
#define IMPORT(typ) __declspec (dllimport) typ __stdcall
#define IMPORT_VAR  __declspec (dllimport)
#else
#define IMPORT(typ) typ
#define IMPORT_VAR
#endif


#define  UNDEF_DBL    GetUndefDbl_ns()
#define  INFINITY_ns  GetInfinityDbl_ns()

typedef enum {NO_CHECK=1, QUICK_CHECK, REGULAR_CHECK, COMPLETE_CHECK, QUERY_CHECK=-1} checking_ns;

typedef enum {NOTHING_ERR=1, REPORT_ERR, NOTICE_ERR, WARNING_ERR, ERROR_ERR, XXX_ERR}
                                  errseverity_ns;

typedef enum {OUT_OF_MEMORY_CND=0x08, USER_ABORTED_CND=0x20, FROM_WRAPPER_CND=0x40,
                                  FROM_DEVELOPER_CND=0x80, INCONS_FINDING_CND=0x200} errcond_ns;

typedef enum {CREATE_EVENT=0x01, DUPLICATE_EVENT=0x02, REMOVE_EVENT=0x04} eventtype_ns;
```

```
typedef enum {CONTINUOUS_TYPE=1, DISCRETE_TYPE, TEXT_TYPE} nodetype_bn;

typedef enum {NATURE_NODE=1, CONSTANT_NODE, DECISION_NODE, UTILITY_NODE, DISCONNECTED_NODE,
                                ADVERSARY_NODE} nodekind_bn;

enum {REAL_VALUE = -25, STATE_VALUE = -20, GAUSSIAN_VALUE = -15, INTERVAL_VALUE = -10,
                                STATE_NOT_VALUE = -7, LIKELIHOOD_VALUE, NO_VALUE = -3};

enum {EVERY_STATE = -5, IMPOSS_STATE, UNDEF_STATE};    /* special values for state_bn */

enum {FIRST_CASE = -15, NEXT_CASE, NO_MORE_CASES};     /* special values for caseposn_bn */

enum {ENTROPY_SENSV = 0x02, REAL_SENSV = 0x04, VARIANCE_SENSV = 0x100, VARIANCE_OF_REAL_SENSV =
                                0x104};   /* for NewSensvToFinding_bn */


#ifndef __NETICA_CPP_H

typedef struct environ_ins   environ_ns;
typedef struct report_ins    report_ns;
typedef struct stream_ins    stream_ns;
typedef struct net_ibn       net_bn;
typedef struct node_ibn      node_bn;
typedef struct nodelist_ibn  nodelist_bn;
typedef struct caseset_ics   caseset_cs;
typedef struct learner_ibn   learner_bn;
typedef struct test_ibn      tester_bn;
typedef struct sensv_ibn     sensv_bn;
typedef struct setting_ibn   setting_bn;
typedef struct dbmgr_ics     dbmgr_cs;

#endif /* __NETICA_CPP_H */

IMPORT (double) GetUndefDbl_ns();              /* use UNDEF_DBL   in your software */
IMPORT (double) GetInfinityDbl_ns();           /* use INFINITY_ns in your software */


typedef int     state_bn;
typedef float   prob_bn;
typedef float   util_bn;
typedef double  level_bn;

typedef int     color_ns;        /* most significant byte(s) is 0, and last 3 bytes are red,
                                    green, blue */
typedef long    caseposn_bn;
typedef unsigned char bool_ns;


#ifndef FALSE
enum {FALSE=0, TRUE};
#endif

#ifndef NULL
#define NULL (void*)0
#endif


#define  MESG_LEN_ns  600
#define  NAME_MAX_ns   30
```

```
typedef enum {COUNTING_LEARNING=1, EM_LEARNING=3, GRADIENT_DESCENT_LEARNING} learn_method_bn;

enum {NEGATIVE_FINDING = -7, LIKELIHOOD_FINDING, NO_FINDING = -3};   /* for GetNodeFinding_bn */

enum {NO_VISUAL_INFO=0, NO_WINDOW=0x10, MINIMIZED_WINDOW=0x30, REGULAR_WINDOW=0x70};   /* for
                                ReadNet_bn */

enum {BELIEF_UPDATE = 0x100}; /* for SetNetAutoUpdate_bn */

enum {LAST_ENTRY = -10};

enum {QUERY_ns = -1};


IMPORT (environ_ns*) NewNeticaEnviron_ns (const char* license, environ_ns* env, const char* locn);
IMPORT (int) InitNetica2_bn (environ_ns* env, char* mesg);
IMPORT (int) CloseNetica_bn (environ_ns* env, char* mesg);
IMPORT (int) GetNeticaVersion_bn (const environ_ns* env, const char** version);
IMPORT (checking_ns) ArgumentChecking_ns (checking_ns setting, environ_ns* env);
IMPORT (const char*) SetLanguage_ns (const char* language, environ_ns* env);
IMPORT (double) LimitMemoryUsage_ns (double max_mem, environ_ns* env);
IMPORT (const char*) ExecuteScript_ns (environ_ns* env, const char* language, const char* script);
IMPORT (void) SetEnvironUserData_ns (environ_ns* env, int kind, void* data);
IMPORT (void*) GetEnvironUserData_ns (environ_ns* env, int kind);

IMPORT (report_ns*) GetError_ns (environ_ns* env, errseverity_ns severity, const report_ns* after);
IMPORT (int) ErrorNumber_ns (const report_ns* error);
IMPORT (const char*) ErrorMessage_ns (const report_ns* error);
IMPORT (errseverity_ns) ErrorSeverity_ns (const report_ns* error);
IMPORT (bool_ns) ErrorCategory_ns (errcond_ns cond, const report_ns* error);
IMPORT (void) ClearError_ns (report_ns* error);
IMPORT (void) ClearErrors_ns (environ_ns* env, errseverity_ns severity);
IMPORT (report_ns*) NewError_ns (environ_ns* env, int number, errseverity_ns severity, const char*
                                mesg);
IMPORT (bool_ns) TestFaultRecovery_ns (environ_ns* env, int test_num);

IMPORT (void) GetAppWindowPosition_ns (environ_ns* env, int* left, int* top, int* width, int*
                                height, int* status);
IMPORT (void) SetAppWindowPosition_ns (environ_ns* env, int left, int top, int width, int height,
                                int status);
IMPORT (int) UserAllowed_ns (int setting, environ_ns* env);
IMPORT (void) PrintToMessagesWindow_ns (environ_ns* env, char* mesg);

IMPORT (stream_ns*) NewFileStream_ns (const char* filename, environ_ns* env, const char* access);
IMPORT (stream_ns*) NewMemoryStream_ns (const char* name, environ_ns* env, const char* access);
IMPORT (void) DeleteStream_ns (stream_ns* file);
IMPORT (void) SetStreamPassword_ns (stream_ns* stream, const char* password);
IMPORT (void) SetStreamContents_ns (stream_ns* stream, const char* buffer, long length, bool_ns
                                copy);
IMPORT (const char*) GetStreamContents_ns (stream_ns* stream, long* length);
IMPORT (void) WriteNet_bn (const net_bn* net, stream_ns* file);
IMPORT (net_bn*) ReadNet_bn (stream_ns* file, int visual);
IMPORT (caseposn_bn) WriteNetFindings_bn (const nodelist_bn* nodes, stream_ns* file, long ID_num,
                                double freq);
IMPORT (void) ReadNetFindings_bn (caseposn_bn* case_posn, stream_ns* file, const nodelist_bn*
                                nodes, long* ID_num, double* freq);
IMPORT (int) SetCaseFileDelimChar_ns (int newchar, environ_ns* env);
IMPORT (int) SetMissingDataChar_ns (int newchar, environ_ns* env);

IMPORT (net_bn*) NewNet_bn (const char* name, environ_ns* env);
IMPORT (net_bn*) CopyNet_bn (const net_bn* net, const char* new_name, environ_ns* new_env, const
                                char* control);
```

```
IMPORT (void) DeleteNet_bn (net_bn* net);
IMPORT (net_bn*) GetNthNet_bn (int nth, environ_ns* env);
IMPORT (node_bn*) NewNode_bn (const char* name, int num_states, net_bn* net);
IMPORT (nodelist_bn*) CopyNodes_bn (const nodelist_bn* nodes, net_bn* new_net, const char*
                                    control);
IMPORT (void) DeleteNode_bn (node_bn* node);
IMPORT (int) AddLink_bn (node_bn* parent, node_bn* child);
IMPORT (void) DeleteLink_bn (int link_index, node_bn* child);
IMPORT (void) SwitchNodeParent_bn (int link_index, node_bn* node, node_bn* new_parent);
IMPORT (void) GetRelatedNodes_bn (nodelist_bn* related_nodes, const char* relation, const node_bn*
                                    node);
IMPORT (void) GetRelatedNodesMult_bn (nodelist_bn* related_nodes, const char* relation, const
                                    nodelist_bn* nodes);
IMPORT (bool_ns) IsNodeRelated_bn (const node_bn* related_node, const char* relation, const
                                    node_bn* node);

IMPORT (void) SetNetName_bn (net_bn* net, const char* name);
IMPORT (void) SetNetTitle_bn (net_bn* net, const char* title);
IMPORT (void) SetNetComment_bn (net_bn* net, const char* comment);
IMPORT (void) SetNetElimOrder_bn (net_bn* net, const nodelist_bn* elim_order);
IMPORT (int) SetNetAutoUpdate_bn (net_bn* net, int auto_update);
IMPORT (void) SetNetUserField_bn (net_bn* net, const char* name, const void* data, int length, int
                                    kind);
IMPORT (void) SetNetUserData_bn (net_bn* net, int kind, void* data);
IMPORT (void) AddNetListener_bn (net_bn* net, int callback (const net_bn* net, eventtype_ns what,
                                    void* object, void* info), void* object, int filter);

IMPORT (void) SetNodeName_bn (node_bn* node, const char* name);
IMPORT (void) SetNodeTitle_bn (node_bn* node, const char* title);
IMPORT (void) SetNodeComment_bn (node_bn* node, const char* comment);
IMPORT (void) SetNodeLevels_bn (node_bn* node, int num_states, const level_bn* levels);
IMPORT (void) SetNodeKind_bn (node_bn* node, nodekind_bn kind);
IMPORT (void) SetNodeStateName_bn (node_bn* node, state_bn state, const char* state_name);
IMPORT (void) SetNodeStateNames_bn (node_bn* node, const char* state_names);
IMPORT (void) SetNodeStateTitle_bn (node_bn* node, state_bn state, const char* state_title);
IMPORT (void) SetNodeStateComment_bn (node_bn* node, state_bn state, const char* state_comment);
IMPORT (void) SetNodeInputName_bn (node_bn* node, int link_index, const char* link_name);
IMPORT (void) SetNodeEquation_bn (node_bn* node, const char* eqn);
IMPORT (void) SetNodeFuncState_bn (node_bn* node, const state_bn* parent_states, state_bn st);
IMPORT (void) SetNodeFuncReal_bn (node_bn* node, const state_bn* parent_states, double val);
IMPORT (void) SetNodeProbs_bn (node_bn* node, const state_bn* parent_states, const prob_bn* probs);
IMPORT (void) SetNodeExperience_bn (node_bn* node, const state_bn* parent_states, double
                                    experience);
IMPORT (void) DeleteNodeTables_bn (node_bn* node);
IMPORT (void) SetNodeUserField_bn (node_bn* node, const char* name, const void* data, int length,
                                    int kind);
IMPORT (void) SetNodeUserData_bn (node_bn* node, int kind, void* data);
IMPORT (void) AddNodeListener_bn (node_bn* node, int callback (const node_bn* node, eventtype_ns
                                    what, void* object, void* info), void* object, int filter);
IMPORT (void) SetNodeVisPosition_bn (node_bn* node, void* vis, double x, double y);
IMPORT (void) SetNodeVisStyle_bn (node_bn* node, void* vis, const char* style);

IMPORT (const char*) GetNetName_bn (const net_bn* net);
IMPORT (const char*) GetNetTitle_bn (const net_bn* net);
IMPORT (const char*) GetNetComment_bn (const net_bn* net);
IMPORT (const nodelist_bn*) GetNetNodes_bn (const net_bn* net);
IMPORT (node_bn*) GetNodeNamed_bn (const char* name, const net_bn* net);
IMPORT (const char*) GetNetFileName_bn (const net_bn* net);
IMPORT (const nodelist_bn*) GetNetElimOrder_bn (const net_bn* net);
IMPORT (int) GetNetAutoUpdate_bn (const net_bn* net);
IMPORT (const char*) GetNetUserField_bn (const net_bn* net, const char* name, int* length, int
                                    kind);
```

```
IMPORT (void) GetNetNthUserField_bn (const net_bn* net, int index, const char** name, const char**
                                value, int* length, int kind);
IMPORT (void*) GetNetUserData_bn (const net_bn* net, int kind);

IMPORT (net_bn*) GetNodeNet_bn (const node_bn* node);
IMPORT (const char*) GetNodeName_bn (const node_bn* node);
IMPORT (const char*) GetNodeTitle_bn (const node_bn* node);
IMPORT (const char*) GetNodeComment_bn (const node_bn* node);
IMPORT (nodetype_bn) GetNodeType_bn (const node_bn* node);
IMPORT (nodekind_bn) GetNodeKind_bn (const node_bn* node);
IMPORT (int) GetNodeNumberStates_bn (const node_bn* node);
IMPORT (const level_bn*) GetNodeLevels_bn (const node_bn* node);
IMPORT (const char*) GetNodeStateName_bn (const node_bn* node, state_bn state);
IMPORT (const char*) GetNodeStateTitle_bn (const node_bn* node, state_bn state);
IMPORT (const char*) GetNodeStateComment_bn (const node_bn* node, state_bn state);
IMPORT (state_bn) GetStateNamed_bn (const char* name, const node_bn* node);
IMPORT (const nodelist_bn*) GetNodeParents_bn (const node_bn* node);
IMPORT (const nodelist_bn*) GetNodeChildren_bn (const node_bn* node);
IMPORT (const char*) GetNodeInputName_bn (const node_bn* node, int link_index);
IMPORT (int) GetInputNamed_bn (const char* name, const node_bn* node);
IMPORT (const char*) GetNodeEquation_bn (const node_bn* node);
IMPORT (state_bn) GetNodeFuncState_bn (const node_bn* node, const state_bn* parent_states);
IMPORT (double) GetNodeFuncReal_bn (const node_bn* node, const state_bn* parent_states);
IMPORT (const prob_bn*) GetNodeProbs_bn (const node_bn* node, const state_bn* parent_states);
IMPORT (double) GetNodeExperience_bn (const node_bn* node, const state_bn* parent_states);
IMPORT (bool_ns) HasNodeTable_bn (const node_bn* node, bool_ns* complete);
IMPORT (bool_ns) IsNodeDeterministic_bn (const node_bn* node);
IMPORT (const char*) GetNodeUserField_bn (const node_bn* node, const char* name, int* length, int
                                kind);
IMPORT (void) GetNodeNthUserField_bn (const node_bn* node, int index, const char** name, const
                                char** value, int* length, int kind);
IMPORT (void*) GetNodeUserData_bn (const node_bn* node, int kind);
IMPORT (void) GetNodeVisPosition_bn (const node_bn* node, void* vis, double* x, double* y);
IMPORT (const char*) GetNodeVisStyle_bn (const node_bn* node, void* vis);

IMPORT (nodelist_bn*) NewNodeList2_bn (int length, const net_bn* net);
IMPORT (void) DeleteNodeList_bn (nodelist_bn* nodes);
IMPORT (void) ClearNodeList_bn (nodelist_bn* nodes);
IMPORT (void) AddNodeToList_bn (node_bn* node, nodelist_bn* nodes, int index);
IMPORT (node_bn*) RemoveNthNode_bn (nodelist_bn* nodes, int index);
IMPORT (int) LengthNodeList_bn (const nodelist_bn* nodes);
IMPORT (node_bn*) NthNode_bn (const nodelist_bn* nodes, int index);
IMPORT (void) SetNthNode_bn (nodelist_bn* nodes, int index, node_bn* node);
IMPORT (int) IndexOfNodeInList_bn (const node_bn* node, const nodelist_bn* nodes, int start_index);
IMPORT (nodelist_bn*) DupNodeList_bn (const nodelist_bn* nodes);

IMPORT (void) MapStateList_bn (const state_bn* src_states,  const nodelist_bn* src_nodes,
                               state_bn* dest_states, const nodelist_bn* dest_nodes);

IMPORT (void) ReviseCPTsByFindings_bn (const nodelist_bn* nodes, int updating, double degree);
IMPORT (void) ReviseCPTsByCaseFile_bn (stream_ns* file, const nodelist_bn* nodes, int updating,
                                double degree);
IMPORT (void) FadeCPTable_bn (node_bn* node, double degree);

IMPORT (void) AddNodeStates_bn (node_bn* node, state_bn first_state, const char* state_names, int
                                num_states, double cpt_fill);
IMPORT (void) RemoveNodeState_bn (node_bn* node, state_bn state);
IMPORT (void) ReorderNodeStates_bn (node_bn* node, const state_bn* new_order);
IMPORT (void) EquationToTable_bn (node_bn* node, int num_samples, bool_ns samp_unc, bool_ns
                                add_exist);
IMPORT (void) ReverseLink_bn (node_bn* parent, node_bn* child);
```

```
IMPORT (void) AbsorbNodes_bn (nodelist_bn* nodes);

IMPORT (void) EnterFinding_bn (node_bn* node, state_bn state);
IMPORT (void) EnterFindingNot_bn (node_bn* node, state_bn state);
IMPORT (void) EnterNodeValue_bn (node_bn* node, double value);
IMPORT (void) EnterNodeLikelihood_bn (node_bn* node, const prob_bn* likelihood);
IMPORT (void) EnterIntervalFinding_bn (node_bn* node, double low, double high);
IMPORT (void) EnterGaussianFinding_bn (node_bn* node, double mean, double std_dev);
IMPORT (state_bn) GetNodeFinding_bn (const node_bn* node);
IMPORT (double) GetNodeValueEntered_bn (const node_bn* node);
IMPORT (const prob_bn*) GetNodeLikelihood_bn (const node_bn* node);
IMPORT (void) RetractNodeFindings_bn (node_bn* node);
IMPORT (void) RetractNetFindings_bn (net_bn* net);

IMPORT (state_bn) CalcNodeState_bn (node_bn* node);
IMPORT (double) CalcNodeValue_bn (node_bn* node);

IMPORT (void) CompileNet_bn (net_bn* net);
IMPORT (void) UncompileNet_bn (net_bn* net);
IMPORT (double) SizeCompiledNet_bn (net_bn* net, int method);
IMPORT (const char*) ReportJunctionTree_bn (net_bn* net);
IMPORT (bool_ns) IsBeliefUpdated_bn (const node_bn* node);
IMPORT (const prob_bn*) GetNodeBeliefs_bn (node_bn* node);
IMPORT (double) GetNodeExpectedValue_bn (node_bn* node, double* std_dev, double* x3, double* x4);
IMPORT (const util_bn*) GetNodeExpectedUtils_bn (node_bn* node);
IMPORT (double) FindingsProbability_bn (net_bn* net);
IMPORT (util_bn) GetNetExpectedUtility_bn (net_bn* net);
IMPORT (double) JointProbability_bn (const nodelist_bn* nodes, const state_bn* states);
IMPORT (void) MostProbableConfig_bn (const nodelist_bn* nodes, state_bn* config, int nth);

IMPORT (int) GenerateRandomCase_bn (const nodelist_bn* nodes, int method, double num, void* gen);

IMPORT (void) AddNodeToNodeset_bn (node_bn* node, const char* nodeset);
IMPORT (void) RemoveNodeFromNodeset_bn (node_bn* node, const char* nodeset);
IMPORT (bool_ns) IsNodeInNodeset_bn (const node_bn* node, const char* nodeset);
IMPORT (const char*) GetAllNodesets_bn (const net_bn* net, bool_ns include_system, void* vis);
IMPORT (color_ns) SetNodesetColor_bn (const char* nodeset, color_ns color, net_bn* net, void* vis);
IMPORT (void) ReorderNodesets_bn (net_bn* net, const char* nodeset_order, void* vis);

IMPORT (sensv_bn*) NewSensvToFinding_bn (const node_bn* Qnode, const nodelist_bn* Vnodes, int
                                what_find);
IMPORT (void) DeleteSensvToFinding_bn (sensv_bn* s);
IMPORT (double) GetMutualInfo_bn (sensv_bn* s, const node_bn* Vnode);
IMPORT (double) GetVarianceOfReal_bn (sensv_bn* s, const node_bn* Vnode);

IMPORT (caseset_cs*) NewCaseset_cs (const char* name, environ_ns* env);
IMPORT (void) DeleteCaseset_cs (caseset_cs* cases);
IMPORT (void) AddFileToCaseset_cs (caseset_cs* cases, const stream_ns* file, double degree, const
                                char* control);
IMPORT (void) WriteCaseset_cs (const caseset_cs* cases, stream_ns* file, const char* control);

IMPORT (dbmgr_cs*) NewDBManager_cs (const char* connect_str, const char* control, environ_ns* env);
IMPORT (void) DeleteDBManager_cs (dbmgr_cs* dbmgr);
IMPORT (void) ExecuteDBSql_cs (dbmgr_cs* dbmgr, const char* sql_cmnd, const char* control);
IMPORT (void) InsertFindingsIntoDB_bn (dbmgr_cs* dbmgr, const nodelist_bn* nodes, const char*
                                column_names, const char* tables, const char* control);
IMPORT (void) AddDBCasesToCaseset_cs (caseset_cs* cases, dbmgr_cs* dbmgr, double degree, const
                                nodelist_bn* nodes, const char* column_names, const char* tables,
                                const char* condition, const char* control);
IMPORT (void) AddNodesFromDB_bn (dbmgr_cs* dbmgr, net_bn* net, const char* column_names, const
                                char* tables, const char* condition, const char* control);
```

```
IMPORT (learner_bn*) NewLearner_bn (learn_method_bn method, const char* info, environ_ns* env);
IMPORT (void) DeleteLearner_bn (learner_bn* algo);
IMPORT (int) SetLearnerMaxIters_bn (learner_bn* algo, int max_iters);
IMPORT (double) SetLearnerMaxTol_bn (learner_bn* algo, double log_likeli_tol);
IMPORT (void) LearnCPTs_bn (learner_bn* algo, const nodelist_bn* nodes, const caseset_cs* cases,
                            double degree);

IMPORT (tester_bn*) NewNetTester_bn (const nodelist_bn* test_nodes, const nodelist_bn*
                            unobsv_nodes, int tests);
IMPORT (void) DeleteNetTester_bn (tester_bn* test);
IMPORT (void) TestWithCaseset_bn (tester_bn* test, const caseset_cs* cases);
IMPORT (double) GetTestConfusion_bn (const tester_bn* test, const node_bn* node, state_bn
                            predicted, state_bn actual);
IMPORT (double) GetTestErrorRate_bn (const tester_bn* test, const node_bn* node);
IMPORT (double) GetTestLogLoss_bn (const tester_bn* test, const node_bn* node);
IMPORT (double) GetTestQuadraticLoss_bn (const tester_bn* test, const node_bn* node);

IMPORT (int) UndoNetLastOper_bn (net_bn* net, double to_when);
IMPORT (int) RedoNetOper_bn (net_bn* net, double to_when);

/*--------------*/

IMPORT (setting_bn*) NewSetting_bn (const nodelist_bn* nodes, bool_ns load);
IMPORT (void) DeleteSetting_bn (setting_bn* cas);
IMPORT (void) SetSettingState_bn (setting_bn* cas, const node_bn* node, state_bn state);
IMPORT (state_bn) GetSettingState_bn (const setting_bn* cas, const node_bn* bnd);
IMPORT (void) ZeroSetting_bn (setting_bn* cas);
IMPORT (bool_ns) NextSetting_bn (setting_bn* cas);
IMPORT (void) MostProbableSetting_bn (setting_bn* cas, int nth);

IMPORT (double) NthProb_bn (const prob_bn* probs, state_bn state);
IMPORT (double) NthLevel_bn (const level_bn* levels, state_bn state);
IMPORT (int) GetChars_ns (const char* str, int index, unsigned short* dest, int num);
IMPORT (int) NthChar_ns (const char* str, int index);
IMPORT (void) SetNthState_bn (state_bn* states, int index, state_bn state);

IMPORT (void) OptimizeDecisions_bn (const nodelist_bn* nodes);


/* Shorthand Notation */
#define NodeNamed_bn   GetNodeNamed_bn
#define StateNamed_bn  GetStateNamed_bn
#define InputNamed_bn  GetInputNamed_bn


/* These definitions are just for compatibility with old versions: */

#ifndef NO_DEPRECATED_NETICA_FUNCS

enum {WILDCARD_STATE = EVERY_STATE};
#define ASSUME_NODE       CONSTANT_NODE
#define SetLinkName_bn    SetNodeInputName_bn
#define GetLinkName_bn    GetNodeInputName_bn
#define LinkNamed_bn      GetInputNamed_bn
#define ReOrderStates_bn  MapStateList_bn
#define GetNodeValue_bn   GetNodeValueEntered_bn
#define SetNodeCenter_bn  SetNodeVisPosition_bn
#define FreeNet_bn        DeleteNet_bn
#define FreeNodeList_bn   DeleteNodeList_bn
#define ReportError_ns    NewError_ns
```

```
#define ErrorDanger_ns     ErrorSeverity_ns
#define errdanger_ns       errseverity_ns
#define GetJointProb_bn    JointProbability_bn
#define FadeProbs_bn       FadeCPTable_bn
#define RetractAllFindings_bn RetractNetFindings_bn
#define DeleteNodeRelation_bn DeleteNodeTables_bn
#define CaseProbability_bn     FindingsProbability_bn
#define GetNodeCalcState_bn    CalcNodeState_bn
#define GetNodeCalcValue_bn    CalcNodeValue_bn
#define CaseRevisesProbs_bn       ReviseCPTsByFindings_bn
#define CaseFileRevisesProbs_bn   ReviseCPTsByCaseFile_bn
extern double BaseExperience_bn;
#define NewStreamFile_ns  NewFileStream_ns
#define MaxMemoryUsage_ns LimitMemoryUsage_ns
#define ReadCase_bn     ReadNetFindings_bn
#define WriteCase_bn    WriteNetFindings_bn
#define RandomCase_bn(nodes, method, num)   GenerateRandomCase_bn (nodes, method, num, NULL)
#define FileNamed_ns(filename, env)  NewFileStream_ns (filename, env, 0)
#define GetNodeLevel_bn(node, state)  (GetNodeLevels_bn (node) ? GetNodeLevels_bn (node) [state] :
                                  UNDEF_DBL)
#define HasRelation_bn(node)    HasNodeTable_bn (node, 0)
#define GetNodeDiscrete_bn(node)  ((GetNodeType_bn (node) == DISCRETE_TYPE) ? TRUE : FALSE)
#define SetNodeFuncValue_bn  SetNodeFuncValue1_bn
static void SetNodeFuncValue1_bn (node_bn* node, const state_bn* parent_states, double func_value){
    if (GetNodeType_bn (node) == DISCRETE_TYPE)
         SetNodeFuncState_bn (node, parent_states, (int)func_value);
    else SetNodeFuncReal_bn (node, parent_states, func_value);
    }
#define GetNodeFuncValue_bn(node, parent_states)  ((GetNodeType_bn (node) == DISCRETE_TYPE) ?
                                  GetNodeFuncState_bn (node, parent_states) : GetNodeFuncReal_bn
                                  (node, parent_states))
#define NewNeticaEnviron_bn(license)  NewNeticaEnviron_ns (license, 0, 0)
#define InitNetica_bn(envp, mesg)   InitNetica2_bn (*(envp), mesg)
static void GetNodeCenter_bn (const node_bn* node, void* vis, int* x, int* y){
    double xd, yd;
    GetNodeVisPosition_bn (node, vis, &xd, &yd);
    if (x) *x = (int)xd;
    if (y) *y = (int)yd;
    }
#define MutualInfo_bn      GetMutualInfo_bn
#define VarianceOfReal_bn  GetVarianceOfReal_bn
#define DuplicateNodes_bn(nodes, new_net)  CopyNodes_bn (nodes, new_net, NULL)
IMPORT (nodelist_bn*) NewNodeList_bn (int length, environ_ns* env);

#endif  /* !NO_DEPRECATED_NETICA_FUNCS */

/* End compatibility definitions */

#ifdef __cplusplus
}
#endif

#endif  /* __NETICA_C_H */
```

13 Functions by Category

System

NewNeticaEnviron_ns	Creates a new environment to pass to InitNetica2_bn
InitNetica2_bn	Initializes the Netica system
CloseNetica_bn	Signals an end to using Netica system, and frees all possible resources (e.g. memory, close any open files)
ArgumentChecking_ns	Adjusts the amount that Netica functions check their arguments
GetNeticaVersion_bn	Gets the software version of Netica currently running
LimitMemoryUsage_ns	Adjusts the amount of memory that Netica can allocate for tables
SetCaseFileDelimChar_ns	The symbol to separate data fields in case files created by Netica
SetMissingDataChar_ns	The symbol indicating missing data in case files created by Netica

Error Handling

GetError_ns	Gets the next error report of a given severity or worse
ErrorMessage_ns	Returns an error message for the given error report
ErrorCategory_ns	Indicates the nature of the error (out of memory, aborted, etc.)
ErrorSeverity_ns	Returns the severity level of the given error report
ErrorNumber_ns	Returns the error number of the given error report
ClearError_ns	Removes the given error report from the system
ClearErrors_ns	Clears away all error reports of up to a given severity
NewError_ns	Make your own error report using Netica
ArgumentChecking_ns	Adjusts the amount that Netica functions check their arguments

File Operations

NewFileStream_ns	Creates a stream for the file with the given name
NewMemoryStream_ns	Creates a stream for reading and writing to buffers in memory
DeleteStream_ns	Closes files, frees resources and deletes either type of stream
SetStreamContents_ns	For memory streams, sets the contents of the buffer
GetStreamContents_ns	For memory streams, gets the contents of the buffer
SetStreamPassword_ns	Sets a password to read or write encrypted files
SetCaseFileDelimChar_ns	The symbol to separate data fields in case files created by Netica

SetMissingDataChar_ns	The symbol indicating missing data in case files created by Netica
WriteNet_bn	Saves a net to a file
ReadNet_bn	Reads a net from a file
WriteNetFindings_bn	Saves a net's current set of findings to a file
ReadNetFindings_bn	Reads findings from a file, and enters into a net
WriteCaseset_cs	Writes all the cases to a file in CSV or UVF format
AddFileToCaseset_cs	Makes the case-set object consist of the cases located in the file
ReviseCPTsByCaseFile_bn	Reads a file of cases to revise probabilities
GetNetFileName_bn	Name of file (with full path) that net was last written to or read from

Findings (Evidence)

EnterFinding_bn	Enters a discrete finding that a node is in a given state
EnterFindingNot_bn	Enters a discrete finding that a node is not in a given state
EnterNodeValue_bn	Enters a real number finding for a continuous node
EnterNodeLikelihood_bn	Enters a likelihood finding for a node
EnterGaussianFinding_bn	Enters a finding given by a Gaussian (normal) distribution
EnterIntervalFinding_bn	Enters a finding uniform over an interval, zero outside
GetNodeFinding_bn	Returns the finding for a node, if there is one
GetNodeLikelihood_bn	Returns the accumulated findings for a node, as a likelihood vector
GetNodeValueEntered_bn	Returns the real number finding entered for a continuous node
RetractNodeFindings_bn	Retracts all findings for a single node
RetractNetFindings_bn	Retracts all findings (i.e. the current case) from a net
FindingsProbability_bn	Returns the joint probability of the findings entered so far

Compiling

CompileNet_bn	Compiles a net for fast belief updating
UncompileNet_bn	Releases the resources (e.g., memory) used by a compiled net
SizeCompiledNet_bn	The size and speed of the compiled net (i.e. of the junction tree)
ReportJunctionTree_bn	Returns a string describing the internal compiled junction tree
SetNetElimOrder_bn	Sets the node order used to guide compilation
GetNetElimOrder_bn	Retrieves the node order used to guide compilation
SetNetAutoUpdate_bn	Automatically propagate beliefs when findings are entered
GetNetAutoUpdate_bn	Returns whether net automatically propagate beliefs
EquationToTable_bn	Builds the CPT for a node based on the equation given to it

Belief Updating and Inference

GetNodeBeliefs_bn	Returns a node's current beliefs, doing belief updating if necessary
GetNodeExpectedValue_bn	Expected value (and std dev) of a continuous or numeric-valued node
GetNodeExpectedUtils_bn	Returns the expected utility of each choice in a decision node
IsBeliefUpdated_bn	Returns whether a node's beliefs have already been calculated to account for current findings
SetNetAutoUpdate_bn	Automatically propagate beliefs when findings are entered
GetNetAutoUpdate_bn	Returns whether net automatically propagate beliefs
JointProbability_bn	Returns a specified joint probability, given the findings entered
FindingsProbability_bn	Returns the joint probability of the findings entered so far

MostProbableConfig_bn	Finds the state for each node which results in the most probable explanation (MPE)
GenerateRandomCase_bn	Creates a case sampled from the net, given the current findings
AbsorbNodes_bn	Removes the given nodes while maintaining the joint distribution of the remaining nodes
GetMutualInfo_bn	Measures the mutual information between two nodes
GetVarianceOfReal_bn	Measures how much a finding at one node is expected to reduce the variance of another node
CalcNodeState_bn	Returns the state of a node calculated from its neighbors, if that can be done deterministically
CalcNodeValue_bn	Returns the numeric value of a node calculated from its neighbors, if that can be done deterministically

Learning From Data

ReviseCPTsByFindings_bn	Uses the current case to revise each node's probabilities
ReviseCPTsByCaseFile_bn	Reads a file of cases to revise probabilities
NewLearner_bn	Creates a new object for use in learning CPTs from case data
DeleteLearner_bn	Deletes a learning object (learner)
LearnCPTs_bn	Performs learning of CPT tables from data
SetLearnerMaxIters_bn	Sets the maximum number of learning-step iterations (i.e., complete passes through the data) which will be done when the learner is used
SetLearnerMaxTol_bn	The minimum change in data log likelihood between consecutive passes through the data, as a termination condition
FadeCPTable_bn	Adjusts a node's probabilities for a changing world
GetNodeProbs_bn	Returns the results of learning
GetNodeExperience_bn	Determines how much experience was involved in the learning
SetNodeProbs_bn	Directly sets the probabilities (or starts them off)
SetNodeExperience_bn	Manually sets the amount of experience (or starts it off)

Decision Nets

GetNodeExpectedUtils_bn	Returns the expected utility of each choice in a decision node
SetNodeKind_bn	Used to create decision nodes and utility nodes

Node Lists

NewNodeList2_bn	Creates a new (empty) list of nodes
AddNodeToList_bn	Inserts a node at the given position of a list, making it one longer
RemoveNthNode_bn	Removes the node at the given index of a list, making it one shorter
SetNthNode_bn	Sets the Nth node of a list to a given node without changing length
NthNode_bn	Returns the Nth node of a list (the first node is numbered 0)
IndexOfNodeInList_bn	Returns the position (index) of a **node** in a list, or -1 if it is not present
LengthNodeList_bn	Returns the number of nodes in a list
DupNodeList_bn	Duplicates a list of nodes
ClearNodeList_bn	Empties a node list without releasing the memory it uses
DeleteNodeList_bn	Frees the memory used by a list of nodes
GetNodeNamed_bn	Returns the node with the given name, from a given net
GetNetNodes_bn	Returns a list of all the nodes in a net

GetNodeParents_bn	Returns a list of the parents of a node
GetNodeChildren_bn	Returns a list of the children of a node
GetRelatedNodes_bn	Finds all the nodes that bear a given relationship (such as D-connected, Markov blanket, ancestors, children, etc.) with a given node
GetRelatedNodesMult_bn	Finds the nodes that bear a given relationship with a given set of nodes
MapStateList_bn	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
WriteNetFindings_bn	Saves a net's current set of findings to a file
ReadNetFindings_bn	Reads findings from a file, and enters into a net
RetractNetFindings_bn	Retracts all findings (i.e. the current case) from a net
FindingsProbability_bn	Returns the joint probability of the findings entered so far
ReviseCPTsByFindings_bn	The current case is used to revise each node's probabilities
ReviseCPTsByCaseFile_bn	Reads a file of cases to revise probabilities
LearnCPTs_bn	Learn CPTs from cases, with choice of algorithm
GenerateRandomCase_bn	Generates a random case in a net, according to the net's distribution
NewCaseset_cs	Creates a new case-set object, initially with no cases
DeleteCaseset_cs	Deletes and frees all resources used by a case-set object
AddDBCasesToCaseset_cs	Searches the given database, adding cases to a case-set object
AddFileToCaseset_cs	Makes the case-set object consist of the cases located in the file
WriteCaseset_cs	Writes all the cases in the given case-set to a file stream
TestWithCaseset_bn	Performance tests a bayes net with a set of cases
MapStateList_bn	Change the order of a list of states to match a given node list

Sensitivity to Findings (Utility-Free Value of Information)

NewSensvToFinding_bn	Creates an object to measure sensitivity
DeleteSensvToFinding_bn	Deletes the sensitivity measuring object
GetVarianceOfReal_bn	Measure the expected reduction in variance due to a finding
GetMutualInfo_bn	Measure the mutual information (entropy reduction)

Performance Testing a Net

NewNetTester_bn	Creates a new tester object, for given tests on given nodes
DeleteNetTester_bn	Deletes a tester object
TestWithCaseset_bn	Reads the cases one-by-one, and for each it does inference and grades the Netica net, gathering statistics
GetTestConfusion_bn	Returns a confusion matrix result of the testing
GetTestErrorRate_bn	Returns the error rate result of the testing
GetTestLogLoss_bn	Returns the logarithmic loss result of the testing
GetTestQuadraticLoss_bn	Returns the quadratic loss result of the testing

Node-Sets

AddNodeToNodeset_bn	Adds the given node to the node-set of the given name
RemoveNodeFromNodeset_bn	Removes the given node from the node-set of the given name

IsNodeInNodeset_bn	Returns whether the given node is a member of the given node-set
ReorderNodesets_bn	Re-orders the node-sets as requested, for priority during display
GetAllNodesets_bn	Returns a list of all node-sets defined for this net, in priority order
SetNodesetColor_bn	Sets the color used to display nodes of a given node-set, and returns old

Database Connectivity

NewDBManager_cs	Creates a new database manager object for a given database
DeleteDBManager_cs	Closes connection and deletes a database manager object
InsertFindingsIntoDB_bn	Adds the current findings in the net into the database as a case
AddDBCasesToCaseset_cs	Adds the cases (or a subset) in the database to a case-set object
ExecuteDBSql_cs	Executes arbitrary SQL commands on the database
AddNodesFromDB_bn	Adds to the given net nodes that match the variables in the database

High-Level Net Modification

ReverseLink_bn	Reverses a single link while maintaining joint probability
AbsorbNodes_bn	Absorbs out (sum or max) some net nodes
EquationToTable_bn	Builds a node's CPT or function table based on its equation
SwitchNodeParent_bn	Switches a link that comes from some node to come from a different node, without changing the child node or its tables
CopyNodes_bn	Duplicates each node in a list, putting them in the same or a new net
CopyNet_bn	Duplicate a whole net (with options to skip tables, etc.)
UndoNetLastOper_bn	Undoes the last operation done to a net
RedoNetOper_bn	Call this to redo an operation that was undone

Low-Level Net Modification

See also "Equations", "Tables", "Visual Display" , "Node-Sets" and "User Data Fields"

NewNet_bn	Creates a new empty net
DeleteNet_bn	Frees all memory used by a net and all its substructures
SetNetName_bn	Changes the name of the net
SetNetAutoUpdate_bn	Changes whether a node does belief updating immediately
SetNetElimOrder_bn	Provides the elimination order to be used for the next compilation
SetNetTitle_bn	Sets the string used to title a net
SetNetComment_bn	Attaches a comment string to the net

NewNode_bn	Creates a new node for a given net
DeleteNode_bn	Removes a node from its net, and frees the memory it required
CopyNodes_bn	Duplicates each node in a list, putting them in same or new net
AddNodesFromDB_bn	Adds to the given net nodes that match the variables in the database
SetNodeName_bn	Changes the name of a node
SetNodeTitle_bn	Sets the string used to title a node
SetNodeComment_bn	Attaches a comment string to the node
SetNodeKind_bn	Changes whether the node is a nature, decision, utility, etc. node
SetNodeStateName_bn	Provides a name for a state of the node
SetNodeStateNames_bn	Name all the states of a node at once with a comma-delimited string
SetNodeStateTitle_bn	Set the title of a state of the node
SetNodeStateComment_bn	Attach a comment to the state of a node

SetNodeLevels_bn	Sets a threshold number for continuous / discrete conversion
SetNodeInputName_bn	Sets the link's name (to be used by the child node in its equation)
AddNodeStates_bn	Insert one or more states into a node's list of states
RemoveNodeState_bn	Remove a state from a node
ReorderNodeStates_bn	Change the order of a node's states
AddLink_bn	Adds a link from one node to another
DeleteLink_bn	Removes a link from one node to another
SwitchNodeParent_bn	Switches a link that comes from some node to come from a different node, without changing the child node or its tables

Retrieving Net Information

See also "Equations", "Tables", "Visual Display" , "Node-Sets" and "User Data Fields"

GetNetName_bn	Returns the name of the net
GetNetAutoUpdate_bn	Returns whether the net does belief updating immediately
GetNetElimOrder_bn	Returns a list of the elimination order used for compiling (triangulation)
GetNetTitle_bn	Returns the string which is the net's title
GetNetComment_bn	Returns the comment associated with the net
GetNetFileName_bn	Name of file (with full path) that net was last written to or read from
GetNetNodes_bn	Returns a list of all the nodes in a net
GetNodeNamed_bn	Returns the node having the given name from the net
GetNthNet_bn	Can be used to return all the nets in the Netica environ, one-by-one

GetNodeNet_bn	Returns the net containing the given node
GetNodeName_bn	Returns the name of the given node
GetNodeType_bn	Returns whether the node is for a discrete or continuous variable
GetNodeKind_bn	Returns whether the node is a nature, decision, utility, etc. node
GetNodeNumberStates_bn	Returns the number of states node can take on
GetNodeStateName_bn	Returns the name of the given state
GetStateNamed_bn	Returns the state number of the state with the given name
GetNodeStateTitle_bn	Returns the title of the given state
GetNodeStateComment_bn	Returns the comment of the given state
GetNodeLevels_bn	Returns a threshold number for continuous / discrete conversion
GetInputNamed_bn	Returns the parent index of the link with the given name
GetNodeParents_bn	Returns a node list of the parents of the node
GetNodeChildren_bn	Returns a node list of the children of the node
GetNodeTitle_bn	Returns the string titling the node
GetNodeComment_bn	Returns a comment string for the node

Equations

SetNodeEquation_bn	Set a node's equation (expressing the node's value or CPT as a function of its parent nodes)
GetNodeEquation_bn	Returns the equation given to a node
EquationToTable_bn	Builds the node's function or CPT table from its equation
SetNodeInputName_bn	Sets the name of a link (to be used by the node's equation instead of the parent node name)
GetNodeInputName_bn	Returns the name associated with a link

CalcNodeState_bn	Calculates, if possible, the state of a node, based on its deterministic equation or table, and findings at its neighbor nodes
CalcNodeValue_bn	Calculates, if possible, the numerical value of a node, based on its deterministic equation or table, and findings at its neighbor nodes

Tables

SetNodeProbs_bn	Sets the conditional probability of the node given its parents values
SetNodeExperience_bn	Attaches an experience level to a conditional probability vector
SetNodeFuncReal_bn	Adds entry(s) to function table of a continuous deterministic node
SetNodeFuncState_bn	Adds entry(s) to function table of a discrete deterministic node
DeleteNodeTables_bn	Removes a node's function, probability, and experience tables
GetNodeProbs_bn	Returns the conditional probabilities of the node given its parents
GetNodeExperience_bn	Returns how much learning is associated with the node
GetNodeFuncReal_bn	Returns the deterministic value of a continuous node
GetNodeFuncState_bn	Returns the deterministic value of a discrete or discretized node
HasNodeTable_bn	Whether the node has a CPT table or function table
IsNodeDeterministic_bn	Discovers if the node is a deterministic function of its parents
MapStateList_bn	Useful for getting states in correct order to access a table
EquationToTable_bn	Build table from equation
LearnCPTs_bn	Performs learning of CPT tables from data
ReviseCPTsByFindings_bn	Modify CPTs by learning from a single case
ReviseCPTsByCaseFile_bn	Modify CPTs by learning from cases
FadeCPTable_bn	Increase uncertainty in CPT table to account for passage of time

Visual Display

SetNodeVisStyle_bn	Sets the style to draw the node in Netica Application
GetNodeVisStyle_bn	Returns the style to draw the node in Netica Application
SetNodeVisPosition_bn	Sets the coordinates of the center of the node in the Netica Application
GetNodeVisPosition_bn	Returns the coordinates of the center of the node in Netica Application

User Data Fields

SetNetUserField_bn	Associates field-by-field info with net, that gets saved to file
GetNetUserField_bn	Retrieves field-by-field info from net by field name
SetNodeUserField_bn	Associates field-by-field info with node, that gets saved to file
GetNodeUserField_bn	Retrieves field-by-field info from node by field name
GetNetNthUserField_bn	Retrieves field-by-field info from net by index
GetNodeNthUserField_bn	Retrieves field-by-field info from node by index
SetNetUserData_bn	Sets the net's "user" field to reference the specified data
GetNetUserData_bn	Returns the net's "user" field (i.e. the value it was set to)
SetNodeUserData_bn	Sets the node's "user" field to reference the specified data
GetNodeUserData_bn	Returns the node's "user" field (i.e. the value it was set to)

14 Function Reference

AbsorbNodes_bn

void AbsorbNodes_bn (nodelist_bn* nodes)

Absorbs all of *nodes* from their net. This removes and deletes (frees) the nodes while maintaining the global relationship (i.e., joint distribution) of the remaining nodes. In the probabilistic literature this is often referred to as "summing out" variables (or "maxing out" when they are decision nodes).

In order to maintain the joint distribution, Netica may have to add links. Absorbing a nature node which has no finding will only add links from the parents of the removed node and its children's parents, to the removed node's children. However, if it has a finding, many links between the ancestors of the removed node may be added (possibly resulting in very large CPT tables leading to slow behavior or an out-of-memory condition). Absorbing nodes with likelihood findings or negative findings is the worst. When a decision node is absorbed, links will be added from its parents to its children. No links are added when a utility node is absorbed. Added links never created a directed cycle, when there wasn't one to begin with.

The order of the nodes in *nodes* doesn't matter. The order in which the absorptions are done will be chosen to minimize intermediate calculations (and if decision nodes are involved, it will be similar to that described in Shachter86).

All of the nodes in *nodes* must be from the same net.

If it is not possible to absorb all of *nodes*, as many as possible will be absorbed, and then an error will be generated explaining why the next node couldn't be absorbed. Reasons it may not be possible to continue are: nodes are missing CPTs, presence of disconnected links, more than one link from a node to another, presence of directed cycles, unacceptable structure between decision and utility nodes, or multiple utility nodes.

WARNING: This function will delete (free) the entire nodelist_bn *nodes* (it's contents would be invalid anyway, since all the nodes in it have been deleted (freed)). You should not call DeleteNodeList_bn on it.

Version:

This function is available in all versions.

See also:

DeleteNode_bn	Removes a node without maintaining joint distribution
LimitMemoryUsage_ns	In case this function is consuming too much memory
NewNodeList2_bn	Make the required list of nodes

Example:

The following function is available in `NeticaEx.c`:

```
// Handy function to absorb a single node
//
```

```
void AbsorbNode (node_bn* node){
    nodelist_bn* nodes = NewNodeList2_bn (1, GetNodeNet_bn (node));
    SetNthNode_bn (nodes, 0, node);
    AbsorbNodes_bn (nodes);
}
```

AddDBCasesToCaseset_cs

```
void AddDBCasesToCaseset_cs (caseset_cs* cases, dbmgr_cs* dbmgr,
                             double degree, const nodelist_bn* nodes,
                             const char* column_names,
                             const char* tables,
                             const char* condition,
                             const char* control)
```

Searches the database attached to *dbmgr* for cases to add to *cases*.

The cases are retrieved from the database by invoking the SQL[1] SELECT statement:

SELECT *column_names* FROM *tables* WHERE *condition*.

degree indicates how each case that is retrieved should be weighted. See ReviseCPTsByFindings_bn for more information about the relative weighting of cases.

nodes represents the nodes whose values will be selected. It must not be NULL, and must contain at least one node.

column_names is a comma-delimited list of database column names. The names in this list must be in the exact same order as their corresponding nodes in *nodes*. If *column_names* is NULL, then for each Node, Netica will use that Node's title (or, if title not defined, then the name) as the corresponding column name. If you are selecting columns from different tables, then you cannot use the NULL option just mentioned, and you must also prefix the column names with the table name followed by a period, as per the standard SQL syntax.

tables is a comma-delimited list of database table names. If the database has only one conventional (non-system) table, then you can submit NULL for this parameter and Netica will find the implied table for you.

condition is the text following the "WHERE" clause. It may be NULL.

Pass NULL for *control*; it is only for future expansion.

Thus, for the SQL command SELECT col1,col2,...,colN FROM table1 WHERE surname="smith",*tables* should be "table1"; *column_names* should be "col1,col2,...,colN"; *nodes* should be a list of nodes in the order node1, node2, ..., nodeN; and *condition* should be "surname=\"smith\"".

If there is a problem with the SQL SELECT command, a Netica error will be generated explaining the nature of the problem.

TEMPORARY LIMITATION: Currently you can only add one file or database retrieval to a caseset.

[1] SQL is a standard query language for accessing databases. To properly use this function, you should have basic familiarity with the SQL SELECT statement.

Version:

Versions 3.15 and later have this function.

See also:

NewDBManager_cs	Creates the dbmgr_cs
NewCaseset_cs	Create an empty caseset_cs
AddFileToCaseset_cs	Add cases from text file instead of database

Example:

```
// Here is an example program to use EM learning to learn Bayes net parametersfrom a database:
    dbmgr_cs *dbmgr = NewDBManager_cs (
         "driver=Microsoft Access Driver (*.mdb); dbq=.\myDB.mdb; UID=dba1;",
         "pooling", env);
```

```
caseset_cs* cases = NewCaseset_cs ("TestDBCases", env);
AddDBCasesToCaseset_cs (cases,
                        dbmgr,
                        1.0,
                        NULL,
                        "Gender, Height, OwnsHouse, NumDogs"
                        "gender, height, \"Owns a house\", \"Number of dogs\"",
                        "'Owns a house' = 'yes'",
                        NULL);
net_bn*  net = NewNet_bn ("TestDB", env);
// ... Put code here to add nodes and links to net ...
const nodelist_bn* nodes = GetNetNodes_bn (net);
learner_bn* learner = NewLearner_bn (EM_LEARNING, NULL, env);
LearnCPTs_bn (learner, nodes, cases, 1.0);
DeleteLearner_bn (learner);
DeleteCaseset_cs (cases);
DeleteDBManager_cs (dbmgr);
```

AddFileToCaseset_cs

`void AddFileToCaseset_cs (caseset_cs* cases, const stream_ns* file,`
`                          double degree, const char* control)`

Indicates that all the cases within *`file`* should be added to the caseset *`cases`*.

`degree` indicates how each case in the stream should be weighted. See ReviseCPTsByFindings_bn for more information about the relative weighting of cases.

Pass `NULL` for *`control`*. It is only for future expansion.

After adding *`file`* to the `caseset_cs`, you should not modify *`file`* until you are done with the `caseset_cs`.

TEMPORARY LIMITATION: Currently you can only add one file or database retrieval to a caseset.

Version:

Versions 2.26 and later have this function.

See also:

WriteCaseset_cs	Reverse function.
NewCaseset_cs	Create a new Caseset.
DeleteCaseset_cs	Free the resources (e.g., memory) used by the Caseset.
LearnCPTs_bn	Use the Caseset for learning.
TestWithCaseset_bn	Use the Caseset for testing a net.
AddDBCasesToCaseset_cs	Add cases from a database instead of a file.

AddLink_bn

`int AddLink_bn (node_bn* parent, node_bn* child)`

Adds a link from node *`parent`* to node *`child`*, and returns the index of the added link.

This index will be one greater than that of the previously added link, and the existing links will maintain their same indexes.

If *`child`* has a table (CPT or function table), its entries are initially duplicated so its values are the same for each possible state of the new parent. In other words, they are independent of the new parent, so that the link from parent to child has no effect on probability computations until the table is changed.

`parent` must be in the same net as *`child`*, or an error will be generated, and no action taken.

A warning will be generated if there is already a link from *parent* to *child*, or if the added link creates a cycle, but the link will be successfully added. If you don't remove one of the offending links, and later try to compile the net or do node absorption, an error will be generated.

Version:

This function is available in all versions.

See also:

DeleteLink_bn	Removes the link between two nodes
SwitchNodeParent_bn	Switches parents without changing conditional probabilities
CopyNodes_bn	Also duplicates all the links between them

AddNodesFromDB_bn

```
void AddNodesFromDB_bn (dbmgr_cs* dbmgr, net_bn* net,
                        const char* column_names,
                        const char* tables,
                        const char* condition,
                        const char* control)
```

Adds new nodes to *net* corresponding to variables in the database given by *dbmgr* (see NewDBManager_cs), if they aren't there already.

For more information on *column_names*, *tables* and *condition*, see AddDBCasesToCaseset_cs.

control may be "favor_discrete" or "favor_continuous" to control whether to add discrete nodes or continuous nodes for questionable database columns.

This function behaves similarly to the Netica Application menu choice Cases -> Add Case File Nodes. You may want to experiment with that before using this function in your code.

Version:

Versions 3.22 and later have this function.

See also:

NewDBManager_cs	Creates the `dbmgr_cs`
ExecuteDBSql_cs	Execute an arbitrary SQL command
InsertFindingsIntoDB_bn	Insert net findings using SQL INSERT
AddDBCasesToCaseset_cs	Retrieve a set of cases using SQL SELECT

Example:

```
// Create a net with two discrete nodes and whatever states are present in the result-set
// retrieved for the two corresponding columns from Table1 in the the database.
// The column names are "Sex" and "Owns a house", so the names assigned to the nodes will
// be "Sex" and "Owns_a_house", respectively.
dbmgr_cs *dbmgr = NewDBManager_cs (
   "driver={Microsoft Access Driver (*.mdb)}; dbq=.\\myDB.mdb;", "pooling", env);
net_bn* net = NewNet_bn ("testNet", env);
AddNodesFromDB_bn (dbmgr,
                   net,
                   "Sex, \"Owns a house\"",
                   "Table1",
                   NULL,   // NULL => no extra conditions
                   "favor_discrete");
```

AddNodeStates_bn

```
void AddNodeStates_bn (node_bn* node, state_bn* first_state,
                       const char* state_names, int num_states,
                       double cpt_fill)
```

Adds one or more states to *node*, inserting them into the existing states. The first one added will have index *first_state*, and *num_states* is the number of states that will be added. If *state* is zero, the states will be added before existing ones, and if it is the same as the number of states of the node (or -1), they will be added at the end.

The *state_names* parameter can be a comma delimited list of new names for the added states, or it can be NULL, in which case the added states will be given default names. It must be NULL if the node's states currently do not have names.

Pass -1.0 for *cpt_fill*. It is only for future expansion.

This function is for discrete nodes only. It is not for continuousnodes, even if they have been discretized (use SetNodeLevels_bn instead).

All relevant parts of *node* will be properly modified to reflect the changes, including findings. The state titles and state comments of the added states will be absent.

The CPTable will be appropriately adjusted. In it, the probability of the new states will be zero.

Version:

Since version 3.01

See also:

RemoveNodeState_bn	Removes a single state
ReorderNodeStates_bn	Assign a new order to the states
GetNodeNumberStates_bn	*first_state* must be between 0 and this, inclusive
GetStateNamed_bn	Retrieve the new indexes of the states
SetNodeStateName_bn	Sets name of one state at a time
SetNodeStateTitle_bn	Doesn't have the restrictions of a name
SetNodeStateComment_bn	Assigns arbitrary text documentation to a state
SetNodeLevels_bn	For continuous nodes

AddNodeToList_bn

```
void AddNodeToList_bn (node_bn* node, nodelist_bn* nodes, int index)
```

Inserts *node* into the list *nodes*, so that its position is *index*, making the list one longer, and maintaining the order of the rest of the nodes.

index can range from zero (which adds *node* to the front), to LengthNodeList_bn(*nodes*) (which adds *node* to the end), or it can be LAST_ENTRY, which also adds *node* to the end.

If *index* is outside these bounds, or the system runs out of memory, the list will not be modified and an error will be generated.

Adding nodes to the end of the list executes the fastest.

You can also build a node list by creating it full length using NewNodeList2_bn, and then filling it with SetNthNode_bn.

Version:

In versions previous to 2.10, INT_MAX was used instead of LAST_ENTRY

See also:

RemoveNthNode_bn	(reverse operation) Removes a node from the list, shortening it
SetNthNode_bn	Put a node in the list without increasing its length
LengthNodeList_bn	Find maximum value for index
NewNodeList2_bn	Create the node list to start with
DupNodeList_bn	To duplicate a list before modifying it

AddNodeToNodeset_bn

```
void AddNodeToNodeset_bn (node_bn* node, const char* nodeset)
```

Adds *node* to the node-set named *nodeset*.

Creates a new node-set if *nodeset* is not yet present in the net containing *node*.

Version:

Versions 3.22 and later have this function.

See also:

RemoveNodeFromNodeset_bn	(inverse operation) To remove the nodes
IsNodeInNodeset_bn	Determines if a node is in a node-set
SetNodesetColor_bn	Change the display color for Netica Application
ReorderNodesets_bn	To change the priority order of a net's node-sets
GetAllNodesets_bn	Returns string listing all node-sets defined

---

ArgumentChecking_ns

```
checking_ns ArgumentChecking_ns (checking_ns setting, environ_ns* env)
```

Whenever a Netica API function is called, its arguments may be automatically checked for validity. Call this function anytime to adjust the degree of checking Netica does until it is called next.

setting should be one of:

NO_CHECK	No checking
QUICK_CHECK	Only checks things that can be checked very quickly
REGULAR_CHECK	Regular checking
COMPLETE_CHECK	Exhaustively checks everything.
QUERY_CHECK	(See below)

Normally during development, the REGULAR_CHECK setting is used. For debugging, the setting can temporarily be changed to COMPLETE_CHECK, but that will run too slowly for most regular development. Once development is complete, production versions would normally have a setting of QUICK_CHECK (NO_CHECK is discouraged, since it isn't much faster, but its also a possibility), but they may occasionally temporarily change it to REGULAR_CHECK in order to do user input checking.

The previous degree of checking is returned. If QUERY_CHECK is passed for *setting*, then the degree of checking is returned without changing it.

Version:

This function is available in all versions.

See also:

ClearErrors_ns	To start off with a clean slate of no recorded errors

GetError_ns	To get the report for an error that has been detected

---

CalcNodeState_bn

```
state_bn CalcNodeState_bn (node_bn* node)
```

Returns the discrete finding entered for *node* if one has been entered, or the state calculated from its neighbors if that can be done deterministically (e.g., by equation or function table), or else UNDEF_STATE.

If *node* is not a discrete or discretized node, then an error is generated (then use CalcNodeValue_bn instead).

Version:

Versions 1.18 and later have this function.
In versions previous to 2.10, this function was named **GetNodeCalcState_bn**.

See also:

CalcNodeValue_bn	For real values (i.e., continuous)
GetNodeFinding_bn	Doesn't do deterministic propagation

CalcNodeValue_bn

```
double CalcNodeValue_bn (node_bn* node)
```

Returns the real-valued finding entered for *node* if one has been entered, or the real value calculated from its neighbors if that can be done deterministically (e.g., by equation or function table), or else UNDEF_DBL.

If *node* is not a continuous node, and doesn't have a levels list defined (see SetNodeLevels_bn), then an error is generated (then use CalcNodeState_bn instead).

Version:

Versions 1.18 and later have this function.
In versions previous to 2.10, this function was named **GetNodeCalcValue_bn**.

See also:

CalcNodeState_bn	For discrete nodes
GetNodeValueEntered_bn	Doesn't do deterministic propagation

ClearError_ns

```
void ClearError_ns (report_ns* error)
```

Removes the error report error from the system.

The memory used by *error* is freed, so you must not use *error* after calling this. In fact, even any string previously returned by ErrorMessage_ns for this *error* will now be invalid because it is deleted as well.

Does nothing if *error* is NULL.

The name of this function is very similar to ClearErrors_ns, but hopefully there won't be any confusion since their arguments are so different.

Remember that just because you clear away the error record doesn't mean that the problem that caused it has necessarily gone away!

Version:

This function is available in all versions.

See also:

ClearErrors_ns	Clears all the errors
GetError_ns	Retrieves the error to be cleared

Example:

The following function is available in `NeticaEx.c`:

```
// Does the same function as ClearErrors_ns, but is less efficient.
//
void ClearErrors (environ_ns* env, errseverity_ns severity){
    report_ns* error = NULL;
    while (1){
        error = GetError_ns (env, NOTHING_ERR, error);
        if (error == NULL)  break;
        if (ErrorSeverity_ns (error) <= severity)
            ClearError_ns (error);
    }
}
```

Example 2:

See GetError_ns

ClearErrors_ns

`void ClearErrors_ns (environ_ns* env, errseverity_ns severity)`

Removes all errors recorded with environment *`env`* which are as serious as *`severity`*, or less serious.

`severity` must be one of `NOTHING_ERR`, `REPORT_ERR`, `NOTICE_ERR`, `WARNING_ERR`, `ERROR_ERR`, `XXX_ERR` (for a description of these see ErrorSeverity_ns). Pass `XXX_ERR` to clear them all.

The memory used by the error reports is freed, so if you previously used GetError_ns to obtain pointers to some of them, you must not use these pointers any more (for example, don't try to call ClearError_ns, or ErrorMessage_ns with one of them).

The name of this function is very similar to ClearError_ns, but hopefully there won't be any confusion since its arguments are so different.

Remember that just because you clear away the error reports doesn't mean that the problems that caused them have necessarily gone away!

Multithreading Note. In a multithreading environment, ClearErrors_ns is well-behaved in that it only clears the errors that were caused by the current (the calling) thread.

Version:

This function is available in all versions.

See also:

ClearError_ns	Clears just a single error
GetError_ns	Retrieves the error to be cleared

ClearNodeList_bn

`void ClearNodeList_bn (nodelist_bn* nodes)`

Removes all the nodes from the list *`nodes`*.

This does not delete the nodelist; you must still use DeleteNodeList_bn for that.

Version:

Versions 3.06 and later have this function.

See also:

DeleteNodeList_bn	Removes the whole list, and frees the memory it uses
NewNodeList2_bn	Creates a new (empty) node list
RemoveNthNode_bn	
LengthNodeList_bn	Will return 0 after clearing the list
AddNodeToList_bn	To add nodes back in

CloseNetica_bn

```
int CloseNetica_bn (environ_ns* env, char* mesg)
```

Call this when completely finished using the Netica system to free all resources (e.g., memory) that it is using.

env must be a pointer to a global environment initialized by a call to InitNetica2_bn. After calling this, the contents of *env* are invalid and should not be used.

mesg must be a pointer to a character array which is allocated at least MESG_LEN_ns characters long. A good-bye message will be left in it.

No data structure that was returned by any Netica API function will have valid contents after calling CloseNetica_bn.

Netica may be stopped (with CloseNetica_bn) and then later restarted (with NewNeticaEnviron_ns and InitNetica2_bn), but no data structures created by one session may be used by another.

If Netica closed successfully, a non-negative integer is returned, whereas if there was some problem a negative integer is returned and an error message left in mesg. Use the return value to check for an error, rather than the regular Netica error system (e.g., GetError_ns), which will not work after calling this function.

In a multi-threaded environment, ensure that only one thread calls CloseNetica_bn, and after that, no threads may use any Netica function.

Version:

This function is available in all versions.

See also:

NewNeticaEnviron_ns	Creates the environ_ns
InitNetica2_bn	Initializes the environ_ns

Example:

See InitNetica2_bn

CompileNet_bn

```
void CompileNet_bn (net_bn* net)
```

Compiles *net* for fast belief updating (i.e., junction tree propagation).

If the net is an auto-update net (see SetNetAutoUpdate_bn) then belief updating will be done immediately afterwards, but if it isn't, then updating won't be done until you request a belief (e.g., with GetNodeBeliefs_bn).

When the net is compiled, first an "elimination ordering" is determined, which is a list giving all the nodes of the net in some order, and using that list a "junction tree" of cliques is formed. The efficiency of the junction tree may depend greatly on the elimination ordering used, so it is important to find a good elimination ordering. You can

determine the elimination ordering used by calling GetNetElimOrder_bn, and you can control it by calling SetNetElimOrder_bn before calling CompileNet_bn.

Calling CompileNet_bn after the net is already compiled has no effect, unless the net or its elimination ordering has been changed, in which case the net will be recompiled.

To find how efficient the compiling was, use SizeCompiledNet_bn, and to get a report on the internal structures created during compiling, use ReportJunctionTree_bn.

Version:

This function is available in all versions.

See also:

UncompileNet_bn	(reverse operation) Releases memory used by a compiled net
GetNodeBeliefs_bn	Use the compiled net to find new beliefs given findings
SizeCompiledNet_bn	Size and speed of the junction tree formed during compiling
SetNetElimOrder_bn	Set the elimination ordering to use when compiling the net
GetNetElimOrder_bn	Obtain the elimination ordering used to compile the net
ReportJunctionTree_bn	Print out the junction tree formed during compiling
SetNetAutoUpdate_bn	Have compiled net automatically find new beliefs when findings entered
LimitMemoryUsage_ns	In case this function is consuming too much memory

---

CopyNet_bn

```
net_bn* CopyNet_bn (const net_bn* net, const char* new_name,
                                environ_ns* new_env, const char* control)
```

Duplicates *net*, giving the new net the name *new_name* and placing it in the environment *new_env*. Returns the new net.

The name of the new net will be *new_name*. It must be a legal IDname (see IDname in the index), which means it must have NAME_MAX_ns (30) or fewer characters, all of which are letters, digits or underscores, and it must start with a letter. Netica will make a copy of *new_name*; it won't modify or free the passed string.

control allows you to control what gets copied. It can be "no_tables", "no_visual", or both separated by a comma. Including "no_tables" means that the CPT, functional, and experience tables will not be copied. Including "no_visual" means that none of the visual-display details of the net and its nodes will becopied.

The user fields of *net* and its nodes will be copied to the new net (see SetNetUserField_bn and SetNodeUserField_bn), but the user data of the new net and new nodes in that net will be set to NULL (see SetNetUserData_bn and SetNodeUserData_bn).

When you are done with the new net, you should pass it to DeleteNet_bn.

If you just wish to duplicate one or more nodes, see CopyNodes_bn. For other, more fine-grained control over the copying process, you may want to write your own duplicating function, in which case the NeticaEx function DuplicateNet may be a useful model.

Version:

In version 3.05 and later.
Versions 2.28 through 3.04 had a function called **DuplicateNet_bn**.
NeticaEx has a similar function called DuplicateNet.

See also:

CopyNodes_bn	Just duplicate some of the nodes in the net.

Example:

```
// This will duplicate net1's structure, but will not copy its CPT, functional,
// or experience tables.
```

```
net_bn* net2 = CopyNet_bn ( net1, "net2", env, "no_tables" );
```

CopyNodes_bn

nodelist_bn* CopyNodes_bn (const nodelist_bn* nodes, net_bn* new_net, const char* control)

Duplicates *nodes*, putting them in the net *new_net*. It is okay if *new_net* is the same as the net they are already in. All of *nodes* must be in the same net to start with.

A new list of the duplicated nodes will be returned. You should free the list when done with it (e.g., with DeleteNodeList_bn), which won't effect the duplicated nodes. The order of the new list will correspond with the order of the old list. The old list, and the nodes it refers to, will not be modified.

In future, *control* will allow you to control what gets copied. For now, pass NULL.

All connectivity strictly between the duplicated nodes will be maintained during the duplication. Parents of duplicated nodes that aren't also being duplicated will result in disconnected links, if the nodes are being duplicated into a different net.

The user fields (SetNodeUserField_bn) of each node in *nodes* will be copied to the corresponding newly created nodes, but the user data (SetNodeUserData_bn) will not (they will each be set to NULL).

If a duplicated node has the same name as a node already in *new_net*, then the name of the duplicated node will be modified by adding a numeric suffix to its name (or changing its numeric suffix if it already has one).

If you wish to duplicate a single node, see the "DuplicateNode" example below. If you wish to duplicate a whole net, see CopyNet_bn.

Version:

In version 3.05 and later.
Earlier versions had a function called **DuplicateNodes_bn** that did not have the *control* parameter.

See also:

DupNodeList_bn	Just duplicates the list, but not the nodes
NewNode_bn	Creates a new node in a net
DeleteNode_bn	Removes a node from its net and frees it
CopyNet_bn	Duplicates the entire net

Example:

The following function is available in NeticaEx.c:

```
// This transfers nodes from the net they are in to new_net,
// and returns a new list of the new nodes in the same order as they
// appeared in nodes.  The old list nodes is deleted.
//
// In the process each node in nodes is deleted, and a new one created,
// so be sure you don't have any dangling pointers to the old nodes.
//
nodelist_bn* TransferNodes (nodelist_bn* nodes, net_bn* new_net){
    int nn, num_nodes = LengthNodeList_bn (nodes);
    nodelist_bn* new_nodes = CopyNodes_bn (nodes, new_net);
    for (nn = 0;  nn < num_nodes;  ++nn)
        DeleteNode_bn (NthNode_bn (nodes, nn));
    DeleteNodeList_bn (nodes);         // because its full of invalid pointers
    return new_nodes;
}
```

Example 2:

The following function is available in NeticaEx.c:

```
// Handy functions to duplicate a single node.
//
node_bn* DuplicateNode (node_bn* node, net_bn* new_net){
    node_bn* new_node;
    nodelist_bn* nodes    = NewNodeList2_bn (1, GetNodeNet_bn (node));
```

```
    nodelist_bn* newnodes = NewNodeList2_bn (1, new_net);
    SetNthNode_bn (nodes, 0, node);
    newnodes = CopyNodes_bn (nodes, new_net);
    new_node = NthNode_bn (newnodes, 0);
    DeleteNodeList_bn (nodes);
    DeleteNodeList_bn (newnodes);
    return new_node;
}

node_bn* DupNode (node_bn* node){
    return DuplicateNode (node, GetNodeNet_bn (node));
}
```

DeleteCaseset_cs

`void DeleteCaseset_cs (caseset_cs* cases)`

Removes the Caseset and frees all its resources (e.g., memory).

If a file of cases has been added to the caseset, this would have no effect on the actual file.

You must not try to use *`cases`* after calling this.

It is okay if *`cases`* is a `NULL` pointer (then no action is taken).

Version:

Versions 2.26 and later have this function.

See also:

NewCaseset_cs	Create a new Caseset.

DeleteDBManager_cs

`void DeleteDBManager_cs (dbmgr_cs* dbmgr)`

Removes *`dbmgr`* from the system, and releases all the resources it uses (memory, connections, etc.).

You must not try to use *`dbmgr`* after calling this.

It is okay if *`dbmgr`* is a `NULL` pointer (then no action is taken).

Version:

Versions 2.26 and later have this function.

See also:

NewDBManager_cs	Create a new database manager

DeleteLearner_bn

`void DeleteLearner_bn (learner_bn* learner)`

Removes the `learner_bn` object and frees all its resources (e.g., memory).

You must not try to use *`learner`* after calling this.

It is okay if *`learner`* is a `NULL` pointer (then no action is taken).

Version:

Versions 2.26 and later have this function.

See also:

NewLearner_bn	Create a new Learner

DeleteLink_bn

`void DeleteLink_bn (int link_index, node_bn* child)`

Removes the link going to *`child`* from the *`link_index`*th parent node of *`child`*.

`link_index` should be 0 for the first parent, and must be less than the number of links entering *`child`* (the parent ordering is given by GetNodeParents_bn).

It is often more useful to be able to delete a link by specifying the 2 nodes it connects. In order to do this use the function DeleteLink defined in the example below, and in `NeticaEx.c`.

If *`child`* has a CPT or function table, it is collapsed as if the removed parent were taking on its first state (state = 0), unless there is a positive finding entered for the parent, in which case it is collapsed with the parent taking on the finding state.

WARNING: When a link is deleted, keep in mind that the numbering of subsequent links changes. For example, to delete all the links entering a node, use the method "DeleteLinksEntering" example below, not: for (pn = 0; pn < num_parents; ++pn) DeleteLink (pn, child);

WARNING: Keep in mind that after deleting a link into node child, any list of parent nodes for child that was previously returned by GetNodeParents_bn is no longer valid.

Version:

This function is available in all versions.

See also:

AddLink_bn	Adds a link between two nodes
SwitchNodeParent_bn	Switches parents without changing conditional probabilities (can be used to disconnect link instead of deleting)

Example:

The following function is available in `NeticaEx.c`:

```
// Removes the single link from node 'parent' to node 'child'.
// If there is no link from 'parent' to 'child', or more than one, it generates an error.
//
void DeleteLink (node_bn* parent, node_bn* child){
    int pn = IndexOfNodeInList (parent, GetNodeParents_bn (child));
    DeleteLink_bn (pn, child);
}
```

Example 2:

The following function is available in `NeticaEx.c`:

```
// Removes all links entering node child
// See DeleteLink_bn comment for explanation
//
void DeleteLinksEntering (node_bn* child){
    int pn, num_parents = LengthNodeList_bn (GetNodeParents_bn (child));
    for (pn = 0;  pn < num_parents;  ++pn)
        DeleteLink_bn (0, child);
}
```

DeleteNet_bn

`void DeleteNet_bn (net_bn* net)`

Removes *`net`* from the system, and releases all resources it uses (e.g., frees memory), including all its substructures

(e.g., nodes).

You must not try to use *net*, or any of the nodes that were in it, after calling this.

It is okay if *net* is a NULL pointer (then no action is taken).

Version:

In versions previous to 2.09, this function was named **FreeNet_bn**.

See also:

NewNet_bn	Creates a new net
DeleteNode_bn	Removes a node from a net, and releases the memory it uses

---

DeleteNetTester_bn

`void DeleteNetTester_bn (tester_bn* test)`

Removes the tester_bn object and frees all its resources (e.g., memory).

You must not try to use *test* after calling this.

It is okay if *test* is a NULL pointer (then no action is taken).

Version:

Versions 2.08 and later have this function.

See also:

NewNetTester_bn	Construct the tester_bn object

DeleteNode_bn

`void DeleteNode_bn (node_bn* node)`

Removes *node* from its net, and frees all resources (e.g., memory) it was using.

If *node* has children, they will end up with disconnected links for parents, and the names of these links (if they weren't already named) will become the name of *node*. If *node* has parents, then links from them will simply be removed.

If a complete net is to be disposed of, use DeleteNet_bn instead, which also deletes all its nodes.

Version:

This function is available in all versions.

See also:

NewNode_bn	(Inverse operation) Creates a new node in a net
AbsorbNodes_bn	Maintains joint distribution while removing
DeleteNet_bn	Deletes all the nodes of a net

Example:

The following function is available in NeticaEx.c:

```
// Removes all of 'nodes' from their net, and deletes them and node list 'nodes'.
//
void DeleteNodes (nodelist_bn* nodes){
    int i, num = LengthNodeList_bn (nodes);
    for (i = 0;  i < num;  ++i){
        node_bn* node = NthNode_bn (nodes, i);
        SetNthNode_bn (nodes, i, NULL);  // so node list stays legal
        DeleteNode_bn (node);
    }
```

```
    DeleteNodeList_bn (nodes);
}
```

DeleteNodeList_bn

`void DeleteNodeList_bn (nodelist_bn* nodes)`

Releases the memory used by the node list *nodes*.

It doesn't modify the nodes within the list at all.

It is okay if *nodes* is a NULL pointer (then no action is taken).

Don't try to delete the nonmodifiable node lists returned by functions like GetNetNodes_bn and GetNodeParents_bn (they will get deleted when the node or net is deleted).

Version:

In versions previous to 2.09, this function was named **FreeNodeList_bn**.

See also:

DeleteNode_bn	Removes a node from a net, and frees the memory it uses
NewNodeList2_bn	Creates a new (empty) node list

DeleteNodeTables_bn

`void DeleteNodeTables_bn (node_bn* node)`

Deletes *node*'s function table, its CPT table, and its experience table.

It does not modify *node*'s equation or its links.

You don't need to call this function if you are deleting the whole node, since DeleteNode_bn and DeleteNet_bn also delete all of their nodes' tables.

Version:

In versions previous to 2.07, this function was named **DeleteNodeRelation_bn**.

See also:

HasNodeTable_bn	Determine if a node currently has a table
DeleteLink_bn	Reduce the number of parents of the node
SetNodeFuncState_bn	Give a node a function table with its parents
SetNodeProbs_bn	Give a node a probabilistic table (CPT) with its parents
NewNode_bn	Creates a new node without any tables

DeleteSensvToFinding_bn

`void DeleteSensvToFinding_bn (sensv_bn* sens)`

Deletes the sensitivity measuring object *sens*, and frees the memory it uses.

You must not try to use *sens* after calling this.

It is okay if *sens* is a NULL pointer (then no action is taken).

Version:

Versions 2.03 and later have this function.

See also:

NewSensvToFinding_bn	(reverse operation) Create a new `sensv_bn` to do sensitivity analysis

Example:

See NewSensvToFinding_bn.

DeleteStream_ns

```
void DeleteStream_ns (stream_ns* file)
```

Releases the resources (e.g., memory) used by *`file`*.

If *`file`* is for a file-system file (i.e., it was created by NewFileStream_ns), that file will be closed, if necessary, but will not be deleted.

If instead this stream is for a memory buffer (created by NewMemoryStream_ns), this function will not free any memory buffer passed to it by SetStreamContents_ns.

It is okay if *`file`* is NULL (no action is taken).

Version:

Versions 2.09 and later have this function.

See also:

NewFileStream_ns	Creates a new file `stream_ns`
NewMemoryStream_ns	Creates a new memory `stream_ns`

DupNodeList_bn

```
nodelist_bn* DupNodeList_bn (const nodelist_bn* nodes)
```

Duplicates the list *`nodes`*, and returns the duplicate list.

When you are finished with the list returned, pass it to DeleteNodeList_bn (not the Standard C 'free' or 'delete').

This only makes a copy of the list; if you want to duplicate the nodes as well, use CopyNodes_bn.

When Netica API functions return const nodelist_bn*, the returned node lists are volatile (they may become invalid after further Netica calls) and nonmodifiable. Duplicating them with this function removes both of these concerns.

Version:

This function is available in all versions.

See also:

DeleteNodeList_bn	Delete the new list created
NewNodeList2_bn	Make a new empty list
CopyNodes_bn	Duplicates the nodes as well as the list

EnterFinding_bn

```
void EnterFinding_bn (node_bn* node, state_bn state)
```

Enters the discrete finding *`state`* for *`node`*. This means that in the case currently being analyzed, *`node`* is known with certainty to have value *`state`*.

state must be between 0 and n - 1 inclusive, where n is the node's number of states.

If *node* could already have a finding that you wish to override with this new finding, RetractNodeFindings_bn should be called first, otherwise an "inconsistent findings" error could result (see SetNodeFinding in the examples below).

If you wish to pass the state by name, see the "EnterFinding" example below.

If *node* is a continuous node that has been discretized, this function will work fine, but it is better to use EnterNodeValue_bn if the real value is known, for possibly improved accuracy when equations are involved, the case is saved to file, or the discretization changes.

If the net has auto-updating (see SetNetAutoUpdate_bn), then a belief updating will be done to reflect the new finding before this function returns (otherwise it will just be done when needed).

Version:

This function is available in all versions. In versions previous to 3.00 there was a `NeticaEx` function called ChangeFinding that is now called SetNodeFinding.

See also:

EnterFindingNot_bn	To indicate that node isn't in some state
EnterNodeValue_bn	To enter the real value of a continuous node
EnterNodeLikelihood_bn	To enter uncertain findings
GetNodeFinding_bn	To retrieve findings entered so far
RetractNodeFindings_bn	To remove the finding entered
GetNodeNumberStates_bn	*state* must be between 0 and one less than this, inclusive

Example:

The following function is available in `NeticaEx.c`:

```
// This function may be useful if we are not sure whether node
// already has a finding, but if it does we just want to override it.
//
void SetNodeFinding (node_bn* node, state_bn state){
    net_bn* net = GetNodeNet_bn (node);
    int saved = SetNetAutoUpdate_bn (net, 0);    // turning it off can greatly aid efficiency
    RetractNodeFindings_bn (node);
    EnterFinding_bn (node, state);
    SetNetAutoUpdate_bn (net, saved);            // if changing further findings, defer this
step
                                                       if possible, for efficiency
}
```

Example 2:

The following function is available in `NeticaEx.c`:

```
// This function is useful to enter a finding based on the names
// of the node and state.
//
void EnterFinding (char* node_name, char* state_name, net_bn* net){
    node_bn* node = GetNodeNamed_bn (node_name, net);
    state_bn state = GetStateNamed_bn (state_name, node);
    EnterFinding_bn (node, state);
}
```

EnterFindingNot_bn

`void EnterFindingNot_bn (node_bn* node, state_bn state)`

Like EnterFinding_bn, but indicates that the value of *node* is known to not be *state*.

state must be between 0 and n - 1 inclusive, where n is the node's number of states.

This function may be called repeatedly to indicate all the states that you know *node* isn't in. It also works in conjunction with EnterNodeLikelihood_bn to accumulate further observations.

If the net has auto-updating (see SetNetAutoUpdate_bn), then a belief updating will be done to reflect the new finding before this function returns (otherwise it will just be done when needed).

Version:

This function is available in all versions.

See also:

EnterFinding_bn	To enter the finding that a node is in a certain state
EnterNodeLikelihood_bn	The most general way to enter node findings
GetNodeLikelihood_bn	Retrieve negative findings that have been entered
RetractNodeFindings_bn	To remove the negative finding entered
GetNodeNumberStates_bn	*state* must be between 0 and one less than this, inclusive

---

EnterGaussianFinding_bn

```
void EnterGaussianFinding_bn (node_bn* node, double mean,
                              double std_dev)
```

Enters a likelihood finding for *node* equivalent to a Gaussian distribution (normal distribution) with a mean of *mean* and a standard deviation of *std_dev*.

This will not remove any findings already entered for *node* (it will accumulate), so you may want to call RetractNodeFindings_bn first.

node must be a continuous node (see NewNode_bn), but discretized.

If the net has auto-updating (see SetNetAutoUpdate_bn), then a belief updating will be done to reflect the new finding before this function returns (otherwise it will just be done when needed).

To work with Gaussian distribution CPTs, see SetNodeEquation_bn.

Version:

Versions 3.15 and later have this function.

See also:

EnterNodeValue_bn	Enter a point value
EnterIntervalFinding_bn	To enter a uniform distribution interval finding

Example:

```
// This function will clear previously entered finding information
// before entering new gaussian information.
//
void SetGaussianFinding (node_bn* node, double mean, double std_dev){
    net_bn* net = GetNodeNet_bn (node);
    int saved = SetNetAutoUpdate_bn (net, 0);    // turning it off can greatly aid efficiency
    RetractNodeFindings_bn (node);
    EnterGaussianFinding_bn (node, mean, std_dev);
    SetNetAutoUpdate_bn (net, saved);

}
```

EnterIntervalFinding_bn

```
void EnterIntervalFinding_bn (node_bn* node, double low, double high)
```

Enters a likelihood finding for *node* equivalent to an interval extending from low to high: [*low*, *high*].

The likelihood outside the interval is zero, while inside the interval it is uniform (i.e., a "rectangular distribution").

This will not remove any findings already entered for *node* (it will accumulate), so you may want to call RetractNodeFindings_bn first.

If *node* is a continuous node (but discretized, see NewNode_bn), then *low* and *high* refer to continuous values the node can take. Then *high* must be greater than *low*.

If it is a discrete node, then *low* and *high* are state numbers, and so must be integers. In that case, the interval includes both end states (so it is okay if *low* = *high*).

If the net has auto-updating (see SetNetAutoUpdate_bn), then a belief updating will be done to reflect the new finding before this function returns (otherwise it will just be done when needed).

Version:

Versions 3.15 and later have this function.

See also:

EnterNodeValue_bn	Enter a point value for a continuous node
EnterFinding_bn	Enter discrete finding
RetractNodeFindings_bn	To remove the finding entered
EnterGaussianFinding_bn	Enter a Gaussian distribution finding

Example:

```
// This function will clear previously entered finding information
// before entering new interval information.
//
void SetIntervalFinding (node_bn* node, double lo, double hi){
    net_bn* net = GetNodeNet_bn (node);
    int saved = SetNetAutoUpdate_bn (net, 0);    // turning it off can greatly aid efficiency
    RetractNodeFindings_bn (node);
    EnterIntervalFinding_bn (node, lo, hi);
    SetNetAutoUpdate_bn (net, saved);
}
```

EnterNodeLikelihood_bn

`void EnterNodeLikelihood_bn (node_bn* node, const prob_bn* likelihood)`

Enters a likelihood finding for *node*, which is a finding that is not completely certain (it is sometimes called "virtual evidence").

likelihood is a vector containing one probability for each state of *node*.

node must be a discrete or discretized nature node (i.e., not a utility or decision node).

By calling this function several times, you can combine the effects of several independent partial observations. If you don't want the likelihood findings to accumulate, call RetractNodeFindings_bn between calls.

The likelihood finding is equivalent to the following scenario:

There are a number of possible observations you can make: A, B, ... N.

$P(B|S_i)$ denotes the probability of making observation B if the true state of *node* is S_i.

L_B is a vector composed of $<P(B|S_1), P(B|S_2), ..., P(B|S_m)>$ where m is the number of states of *node*.

You actually make observation B, so you enter the vector L_B as a likelihood finding for *node* (or L_A if observation A was made, etc.). You pass it to EnterNodeLikelihood_bn as the *likelihood* parameter.

Notice that each component of a likelihood vector is between 0 and 1 inclusive, they must not all be zero, and they aren't required to sum to 1.

If you enter several accumulating likelihood findings for a node, they should correspond to observations that are independent given the value of the node (if not, look up "likelihood finding, not independent" in the index).

If the net has auto-updating (see SetNetAutoUpdate_bn), then a belief updating will be done to reflect the new finding before this function returns (otherwise it will just be done when needed).

Version:

This function is available in all versions.

See also:

EnterFinding_bn	To enter a certain finding that a node is in some state
EnterFindingNot_bn	To indicate that node isn't in some state
GetNodeLikelihood_bn	To retrieve the likelihood finding entered
RetractNodeFindings_bn	To remove the findings entered

Example:

See GetNodeFinding_bn

EnterNodeValue_bn

```
void EnterNodeValue_bn (node_bn* node, double value)
```

Enters a real number finding for *node* (which is normally a continuous variable node).

If the continuous node has been discretized, then the finding can also be entered as a state using EnterFinding_bn, but if the actual continuous value is known then it is recommended to use that, since it provides more detailed information for functions like WriteNetFindings_bn, and it will automatically be converted to a discrete state when that is needed.

If *node* is continuous discretized, and *value* is out of range (less than the low end of the first state, or more than the high end of the last state), then a suitable error will be generated.

If *node* is discrete (i.e., not continuous), then it must have levels defined, and *value* must exactly match one of the levels.

If the net has auto-updating (see SetNetAutoUpdate_bn), then a belief updating will be done to reflect the new finding before this function returns (otherwise it will just be done when needed).

If *node* could already have a finding that you wish to override with this new finding, RetractNodeFindings_bn should be called first, otherwise an "inconsistent findings" error could result.

Version:

This function is available in all versions. In versions previous to 3.00 there was a `NeticaEx` function called ChangeValue that is now called SetNodeValue.

See also:

EnterFinding_bn	To enter a finding for a discrete or discretized node
GetNodeValueEntered_bn	To retrieve the value entered
RetractNodeFindings_bn	To clear away the finding entered

Example:

The following function is available in `NeticaEx.c`:

```
// This function is useful to enter a new value for node, whether or not it already has one.
//
void SetNodeValue (node_bn* node, double value){
    net_bn* net = GetNodeNet_bn (node);
    int saved = SetNetAutoUpdate_bn (net, 0);    // turning it off can greatly aid efficiency
    RetractNodeFindings_bn (node);
    EnterNodeValue_bn (node, value);
    SetNetAutoUpdate_bn (net, saved);       // if changing further findings,
                                            // defer this step if possible, for efficiency
}
```

EquationToTable_bn

```
void EquationToTable_bn (node_bn* node, int num_samples,
                         bool_ns samp_unc, bool_ns add_exist)
```

Builds the CPT for *node* based on the equation that has been associated with it (see SetNodeEquation_bn).

num_samples is the number of samples to make per parent condition. The higher the number, the more accurate the conversion will be, but the longer it will take. If *node* and its parents are discrete, then it only takes one sample to generate an exact probability, and so in that case this argument is ignored.

samp_unc indicates whether to include in the generated probability table the uncertainty due to the sampling process or not. If the equations are simple (don't have narrow spikes), and the value passed for *num_samples* is high enough, it is better to make this argument FALSE, so that the CPT entries for 'impossible' are zero, rather than close to zero. Otherwise make it TRUE.

Normally you pass FALSE for *add_exist*, but you can pass TRUE if you wish the new sampling to be added to the table which already exists. If the equation conversion to table is nondeterministic (i.e., requires sampling), then calling this function twice with *add_exist* = TRUE is equivalent to calling it once with a value of *num_samples* twice as large. So you can increase the accuracy of the conversion in small steps by repeatedly calling with *add_exist* = TRUE. Or if you want to blend equations (say you want to indicate a 30% chance of equation 1 and a 70% chance of equation 2), you can call it twice, first setting equation 1 and using *num_samples* = 3, then setting equation 2 and using *num_samples* = 7. Similarly, you can blend equations with learned probabilities (see ReviseCPTsByCaseFile_bn), and those entered manually with SetNodeProbs_bn and SetNodeExperience_bn.

Version:

Versions 1.18 and later have this function.

See also:

SetNodeEquation_bn	Specifies the equation to be used
GetNodeProbs_bn	Retrieve the table, if its probabilistic
GetNodeFuncState_bn	Retrieve the table, if its deterministic discrete
GetNodeFuncReal_bn	Retrieve the table, if its deterministic continuous

---

ErrorCategory_ns

```
bool_ns ErrorCategory_ns (errcond_ns errcnd, const report_ns* error)
```

Returns a boolean to indicate whether *error* was caused by the condition *errcnd*.

This is to discover the reason behind an error which has occurred. It groups together errors into broad classes.

For *errcnd* pass one of the conditions below, and the return value will be TRUE iff that was a cause of the error. Note that some errors could have more than one cause.

Possible values for *errcnd* are:

OUT_OF_MEMORY_CND	System did not have enough memory to complete operation
INCONS_FINDING_CND	Inconsistent finding (only)
USER_ABORTED_CND	User halted the function before it completed (not possible when using a Netica API version without the user interface)
FROM_DEVELOPER_CND	Your program indicated the error by calling NewError_ns
FROM_WRAPPER_CND	Error occurred in a language specific converter for VB, JAVA, or C++, etc.

Version:

Versions 1.30 and later have this function.

See also:

ErrorNumber_ns	Return the error's identification number
ErrorMessage_ns	Return a complete error message
ErrorSeverity_ns	Return how serious the error is
GetError_ns	Obtains the report_ns in the first place

ErrorMessage_ns

const char* ErrorMessage_ns (const report_ns* error)

Given a report of an error, this returns a message explaining the error.

The message will start with the name of the Netica API function which was executing when the error occurred (the one you called, not any that are called internally), followed by a colon, and then a descriptive part. Generally the descriptive part is not just a generic message corresponding to the error number, but rather names the elements involved, and describes what went wrong.

If you need the string to persist, make a copy of the string returned, since its contents may become invalid after further calls to Netica API (such as ClearError_ns or ClearErrors_ns). Do not try to directly modify or free the string returned.

Version:

This function is available in all versions.

See also:

ErrorNumber_ns	Returns the error's identification number
ErrorSeverity_ns	Returns how serious the error is
ErrorCategory_ns	Returns what kind of error it is
GetError_ns	Obtains the report_ns in the first place

Example:

See GetError_ns

ErrorNumber_ns

int ErrorNumber_ns (const report_ns* error)

Given a report *error*, this returns its error number, which identifies what type of error it is.

Version:

This function is available in all versions.

See also:

ErrorCategory_ns	Returns a more general categorization of the error
ErrorMessage_ns	Returns a complete error message
GetError_ns	Obtains the report_ns in the first place

ErrorSeverity_ns

errseverity_ns ErrorSeverity_ns (const report_ns* error)

Given *error*, a report of an error which occurred, this returns an indicator of how serious the error is.

These are some of the values, in order from least to most serious, that may be returned:

NOTHING_ERR	Not anything (nothing to report)
REPORT_ERR	Not an error, but a report of success
NOTICE_ERR	Notice of something unusual

WARNING_ERR	Event occurred at "warning" level - requested operation was completed, but results are suspect in some way
ERROR_ERR	Event occurred at "error" level - requested operation was not properly finished, but no internal inconsistencies resulted
XXX_ERR	Internal error, things left inconsistent - continuing could crash system

The less serious errors have lower numerical value, so it is okay to use expressions like >= WARNING_ERR. In fact, it is better to use inequalities than equalities, since later some error levels may be inserted between those of the current list. NOTHING_ERR will always be the lowest, and XXX_ERR will always be the highest.

If the severity is XXX_ERR, then the event causing it is the fault of Netica, and you should contact Norsys about it (support@norsys.com), but if it is any of the others, you should be able to change your software to remove any problems.

Version:

In versions previous to 2.10, this function was named **ErrorDanger_ns**, and errseverity_ns was named errdanger_ns.

See also:

ErrorNumber_ns	Return the error's identification number
ErrorCategory_ns	Return what kind of error it is
GetError_ns	Obtains the report_ns in the first place

ExecuteDBSql_cs

```
void ExecuteDBSql_cs (dbmgr_cs* dbmgr, const char* sql_cmnd,
                      const char* control)
```

Executes *sql_cmnd*, an arbitrary SQL[1] command, on the database managed by *dbmgr*.

This function is useful for doing database administration tasks. Netica makes no attempt to interpret the command; it just passes it directly to the database driver.

If there is a problem with the SQL command, a Netica error will be generated explaining the nature of the problem.

WARNING: This function can severely modify the database.

[1] SQL is a standard query language for accessing databases. To properly use this function, you should have familiarity with SQL.

Pass NULL for *control*; it is only for future expansion.

Version:

Versions 2.26 and later have this function.

See also:

NewDBManager_cs	Creates the dbmgr_cs
InsertFindingsIntoDB_bn	Insert net findings using SQL INSERT
AddDBCasesToCaseset_cs	Retrieve a set of cases using SQL SELECT
AddNodesFromDB_bn	Add nodes to a net using cases from SQL SELECT

Example:

See NewDBManager_cs

FadeCPTable_bn

`void FadeCPTable_bn (node_bn* node, double degree)`

Smoothes the conditional probabilities (CPT) of *`node`* to indicate greater uncertainty, which accounts for the idea that the world may have changed a little since they were last learned.

`degree` must be between 0 and 1, with 0 having no effect and 1 creating uniform distributions with no experience. Calling FadeCPTable_bn once with *`degree`* = 1-d, and again with *`degree`* = 1-f, is equivalent to a single call with *`degree`* = 1-df.

The global variable *`BaseExperience_bn`* is used in the calculation as shown below. It's value should be the same as it was when the learning from cases was done (if it was). It must be greater than 0, and the most common value for it is 1 (1/2 is also commonly used). You will normally set it to one of these choices, depending on your philosophy, and leave it that way permanently.

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 - (prob * exper - *`BaseExperience_bn`*) * *`degree`*)
prob' = normalize (prob * exper * (1 - *`degree`*) + *`degree`* * *`BaseExperience_bn`*)
exper' is obtained as the normalization factor from above. So:
prob' * exper' = prob * exper * (1 - *`degree`*) + *`degree`* * *`BaseExperience_bn`*

When learning in a changing environment, you would normally call FadeCPTable_bn 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`* = $1 - r^{Dt}$ where Dt 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. See the example below.

Version:

In versions previous to 2.10, this function was named **FadeProbs_bn**.

See also:

ReviseCPTsByFindings_bn	Is passed a 'degree', which also can be used to weight the impact of learning a case
GetNodeProbs_bn	Read out the faded probabilities table
GetNodeExperience_bn	Read out the faded experience table

Example:

The following function is available in `NeticaEx.c`:

```
// The following does the same fading for a list of nodes:
//
void FadeCPTsForNodes (const nodelist_bn* nodes, double degree){
    int nn, num_nodes = LengthNodeList_bn (nodes);
    for (nn = 0;  nn < num_nodes;  ++nn)
        FadeCPTable_bn (NthNode_bn (nodes, nn), degree);
}
```

Example 2:

```
// The following bit of code may be executed in a loop which is
//   traversed as the cases are learned, in order to do the
//   required fading:
//     time - the occurrence time of the last case learned
//     lasttime - a number initialized to the time of the 1st case
//     mindelay - a number controlling how often fading is done
//     rate - a number determining how much fading is done
//     net - the net being learned
if (time - lasttime >= mindelay){
    double degree = 1 - pow (rate, time - lasttime);
    FadeCPTsForNodes (GetNetNodes_bn (net), degree);
    lasttime = time;
}
```

FindingsProbability_bn

```
double FindingsProbability_bn (net_bn* net)
```

Returns the joint probability of the findings entered into net so far (including any negative or likelihood findings).

If the computations for belief updating haven't been done since the last findings were entered, or the last net modifications made, they will be done before this function returns, which can be quite time consuming.

WARNING: The number will not be valid if likelihood findings were entered.

Version:

In versions previous to 2.10, this function was named **CaseProbability_bn**.

See also:

JointProbability_bn	Explore probability of case without entering findings
IsBeliefUpdated_bn	Indicates if FindingsProbability_bn will trigger belief updating
GetNodeBeliefs_bn	Finds the marginal probability for each of the nodes

GenerateRandomCase_bn

```
int GenerateRandomCase_bn (const nodelist_bn* nodes, int method,
                           double timeout, void* gen)
```

Generates a random case for *nodes* (i.e., positive findings for each of them), by sampling from a probability distribution matching that of the net containing *nodes*, and conditioned on all findings already entered in the net.

If *method* is 1, then the net must be compiled, and the junction tree is used to do very fast sampling with no rejections (i.e., findings don't slow it down).

If *method* is 2, then forward sampling is used. This evaluates equations directly if they are available, rather than just using CPT entries (which may just approximate the equation). However, it uses a rejection method, so it may be very slow if the findings currently entered are improbable.

If *method* is 0 (the recommended value), then the default method is used. Currently this is method 2 if rejections won't be a problem or the net is uncompiled, otherwise method 1.

timeout indicates how much time to allocate for the task (in relative units). If it cannot finish in time, it will return a negative quantity (no Netica error will be generated). If *method* is 1, or no findings are entered, then it always returns successfully, and within a fixed amount of time, so then *timeout* is ignored.

Pass `NULL` for *gen*; it is only for future expansion.

Version:

In versions previous to 2.26, this function did not have the *gen* parameter.
In versions previous to 2.09, this function was named **RandomCase_bn**.
In versions previous to 1.07, this function always used forward sampling.

See also:

GetNodeFinding_bn	Retrieve the random case generated
GetNodeValueEntered_bn	Retrieve the random case generated for a continuous node, method 0
NewNodeList2_bn	Create the node list

GetAllNodesets_bn

```
const char* GetAllNodesets_bn (net_bn* net, bool_ns include_system,
                               void* vis)
```

Returns a string which is a list of all node-sets defined for *net*, separated by commas, in priority order, with highest priority first.

If *include_system* is TRUE, then the returned list will also contain nodesets internally defined by Netica, otherwise it will just contain user-defined ones. Each internally defined node-set will have a dash ("-") preceeding its name.

Pass NULL for *vis*; it is only for future expansion.

The lifetime of the string returned is only until this function is called next on the same net; it should not be used after that.

Version:

Versions 3.22 and later have this function.

See also:

AddNodeToNodeset_bn	Creates the user-defined node-sets that appear in the list
IsNodeInNodeset_bn	Determines if a node is in a node-set
SetNodesetColor_bn	How the node-set is displayed in Netica Application
ReorderNodesets_bn	To change the priority order of a net's node-sets

GetError_ns

```
report_ns* GetError_ns (environ_ns* env, errseverity_ns severity,
                        const report_ns* after)
```

If *after* is NULL, this returns a report for the first error in environment *env* which is at least as serious as *severity*.

If *after* is an error report currently in *env*, then the next error after it (at least as serious as *severity*) will be returned. This can be used to step through the errors, as shown in the second example below.

If there is no such error to be returned, GetError_ns returns NULL.

severity must be one of NOTHING_ERR, REPORT_ERR, NOTICE_ERR, WARNING_ERR, ERROR_ERR, or XXX_ERR (for a description of these see ErrorSeverity_ns).

This function can be used to report every error condition detected by (or occurring within) any Netica function except InitNetica2_bn and CloseNetica_bn (for these two functions you should use their return values to check for errors), and NewNeticaEnviron_ns (its errors will be detected by the subsequent call to InitNetica2_bn).

Use ErrorNumber_ns to get its error number, and ErrorMessage_ns to get an error message for it.

Multithreading Note. In a multithreading environment, GetError_ns is well-behaved in that it only returns errors that were caused by the current (the calling) thread.

Version:

This function is available in all versions.

See also:

ErrorMessage_ns	Gets a message for the error report
ErrorNumber_ns	Gets the identification number for the error report
ErrorSeverity_ns	Returns how serious the error is
ClearError_ns	Removes the error report from the system

Example:

```
// The following prints each serious error and then removes it
//
while ((error = GetError_ns (env, ERROR_ERR, NULL)) != NULL){
    printf ("%s\n", ErrorMessage_ns (error));
    ClearError_ns (error);
}
```

Example 2:

The following function is available in `NeticaEx.c`:

```
// This function prints all the serious errors (without removing them)
//   that are currently registered with the environment in global variable 'env'.
//
void PrintErrors (void){
    report_ns* error = NULL;
    while (1) {
        error = GetError_ns (env, ERROR_ERR, error);
        if (!error)  break;
        printf ("%s\n", ErrorMessage_ns (error));
    }
}
```

GetInputNamed_bn

`int GetInputNamed_bn (const char* name, const node_bn* node)`

Returns the link index number of the link whose name is *`name`*, or -1 if there isn't one with that name (case sensitive comparison). This is the same index as would be used to find the parent of the link in the node list returned by GetNodeParents_bn.

The value returned is particular to the node passed; another node may have a link with the same name, but a different link index.

Netica won't modify or free the passed *`name`* string.

Version:

In versions 1.17 and earlier, this function was named **LinkNamed_bn**.

In versions 1.18 to 3.04, this function was named **InputNamed_bn**.

See also:

GetNodeInputName_bn	(inverse function) Returns the name of a link given its index

---

GetMutualInfo_bn

`double GetMutualInfo_bn (sensv_bn* sens, const node_bn* Vnode)`

Measures the mutual information between two nodes, which is how much a finding at one node (called the "varying node") is expected to alter the beliefs (measured as decrease in its entropy) at another node (called the "query node").

The query node is set by the particular `sensv_bn` created (see NewSensvToFinding_bn). The varying node is passed as *`Vnode`*.

This function returns the mutual information between two nodes (measured in bits). It can be used with any discrete or discretized nodes. Mutual information is the expected reduction in entropy of one node (measured in bits) due to a finding at another.

The maximum possible decrease in entropy of the query node is when entropy goes to zero, i.e., all uncertainty is removed. That happens when a finding is obtained for the query node itself. So to find the entropy of a node,

measure the mutual information between a node and itself (that is why "entropy" is sometimes called "self information").

To create a sensv_bn that can measure mutual information, pass ENTROPY_SENSV for *what_find* when calling NewSensvToFinding_bn. For its *Vnodes* argument, pass a list of all the nodes that might later be passed as *Vnode* to this function.

Mutual information is symmetric between nodes (i.e., it has the same value when varying and query nodes are reversed). That makes this function useful to measure the degree to which one varying node can effect a number of different query nodes in an efficient way, by just passing the varying node to NewSensvToFinding_bn, and each of the query nodes to this function.

The mutual information between two nodes can depend greatly on what findings are entered elsewhere in the net, and this function will properly take that into account.

The first time this function is called by some sensv_bn after the findings of a net have changed, it takes longer to return, but after that, for each *Vnode* passed, it returns quickly.

This function is available as "Network -> Sensitivity to Finding" in Netica Application. For more information on it, contact Norsys for the "Sensitivity" document.

Version:

Versions 2.03 and later have this function. In versions previous to 3.05, this function was named **MutualInfo_bn**.

See also:

GetVarianceOfReal_bn	Use a different measure of sensitivity: variance reduction
NewSensvToFinding_bn	Create the sensv_bn
GetTestLogLoss_bn	Get the "logarithmic loss" score, when testing a Bayes net with case data

Example:

See NewSensvToFinding_bn.

GetNetAutoUpdate_bn

```
int GetNetAutoUpdate_bn (const net_bn* net)
```

Returns BELIEF_UPDATE, or greater, if belief updating will be done automatically whenever some finding (positive, likelihood or value) is entered for a node in *net*, otherwise it returns 0. The returned value can later be passed to SetNetAutoUpdate_bn to restore the current condition.

SetNetAutoUpdate_bn also returns the value of auto-update (the value it had before it got changed).

Version:

This function is available in all versions.
In versions previous to 2.11, the documentation for this function only specified that a value greater than 0 was returned if auto belief updating was turned on (and in fact 1 was returned).

See also:

SetNetAutoUpdate_bn	Sets value, and returns old value

Example:

See EnterFinding_bn for an example of saving and restoring auto-update.

GetNetComment_bn

```
const char* GetNetComment_bn (const net_bn* net)
```

Returns a C character string which contains the comment associated with *net*, or the empty string (rather than

NULL) if *net* does not have a comment.

The comment may contain anything, but see SetNetComment_bn for what it's meant to contain.

There is no restriction on the length of the comment, or on what characters it might contain.

If you need the string to persist, make a copy of the string returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the string returned.

Version:

This function is available in all versions.

See also:

SetNetComment_bn	Sets it
GetNodeComment_bn	Get the comment for a particular node

GetNetElimOrder_bn

```
const nodelist_bn* GetNetElimOrder_bn (const net_bn* net)
```

Returns a list of the nodes of *net* in their "elimination order" (which is used for triangulation in the compilation of *net*), or NULL if there is no order currently associated with *net*.

Compiling a net, or using SetNetElimOrder_bn, can add an elimination ordering to a net, while changing the net structure, or using SetNetElimOrder_bn, can remove an ordering from the net.

Only appropriate nodes will be included in the list returned (for example, nodes of kind CONSTANT_NODE won't be).

If you need the list to persist, make a copy of the list returned (with DupNodeList_bn), since its contents may become invalid after further calls to Netica API. This is a list managed by Netica (declared *const*), so do not call any function to modify or free it (e.g., DeleteNodeList_bn).

Version:

This function is available in all versions.

See also:

SetNetElimOrder_bn	Sets it
SizeCompiledNet_bn	See how good the ordering is
ReportJunctionTree_bn	Analyze the effect of the order

GetNetFileName_bn

```
const char* GetNetFileName_bn (const net_bn* net)
```

Returns a C character string which is the name of the file (including full path) that *net* was last written to or read from.

If *net* was not read from a file, and has not yet been written to a file, the empty string (rather than NULL) is returned.

If you need the string to persist, make a copy of the string returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the list returned.

Version:

Versions 2.09 and later have this function.

See also:

GetNetName_bn	The actual internal name of the net

GetNetTitle_bn	A descriptive title for the net
ReadNet_bn	Initializes net's filename with the name of the file read
WriteNet_bn	Sets or changes net's filename

GetNeticaVersion_bn

`int GetNeticaVersion_bn (environ_ns* env, const char** version)`

Returns the version number of Netica, multiplied by 100. For example, if the version of Netica currently running is 1.21, then 121 is returned.

If *`version`* is not `NULL`, then *`*version`* is set to a C string providing information on the version of Netica running. This consists of the full version number, a space, a code for the type of machine it is running on, a comma, the name of the program, and finally a code indicating some build information (in parentheses). An example is:

2.06 Win, Netica (AB)

This function can be called before InitNetica2_bn (but of course it must be called after NewNeticaEnviron_ns, because that is needed to form *`env`*).

Do not try to modify or free the string returned as *`*version`*.

Version:

This function is available in all versions.

See also:

NewNeticaEnviron_ns	Form the required environ_ns

Example:

The following function is available in `NeticaEx.c`:

```
// This requires a global variable env,
// initialized by a call to NewNeticaEnviron_ns:
//   env = NewNeticaEnviron_ns (NULL, NULL, NULL);
//
void PrintNeticaVersion (void){
    const char* version;
    GetNeticaVersion_bn (env, &version);
    printf ("Version of Netica running: %s\n", version);
}
```

GetNetName_bn

`const char* GetNetName_bn (const net_bn* net)`

Returns a C character string which is the name of *`net`*.

This may be different from the file name used to save *`net`*, and different from *`net`*'s title.

You can count on the name to be present, and to be a legal IDname (see IDname in the index), which means that it is `NAME_MAX_ns` (30) or fewer characters (not including terminating 0)..

Note that two different nets in Netica's memory may have the same name.

If you need the string to persist, make a copy of the string returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the string returned.

Version:

This function is available in all versions.

See also:

SetNetName_bn	Sets it

GetNetTitle_bn	Longer, unrestricted label
GetNetFileName_bn	Gets the name of file the net has last been read from or saved to

---

GetNetNodes_bn

```
const nodelist_bn* GetNetNodes_bn (const net_bn* net)
```

Returns a list of all the nodes in *net*.

If *net* has no directed cycles, the list will be in topological order (i.e., a parent will always appear before its children).

To obtain the number of nodes in the net, use the length of the returned list (it will not contain duplicates or NULL entries).

If you need the list to persist, make a copy of the list returned (with DupNodeList_bn), since its contents may become invalid after further calls to Netica API (e.g., one that changes the nodes of a net, such as NewNode_bn). This is a list managed by Netica (declared *const*), so do not call any function to modify or free it (e.g., DeleteNodeList_bn).

Consecutive calls to this function may yield lists in different orders, so if you are depending on a consistent ordering (for example, by using integers to index the nodes), construct a fixed list by calling DupNodeList_bn on the list returned.

Version:

This function is available in all versions.

See also:

DeleteNode_bn	Removes a node from the net
NewNode_bn	Creates a new node for the net

GetNetNthUserField_bn

```
void GetNetNthUserField_bn (const net_bn* net, int index,
                            const char** name, const void** value,
                            int* length, int kind)
```

This returns the user-defined named field (i.e., attribute-value) data associated with *net*, by index rather than field name.

It works equivalent to GetNodeNthUserField_bn; for more information, see that function.

For more information on user fields, see SetNodeUserField_bn.

Version:

Versions 2.07 and later have this function.

See also:

SetNetUserField_bn	Sets them
GetNetUserField_bn	Retrieve field by name
GetNodeNthUserField_bn	The equivalent function for nodes

GetNetTitle_bn

```
const char* GetNetTitle_bn (const net_bn* net)
```

Returns a C character string which is the title of *net*, or the empty string (rather than NULL) if *net* does not have a title.

This may be different from *net*'s "name", and from the file name used to save *net*.

There is no restriction on the length of the title, or on what characters it might contain.

If you need the string to persist, make a copy of the string returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the string returned.

Version:

This function is available in all versions.

See also:

SetNetTitle_bn	Sets it
GetNetName_bn	Gets the net's name (limited chars and length, always exists)
GetNetFileName_bn	Gets the name of file the net has last been read from or saved to
GetNodeTitle_bn	Same, but for nodes

GetNetUserData_bn

```
void* GetNetUserData_bn (const net_bn* net, int kind)
```

Returns a pointer to information previously attached to *net* using SetNetUserData_bn, or NULL if none has been attached.

This information is understood only by the program using the Netica API. It may point to whatever is desired, possibly a large structure with many fields. It is not saved to file with *net* (for that see GetNetUserField_bn).

Pass 0 for *kind*. It is only for future expansion.

Version:

This function is available in all versions.

See also:

SetNetUserData_bn	Sets it
GetNetUserField_bn	Named field (attribute-value) data, which gets saved to file with net
GetNodeUserData_bn	Retrieve the user pointer attached to a particular node

GetNetUserField_bn

```
const char* GetNetUserField_bn (const net_bn* net, const char* name,
                                int* length, int kind)
```

Retrieves the user-defined data associated with *net* on a field-by-field basis.

It works exactly like GetNodeUserField_bn; see that function for usage information.

Version:

Versions 2.00 and later have this function.

See also:

SetNetUserField_bn	Sets it
GetNetNthUserField_bn	Retrieve field by index. Iterate over fields
GetNetUserData_bn	For user-managed data which is not saved to file with net
GetNodeUserField_bn	Field-by-field data attached to a particular node

---

GetNodeBeliefs_bn

```
const prob_bn* GetNodeBeliefs_bn (node_bn* node)
```

Returns a belief vector indicating the current probability for each state of *node* (each entry is a prob_bn, which is a 'float').

The vector will be indexed by states, with one probability for each state (if required, the state indexes can be found from their names using GetStateNamed_bn). It will be normalized, so that the sum of its entries is 1.

This provides the current beliefs (i.e., posterior probabilities) that the variable represented by *node* is in each of its states, given the net model and all findings entered into all nodes of the net (positive findings, negative findings and likelihood findings).

The net containing *node* must have been compiled before calling this (with CompileNet_bn), or an error will be generated. If the net has been modified it must be recompiled, but just entering findings does not require a recompile.

node should be a discrete or discretized nature node.

If belief updating hasn't been done since the last findings were entered, it will be done before this function returns, which can be time consuming (you can call IsBeliefUpdated_bn before calling this to find out if belief updating will be done).

If you need the beliefs to persist, make a copy of the vector returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the vector returned.

Version:

This function is available in all versions.

See also:

IsBeliefUpdated_bn	Tells whether GetNodeBeliefs_bn will trigger belief updating
GetNodeNumberStates_bn	Determine length of vector returned
JointProbability_bn	More than one node at a time
CompileNet_bn	To do the initial compiling before entering findings
GetNodeExpectedUtils_bn	Get the resulting expected utility of a decision node
GetNodeExpectedValue_bn	For numeric nodes, get expected value and standard deviation

Example:

```
// This function is useful to get the belief that a certain node is in
// a certain state, based on the names of the node and state.
```

The following function is available in `NeticaEx.c`:

```
//
double GetNodeBelief (char* node_name, char* state_name, net_bn* net){
    node_bn* node = GetNodeNamed_bn (node_name, net);
    state_bn state = GetStateNamed_bn (state_name, node);
    return GetNodeBeliefs_bn (node) [state];
}
```

GetNodeChildren_bn

```
const nodelist_bn* GetNodeChildren_bn (const node_bn* node)
```

Returns a list of the children of *node*. Those are the nodes that have a link going to them from *node*. If it has no children then the empty list (rather than NULL) will be returned.

If there are several links from *node* to the same child, then that child will appear only once in the list returned, so the length of the returned list may be used to provide the number of unique children of *node*.

If you need the list to persist, make a copy of the list returned (with DupNodeList_bn), since its contents may become invalid after further calls to Netica API (e.g., one that changes the links of a net, such as AddLink_bn). This is a list managed by Netica (declared *const*), so do not call any function to modify or free it (e.g., DeleteNodeList_bn).

Consecutive calls to this function may yield lists in different orders, so if you are depending on a consistent ordering (for example, by using integers to index the nodes), construct a fixed list by calling DupNodeList_bn on the list returned.

Version:

This function is available in all versions.

See also:

GetNodeParents_bn	Get a list of the parents
AddLink_bn	Create a new child
DeleteLink_bn	Remove a child

GetNodeComment_bn

```
const char* GetNodeComment_bn (const node_bn* node)
```

Returns a C character string which contains the comment associated with *node*, or the empty string (rather than NULL) if *node* does not have a comment.

The comment may contain anything, but see SetNodeComment_bn for what it's meant to contain.

There is no restriction on the length of the comment, or on what characters it might contain.

If you need the string to persist, make a copy of the string returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the string returned.

Version:

This function is available in all versions.

See also:

SetNodeComment_bn	Sets it
GetNetComment_bn	Get the comment for a whole net

GetNodeEquation_bn

```
const char* GetNodeEquation_bn (const node_bn* node)
```

Returns a null terminated C character string which contains the equation associated with *node*, or the empty string (rather than NULL), if *node* does not have an equation.

For information on Netica equations, see the "Equation" chapter of Netica Application's onscreen help.

If you need the string to persist, make a copy of the string returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the string returned.

Version:

Versions 1.30 and later have this function.

See also:

SetNodeEquation_bn	Sets it
EquationToTable_bn	If this hasn't been done, equation may not match CPT table

---

GetNodeExpectedUtils_bn

```
const util_bn* GetNodeExpectedUtils_bn (node_bn* node)
```

Returns a vector providing the expected utility of each choice in a decision node, considering findings currently entered (each entry is a util_bn, which is a 'float').

The vector will be indexed by states, with one utility for each state (i.e., choice).

The net containing *node* must be a decision net (i.e., have decision and utility nodes), which has been compiled before calling this (with CompileNet_bn), or an error will be generated. If the net has been modified it must be recompiled, but just entering findings does not require a recompile.

Before calling this all preceding decision nodes must have positive findings entered.

node must be a decision node.

If belief updating hasn't been done since the last findings were entered, it will be done before this function returns, which can be time consuming (you can call IsBeliefUpdated_bn before calling this to find out if belief updating will be done).

If you need the utilities to persist, make a copy of the vector returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the vector returned.

If you wish to retrieve a whole table of values, providing the optimal state to choose given the parent states, use GetNodeFuncState_bn.

Version:

Versions 1.07 and later have this function.

See also:

GetNodeFuncState_bn	Retrieve table of optimal choices as a function of parent values
GetNodeNumberStates_bn	Determine the length of the vector returned
GetNodeBeliefs_bn	For the beliefs of a nature node
GetNodeExpectedValue_bn	Don't confuse it with this function, which gets the expected real value of a nature node

---

GetNodeExpectedValue_bn

```
double GetNodeExpectedValue_bn (node_bn* node, double* std_dev,
                                double* x3, double* x4)
```

Returns the expected real value of *node*, based on the current beliefs for *node*, and if *std_dev* is non-NULL, **std_dev* will be set to the standard deviation. Returns UNDEF_DBL if the expected value couldn't be calculated.

Pass NULL (i.e., 0) for arguments *x3* and *x4*; they are there for future expansion.

node must be continuous discretized, or must be discrete with a levels list defined to supply real values.

For continuous discretized nodes it assumes the belief for each state is distributed evenly over each discretized interval. Because of that, it can't handle infinite tails (returns UNDEF_DBL).

This function is not for expected utility; for that see GetNodeExpectedUtils_bn.

Version:

Versions 2.09 and later have this function.

See also:

GetNodeBeliefs_bn	Returns beliefs for each state individually
GetNodeExpectedUtils_bn	For expected utility, rather than expected real value

---

GetNodeExperience_bn

```
double GetNodeExperience_bn (const node_bn* node,
                             const state_bn* parent_states)
```

Given *parent_states*, a vector of states for the parents of *node*, this returns the "experience" of the node for the situation described by the parent states.

The experience is also known as the "number of cases", ess, or estimated sample size (see the Learning Nets chapter).

If no experience value has been assigned to this parent configuration (either by learning or SetNodeExperience_bn), then UNDEF_DBL is returned, without generating an error.

The order of the states in *parent_states* should match the order of the nodes in the list returned by GetNodeParents_bn (this will be the same order that parents were added using AddLink_bn). MapStateList_bn may be useful for that. *parent_states* can be NULL if *node* has no parents.

To cycle through all the possibilities of *parent_states*, see the NeticaEx function NextStates.

Version:

This function is available in all versions.

See also:

SetNodeExperience_bn	Sets them
GetNodeProbs_bn	Get the corresponding probability vector
ReviseCPTsByFindings_bn	Increments experience
ReviseCPTsByCaseFile_bn	Experience will measure the number of cases with each parent configuration
MapStateList_bn	To create the state list passed in

GetNodeFinding_bn

```
state_bn GetNodeFinding_bn (const node_bn* node)
```

If a positive finding has been entered for *node*, this returns the finding. If no findings have been entered it returns NO_FINDING, and it can also return NEGATIVE_FINDING or LIKELIHOOD_FINDING.

The value returned will be one of:

>= 0	The positive (certain) finding which has been entered
NO_FINDING	No findings have been entered, or likelihood findings exactly cancel
NEGATIVE_FINDING	One or more negative findings have been entered
LIKELIHOOD_FINDING	One or more likelihood findings have been entered

The value returned indicates the simplest way to express the accumulation of all findings entered for *node* since the last retraction. For instance, if the node has 3 possible states, and negative findings have been entered for 2 of them, then the value returned will be the remaining state. If the accumulation of all likelihood findings entered so far result in a likelihood vector with only one nonzero entry, that state will be returned. If it results in a likelihood vector with some zero entries and some nonzero, but equal, entries, NEGATIVE_FINDING will be returned; if entries are unequal then LIKELIHOOD_FINDING will be returned, and if they are all equal then NO_FINDING will be returned (i.e., a number of likelihood findings which exactly canceled each other were entered).

Note that positive findings cannot cancel; if 2 differing positive findings are entered for a node, an error is generated.

If you wish to obtain the actual result of accumulated likelihood or negative findings, use GetNodeLikelihood_bn.

This function is for discrete or discretized nodes; for continuous nodes, use GetNodeValueEntered_bn.

Version:

This function is available in all versions.

See also:

GetNodeLikelihood_bn	To get likelihood or negative findings
GetNodeValueEntered_bn	To get a real valued finding for a continuous node
EnterFinding_bn	To enter a finding
RetractNodeFindings_bn	To clear away all findings entered so far for this node

---

GetNodeFuncReal_bn

```
double GetNodeFuncReal_bn (const node_bn* node,
                           const state_bn* parent_states)
```

This is for deterministic nodes that are continuous or have been given real levels (e.g., by SetNodeLevels_bn). Given a vector of states for the parents of *node*, this returns the real value of *node* (which is functionally determined by the parent values) by looking it up in the nodes function table. If the function table between *node* and its parents has not yet been created, or if it is probabilistic (i.e., a CPT) rather than deterministic, this returns UNDEF_DBL, without generating an error.

If *node* is discrete, with no real levels defined, an error will be generated (use GetNodeFuncState_bn instead).

The order of the states in *parent_states* should match the order of the nodes in the list returned by GetNodeParents_bn (this will be the same order that parents were added using AddLink_bn). MapStateList_bn may be useful for that. *parent_states* can be NULL if *node* has no parents.

If the node has been given a deterministic equation with SetNodeEquation_bn, you must call EquationToTable_bn before this can be used to retrieve values (if you needed to find values without generating the whole table you would enter findings for the parents, and use CalcNodeValue_bn).

This function ignores any findings entered in the net.

To cycle through all the possibilities of *parent_states*, see the NeticaEx function NextStates.

Version:

Versions 2.06 and earlier didn't have this function, but had one called **GetNodeFuncValue_bn**, which worked almost the same, but took both discrete and continuous nodes (i.e., combined this and GetNodeFuncState_bn).

See also:

SetNodeFuncReal_bn	Sets them
GetNodeFuncState_bn	Same, but returns state integer instead of real value
IsNodeDeterministic_bn	To check if this function is applicable
MapStateList_bn	To create the state list passed in

GetNodeFuncState_bn

```
int GetNodeFuncState_bn (const node_bn* node,
                         const state_bn* parent_states)
```

This is for discrete or discretized nodes that are deterministic. Given a vector of states for the parents of *node*, this returns the state of *node* (which is functionally determined by the parent values) by looking it up in the nodes function tablse. If the function table between *node* and its parents has not yet been created, or if it is probabilistic (i.e., a CPT) rather than deterministic, this returns UNDEF_STATE, without generating an error.

If *node* is continuous, and not discretized, an error will be generated (use GetNodeFuncReal_bn instead).

The order of the states in *parent_states* should match the order of the nodes in the list returned by GetNodeParents_bn (this will be the same order that parents were added using AddLink_bn). MapStateList_bn may be useful for that. *parent_states* can be NULL if *node* has no parents.

If the node has been given a deterministic equation with SetNodeEquation_bn, you must call EquationToTable_bn before this can be used to retrieve values (if you needed to find values without generating the whole table you would enter findings for the parents, and use CalcNodeState_bn).

If *node* is a decision node in a decision net which has been compiled, and belief propagation has been done by calling GetNodeExpectedUtils_bn on *node*, then this function can be used to read out the table of optimal decisions under the different scenarios indicated by *parent_states*.

If SetNodeProbs_bn was used to provide *node* with conditional probabilities that were all 0 or 1, GetNodeFuncState_bn can be used to retrieve the deterministic state of *node* as a function of its parents.

This function ignores any findings entered in the net.

To cycle through all the possibilities of *parent_states*, see the NeticaEx function NextStates.

Version:

Versions 2.06 and earlier didn't have this function, but had one called **GetNodeFuncValue_bn**, which worked almost the same, but took both discrete and continuous nodes (i.e., combined this and GetNodeFuncReal_bn).

See also:

SetNodeFuncState_bn	Sets it
GetNodeFuncReal_bn	Same, but returns real value instead of state integer
IsNodeDeterministic_bn	To check if this function is applicable
GetNodeProbs_bn	For nondeterministic discrete nodes
MapStateList_bn	To create the state list passed in

GetNodeInputName_bn

```
const char* GetNodeInputName_bn (const node_bn* node, int input_index)
```

Returns a string which is the name for input number *input_index* of *node*, or the empty string (rather than NULL) if the link does not have a name. Numbering for *input_index* starts at 0 and proceeds in the same order as parents returned by GetNodeParents_bn.

If the name is present, you can count on it to be a legal IDname (see IDname in the index), which means that it is NAME_MAX_ns (30) or fewer characters (not including terminating 0).

Input names are used to document what each link means, local to the node, which is especially important if the link is disconnected, or if its parents are continuously being switched. They are also useful as local parameters in equations, instead of using the names of parents' nodes, so the equation stays valid even if the parents change.

If you need the string to persist, make a copy of the string returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the string returned.

Version:

In versions 1.17 and earlier, this function was named **GetLinkName_bn**.

See also:

SetNodeInputName_bn	Sets it
GetNodeParents_bn	Gets the actual parents of the links
GetInputNamed_bn	(inverse function) Returns input index given the name

---

GetNodeKind_bn

```
nodekind_bn GetNodeKind_bn (const node_bn* node)
```

Returns whether *node* is a nature, decision, utility or constant node.

The value returned will be one of:

NATURE_NODE	Bayes nets are composed only of this type (and constant nodes). This is a "chance" or "deterministic" node of an influence diagram.
DECISION_NODE	Indicates a variable that can be controlled. This is a "decision" node of an influence diagram.
UTILITY_NODE	A variable to maximize the expected value of. This is a "value" node of an influence diagram.
CONSTANT_NODE	A fixed parameter, useful as an equation constant. When its value changes, equations should be reconverted to CPT tables, and maybe the net recompiled.
DISCONNECTED_NODE	The (virtual) parent node of a link which has been disconnected. See example program below.

Version:

In versions 1.09 and earlier, CONSTANT_NODE was called ASSUME_NODE.

See also:

SetNodeKind_bn	Sets it
IsNodeDeterministic_bn	To distinguish between "chance" and "deterministic" nodes
GetNodeType_bn	Indicates whether the node is for a discrete or continuous variable

Example:

The following function is available in NeticaEx.c:

```
// Returns whether link 'link_index' entering 'node' is disconnected.
//
bool_ns IsLinkDisconnected (int link_index, const node_bn* node){
    const node_bn* parent = NthNode_bn (GetNodeParents_bn (node), link_index);
    return GetNodeKind_bn (parent) == DISCONNECTED_NODE;
}
```

GetNodeLevels_bn

```
const level_bn* GetNodeLevels_bn (const node_bn* node)
```

Returns the list of numbers used to enable a continuous node to act discrete, or enables a discrete node to provide real-valued numbers. Levels are used to discretize continuous nodes, or to map from discrete nodes to real numbers. See SetNodeLevels_bn for a full description of the level numbers (they are 'level_bn's, which are 'double's).

Returns NULL if *node* does not have a levels list.

If you need the results to persist, make a copy of the vector returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the vector returned.

Since the usage of levels is a little different for each type of node, each is discussed separately:

***node* is continuous:** (GetNodeType_bn would return DISCRETE_TYPE)

The length of the list returned is one more than the number of states of *node*. The node is discretized into states, and the list returned has the thresholds (monotonically increasing or decreasing). Each range is from *levels*[state] to *levels*[state+1]., where *levels* is the list returned. Normally each interval includes its lower endpoint, but not its upper.

***node* is discrete:** (GetNodeType_bn would return CONTINUOUS_TYPE)

The length of the list returned is the number of states of *node*, with each element being the real number associated with the corresponding state.

Version:

In version 2.07 and later. Earlier versions had a function called **GetNodeLevel_bn**, which took an extra argument representing the state, and returned a single level.

See also:

SetNodeLevels_bn	Sets them
GetNodeNumberStates_bn	Length of the vector returned (plus one if *node* continuous)

GetNodeLikelihood_bn

```
const prob_bn* GetNodeLikelihood_bn (const node_bn* node)
```

Returns a likelihood vector with one entry for each state of *node*, indicating the findings that have been entered for *node* since the last retraction, including positive findings, negative findings, and likelihood findings.

If a positive finding has been entered, then the likelihood vector will consist of all zero entries, except a nonzero entry for the state corresponding to the finding (for more details on likelihood vectors, see EnterNodeLikelihood_bn).

If a number of likelihood findings and/or negative findings have been entered for this node, they will be accumulated in the vector returned.

If no findings have been entered for this node since the last retraction, a uniform vector consisting of all 1's is returned. This is consistent with the concept of likelihood, since a likelihood finding which is a uniform vector is equivalent to no finding at all, and will not modify any beliefs.

This function cannot be used on a continuous node, unless the node has first been discretized (see SetNodeLevels_bn).

If you need the results to persist, make a copy of the vector returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the vector returned.

Version:

This function is available in all versions.

See also:

GetNodeFinding_bn	Returns a scalar saying whether findings have been entered, and what kind
GetNodeBeliefs_bn	Current belief for a node (considers findings entered at other nodes)
EnterNodeLikelihood_bn	To enter a vector of uncertain findings (see example below)
RetractNodeFindings_bn	To clear away all findings entered so far for this node
GetNodeNumberStates_bn	Determine the length of the vector returned

Example:

```
// To temporarily remove all findings from a node and later restore them
//
```

```
int size = GetNodeNumberStates_bn (node) * sizeof (prob_bn);
char* save = malloc (size);
memcpy (save, GetNodeLikelihood_bn (node), size);
RetractNodeFindings_bn (node);
... [stuff without the evidence] ...
RetractNodeFindings_bn (node);                  // in case any findings were introduced above
EnterNodeLikelihood_bn (node, save);            // restores to original
free (save);
```

GetNodeName_bn

`const char* GetNodeName_bn (const node_bn* node)`

Returns a C character string which is the name of *node*.

You can count on the name to be present, and to be a legal IDname (see IDname in the index), so it is guaranteed to be NAME_MAX_ns (30) or fewer characters (not including terminating 0).

Don't confuse this with the "title" of *node*, which is available by GetNodeTitle_bn.

If you need the string to persist, make a copy of the string returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the string returned.

Version:

This function is available in all versions.

See also:

SetNodeName_bn	Sets it
GetNodeNamed_bn	(inverse function) Gets the node given its name
GetNodeTitle_bn	Gets the node's title

GetNodeNamed_bn

`node_bn* GetNodeNamed_bn (const char* name, const net_bn* net)`

Returns the node of *net* which has a name exactly matching *name* (case sensitive comparison). If there is no such node, it will return NULL (without generating an error).

name can be any string; it need not be a legal IDname (of course if is not legal, NULL will be returned).

To search a node list for a node with a given name, see the FindNodeNamed example below.

Netica won't modify or free the passed name string.

Version:

This function is available in all versions. In versions previous to 3.05, this function was named **NodeNamed_bn**.

See also:

GetNodeName_bn	(inverse function) Returns the name of the node

Example:

The following function is available in NeticaEx.c:

```
// Like GetNodeNamed_bn, but generates an error if the name doesn't exist.
// In versions prior to 3.05 this function was called NodeNamed
//
node_bn* GetNode (char* node_name, net_bn* net){
    node_bn* node = GetNodeNamed_bn (node_name, net);
    if (node == NULL)
        NewError (env, 0, ERROR_ERR,      // for NewError, see NewError_ns
                    "NodeNamed: There is no node named '%s' in net '%s'",
                    node_name, GetNetName_bn (net));
```

```
    return node;
}
```

Example 2:

The following function is available in NeticaEx.c:

```
// Returns the index of the node identified by name in the list nodes,
//   or -1 if it doesn't appear.
// All of nodes must be in the same net.
//
int FindNodeNamed (const char* name, const nodelist_bn* nodes){
    if (LengthNodeList_bn (nodes) == 0)  return -1;
    else {
        net_bn* net = GetNodeNet_bn (NthNode_bn (nodes, 0));
        node_bn* node = GetNodeNamed_bn (name, net);
        if (node == NULL)  return -1;
        return IndexOfNodeInList_bn (node, nodes, 0);
    }
}
```

GetNodeNet_bn

```
net_bn* GetNodeNet_bn (const node_bn* node)
```

Returns the net that *node* is part of. Every node is part of some net.

Version:

This function is available in all versions.

See also:

GetNetNodes_bn	(inverse function) Get the list of nodes comprising a net
CopyNodes_bn	Copy nodes from one net to another
NewNet_bn	Originally created the net

GetNodeNthUserField_bn

```
void GetNodeNthUserField_bn (const node_bn* node, int index,
                             const char** name, const void** value,
                             int* length, int kind)
```

Returns the user-defined named field (i.e., attribute-value) data associated with *node*, by index rather than field name. For more information on user fields, see SetNodeUserField_bn.

Pass any non-negative integer for *index*. If it is larger than the last field, **name* will be set to the empty string (""), and *length* will be set to -1.

**name* will be set to the name of the field which was passed to SetNodeUserField_bn when the data was set, **value* will be set to the data, and **length* set to the length that was also passed in.

If *value* or *length* are NULL, they won't be set.

Pass 0 for *kind*. It is only for future expansion.

This function is meant to iterate through the various fields. Don't assume that their ordering will remain the same after a call to SetNodeUserField_bn on *node*.

Netica always places two null bytes after the end of the data (withoutaltering *length* of course), which is of no consequence if the data isarbitrary bytes, but may be helpful if it is an ascii or Unicode string, andyou want to safely retrieve it solely by pointer, ignoring *length*.

If you need the result to persist, make a copy of the data returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the string returned.

Version:

Versions 2.07 and later have this function.

See also:

SetNodeUserField_bn	Sets them
GetNodeUserField_bn	Retrieve field by name
GetNetNthUserField_bn	The equivalent function for nets

GetNodeNumberStates_bn

```
int GetNodeNumberStates_bn (const node_bn* node)
```

Returns the number of states that *node* can take on, or zero if *node* is a continuous node that hasn't been discretized.

Version:

This function is available in all versions.

See also:

NewNode_bn	Sets the number of states for a discrete node
SetNodeLevels_bn	Sets the number and boundaries of discretization for a continuous node
GetNodeStateName_bn	

Example:

See the examples for SetNodeProbs_bn.

GetNodeParents_bn

```
const nodelist_bn* GetNodeParents_bn (const node_bn* node)
```

Returns a list of the parents of *node*. Those are the nodes with a link going to *node*. If it has no parents then an empty list (rather than `NULL`) will be returned.

The order of the list is significant. Numbering each node in the list (starting from 0) provides a numbering for the links entering *node*, which is used by some other functions.

If there are several links from the same parent to *node*, then in the list returned that parent will be repeated once for each link.

To obtain the number of links entering *node*, use the length of the returned list.

If you need the list to persist, make a copy of the list returned (with DupNodeList_bn), since its contents may become invalid after further calls to Netica API (e.g., one that changes the links of a net, such as AddLink_bn). This is a list managed by Netica (declared *const*), so do not call any function to modify or free it (e.g., DeleteNodeList_bn).

Version:

This function is available in all versions.

See also:

GetNodeChildren_bn	Gets a list of the children
LengthNodeList_bn	Use on returned list to find the number of parents
GetNodeInputName_bn	
AddLink_bn	Add a parent
DeleteLink_bn	Remove a parent
SwitchNodeParent_bn	Switch one of the parents for a different one

GetNodeKind_bn	To determine if a link is disconnected (returns DISCONNECTED_NODE)

GetNodeProbs_bn

```
const prob_bn* GetNodeProbs_bn (const node_bn* node,
                                const state_bn* parent_states)
```

Returns the conditional probabilities of *node*, given that its parents are in the states indicated by the *parent_states* vector, by looking them up in the node's CPT table. The length of *parent_states* must be the number of parents of *node*, and each of its entries provides a state for the corresponding parent. The length of the array returned is the number of states of *node*, and consists of 'prob_bn's (i.e.'float's), which are the conditional probabilities:

P (*node* = state0 | parents take on *parent_states*)

P (*node* = state1 | parents take on *parent_states*)

...

P (*node* = stateN | parents take on *parent_states*)

Notice that it is not conditioned on any findings (evidence) entered into the net, so its value will not change as findings are added or belief updating is done.

NULL will be returned if no CPT table has been associated with *node* (for example by SetNodeProbs_bn, SetNodeFuncState_bn, EquationToTable_bn, ReviseCPTsByCaseFile_bn or ReviseCPTsByFindings_bn), or if the table has been removed (for example by DeleteNodeTables_bn), but no error will be generated. If you use only SetNodeEquation_bn to indicate a node's relation with its parents, you must also call EquationToTable_bn before this will return non-NULL.

The order of the states in *parent_states* should match the order of the nodes in the list returned by GetNodeParents_bn (this will be the same order that parents were added using AddLink_bn). MapStateList_bn may be useful for that. *parent_states* can be NULL if *node* has no parents.

parent_states should not include EVERY_STATE or UNDEF_STATE.

If SetNodeFuncState_bn was used to provide *node* with a function table, then GetNodeProbs_bn can be used to retrieve that table in the form of conditional probabilities, which will all be 0 or 1.

If you need the results to persist, make a copy of the vector returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the vector returned.

To get all the conditional probabilities of *node* at once, see the GetNodeAllProbs example below.

To cycle through all the possibilities of *parent_states*, see the NeticaEx function NextStates.

If *parent_states* is NULL then the entire table is returned.

Version:

This function is available in all versions.

See also:

SetNodeProbs_bn	Sets them
HasNodeTable_bn	Determine if GetNodeProbs_bn is going to return NULL
GetNodeBeliefs_bn	Conditioned on findings, but not parents
AbsorbNodes_bn	Can be used to find probabilities conditioned on parents and findings
GetNodeFuncState_bn	For deterministic nodes
GetNodeExperience_bn	The confidence of the probabilities obtained
GetNodeParents_bn	Indicates the order of entries in parent_states
GetNodeNumberStates_bn	Length of the array returned (plus one if node continuous)
MapStateList_bn	To create the state list passed in

Example:

```
// To just get the probability that node is in state, given parent_states
```

```
//
double prob = GetNodeProbs_bn (node, parent_states) [state];
```

Example 2:

The following function is available in NeticaEx.c:

```
// Puts all the conditional probabilities of node into the array probs.
// You could allocate probs as follows (SizeCartesianProduct is defined
// in NeticaEx.c):
//   probs = malloc (SizeCartesianProduct (GetNodeParents_bn (node)) *
//                   GetNodeNumberStates_bn (node) * sizeof (prob_bn));
//
void GetNodeAllProbs (node_bn* node, prob_bn* probs){
    nodelist_bn* parents = GetNodeParents_bn (node);
    int num_states  = GetNodeNumberStates_bn (node);
    int num_parents = LengthNodeList_bn (parents);
    state_bn st, *parent_states = calloc (num_parents, sizeof (state_bn));
    while (1){
        const prob_bn* vecp = GetNodeProbs_bn (node, parent_states);
        if (!vecp)  break;
        for (st = 0;  st < num_states;  ++st)  *probs++ = *vecp++;
        if (NextStates (parent_states, parents))          // defined in NeticaEx.c
            break;
        if (GetError_ns (env, ERROR_ERR, NULL))  break;
    }
    free (parent_states);
}
```

GetNodeStateComment_bn

```
const char* GetNodeStateComment_bn (const node_bn* node,
                                    state_bn state)
```

Given an integer index representing a state of *node*, this returns the associated comment of that state, or the empty string (rather than NULL) if it does not have a comment.

There is no restriction on the length of the comment, or on what characters it might contain. *node* may have some states commented, and others not.

If you need the string to persist, make a copy of the string returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the string returned.

Version:

Versions 2.26 and later have this function.

See also:

SetNodeStateComment_bn	Sets it
GetNodeStateTitle_bn	Gets the state's title
GetNodeNumberStates_bn	*state* must be between 0 and one less than this, inclusive

---

GetNodeStateName_bn

```
const char* GetNodeStateName_bn (const node_bn* node, state_bn state)
```

Given an integer index representing a state of *node*, this returns the associated name of that state, or the empty string (rather than NULL) if it does not have a name.

Either all of the states have names, or none of them do.

If the name is present, you can count on it to be a legal IDname (see IDname in the index), which means that it is NAME_MAX_ns (30) or fewer characters (not including terminating 0).

If you need the string to persist, make a copy of the string returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the string returned.

Version:

This function is available in all versions.

See also:

SetNodeStateName_bn	Sets it
SetNodeStateNames_bn	Sets names of all states at once
GetStateNamed_bn	(inverse function) Returns the state with a given state name
GetNodeStateTitle_bn	
GetNodeNumberStates_bn	*state* must be between 0 and one less than this, inclusive

---

GetNodeStateTitle_bn

```
const char* GetNodeStateTitle_bn (const node_bn* node, state_bn state)
```

Given an integer index representing a state of *node*, this returns the associated title of that state, or the empty string (rather than NULL) if it does not have a title.

This may be different from *state*'s "name" (see GetNodeStateName_bn).

There is no restriction on the length of the title, or on what characters it might contain. *node* may have some states titled, and others not.

If you need the string to persist, make a copy of the string returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the string returned.

Version:

Versions 1.18 and later have this function.

See also:

SetNodeStateTitle_bn	Sets it
GetNodeStateName_bn	Returns the state's restricted name
GetNodeStateComment_bn	Returns the state's comment
GetNodeNumberStates_bn	*state* must be between 0 and one less than this, inclusive

---

GetNodeTitle_bn

```
const char* GetNodeTitle_bn (const node_bn* node)
```

Returns a C character string which is the title of *node*, or the empty string (rather than NULL) if node does not have a title.

This may be different from *node*'s "name" (see GetNodeName_bn).

There is no restriction on the length of the title, or on what characters it might contain.

If you need the string to persist, make a copy of the string returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the string returned.

Version:

This function is available in all versions.

See also:

SetNodeTitle_bn	Sets it
GetNodeName_bn	Gets the node's name (limited chars and length, always exists)

GetNodeType_bn

```
nodetype_bn GetNodeType_bn (const node_bn* node)
```

Returns DISCRETE_TYPE if the variable corresponding to *node* is discrete (digital), and CONTINUOUS_TYPE if it is continuous (analog).

It should be emphasized that the value returned only concerns the underlying physical variable that *node* represents, not how *node* acts within the net. For example, continuous variables may be discretized by subdividing them into ranges, and discrete variables may provide real values in continuous settings. See SetNodeLevels_bn for more information.

In order to determine whether a node may act discrete, call GetNodeNumberStates_bn, and if the returned value is greater than 0 then the node can act as a discrete node.

There is no API function to change the type of a node; this can only be set when the node is first created by NewNode_bn. A continuous node may be discretized anytime, by using SetNodeLevels_bn.

WARNING: This function may return other types in the future, so check the return value completely and act appropriately if it has an unexpected value.

Version:

In version 2.07 and later.
Earlier versions had a function called **GetNodeDiscrete_bn**, which returned TRUE iff a node was discrete.

See also:

GetNodeNumberStates_bn	To determine if a node can act discrete
SetNodeLevels_bn	To discretize a continuous variable, so it may act discrete
NewNode_bn	Originally sets the value that GetNodeType_bn returns
IsNodeDeterministic_bn	Return whether a node is deterministically related to its parents
GetNodeKind_bn	Whether the node is a nature, decision, utility, constant, etc

GetNodeUserData_bn

```
void* GetNodeUserData_bn (const node_bn* node, int kind)
```

Returns a pointer to information previously attached to *node* using SetNodeUserData_bn, or NULL if no pointer has been attached yet.

This information is understood only by the program using Netica API. It may point to whatever is desired, possibly a large structure with many fields. It is not saved to file with *node* (for that see GetNodeUserField_bn).

Pass 0 for *kind*. It is only for future expansion.

Version:

This function is available in all versions.

See also:

SetNodeUserData_bn	Sets it
GetNodeUserField_bn	Named field (attribute-value) data, which is saved to file
GetNetUserData_bn	Retrieve the user pointer attached to a whole net

GetNodeUserField_bn

```
const char* GetNodeUserField_bn (const node_bn* node,
                                 const char* name, int* length, int kind)
```

Returns the user-defined data associated with *node* on a named-field basis (i.e., attribute-value). For more information, see SetNodeUserField_bn.

For *name* pass the name of the field to be read, which was passed to SetNodeUserField_bn when the data was set.

Pass 0 for *kind*. It is only for future expansion.

This function will return a pointer to the data, and set **length* to the length that was passed into SetNodeUserField_bn. If you try to retrieve a field that was never set, this will return a null terminated empty string (""), and will set **length* to -1.

Netica always places two null bytes after the end of the data (without altering length of course), which is of no consequence if the data is arbitrary bytes, but may be helpful if it is an ascii or Unicode string, and you want to safely retrieve it solely by pointer, ignoring *length*. The return type is a pointer to char, which is also of no consequence if arbitrary bytes were passed to SetNodeUserField_bn, but is handy if strings were passed, which is recommended when feasible.

If you need the result to persist, make a copy of the data returned, since its contents may become invalid after further calls to Netica API. Do not try to directly modify or free the string returned.

Some helpful functions to read user fields which are integers, real numbers and strings are: GetNodeUserInt, GetNodeUserNumber and GetNodeUserString, which are provided in `NeticaEx.c`, and in the examples below. See SetNodeUserField_bn for the matching functions that do the setting.

Version:

Versions 2.00 and later have this function.

See also:

SetNodeUserField_bn	Sets them
GetNodeNthUserField_bn	Retrieve field by index. Iterate over fields
GetNodeUserData_bn	For user-managed data, which is not saved to file
GetNetUserField_bn	User field data attached to the whole net

Example:

```
// To get a user field which is an ascii string
//
const char* GetNodeUserString (node_bn* node, const char* fieldname){
    return GetNodeUserField_bn (node, fieldname, NULL, 0);
}
```

Example 2:

The following function is available in `NeticaEx.c`:

```
// To get a user field which is an integer
//
#include <stdlib.h>

long GetNodeUserInt (node_bn* node, const char* fieldname){
    int length;
    const char* str = GetNodeUserField_bn (node, fieldname, &length, 0);
    if (length == -1)
        NewError (env, 0, ERROR_ERR,
                  "GetNodeUserInt: There is no user field named '%s' in node '%s'",
                  fieldname, GetNodeName_bn (node));
    else {
        char* end;
        long num = strtol (str, &end, 10);
        if (*end != 0)
            NewError (env, 0, ERROR_ERR,
                 "GetNodeUserInt: Field named '%s' of node '%s' was not storing an integer",
                 fieldname, GetNodeName_bn (node));
        else return num;
```

```
    }
    return 0;
}
```

Example 3:

The following function is available in NeticaEx.c:

```
// To get a user field which is a real number
//
#include <stdlib.h>

double GetNodeUserNumber (node_bn* node, const char* fieldname){
    int length;
    const char* str = GetNodeUserField_bn (node, fieldname, &length, 0);
    if (length == -1)
    NewError (env, 0, ERROR_ERR,
                "GetNodeUserInt: There is no user field named '%s' in node '%s'",
                fieldname, GetNodeName_bn (node));
    else {
        char* end;
        double num = strtod (str, &end);
        if (*end != 0)
            NewError (env, 0, ERROR_ERR,
                "GetNodeUserInt: Field named '%s' of node '%s' was not storing a number",
                fieldname, GetNodeName_bn (node));
        else return num;
    }
    return 0;
}
```

GetNodeValueEntered_bn

`double GetNodeValueEntered_bn (const node_bn* node)`

Returns the real-valued finding entered for *node*, or UNDEF_DBL if none has been entered since the last retraction.

Usually this function is for continuous nodes. If *node* is not a continuous node, but has been given a levels list, and a discrete finding has been entered, then that finding will be converted to a real-value by the levels list, and the real-value returned (see SetNodeLevels_bn for an explanation of the levels list).

If *node* is not a continuous node, and doesn't have a levels list defined, then an error is generated.

Version:

In version 1.18 and earlier, this function was named **GetNodeValue_bn**.

See also:

GetNodeFinding_bn	For discrete findings, rather than continuous
CalcNodeValue_bn	Will compute from neighbors if deterministic
EnterNodeValue_bn	To enter a real-valued finding into a node
RetractNodeFindings_bn	To clear away all findings entered into a node

---

GetNodeVisPosition_bn

`void GetNodeVisPosition_bn (const node_bn* node, void* vis, double* x, double* y)`

Sets **x*, **y* to the coordinates of the center of *node*, as it would appear in a visual display (e.g., in Netica Application).

Pass NULL for *vis*; it is only for future expansion.

Version:

Versions 2.07 and later have this function, and versions 1.15 to 2.06 have an equivalent function called **GetNodeCenter_bn**, but that supplied ints instead of doubles.

See also:

SetNodeVisPosition_bn	Sets it
SetNodesetColor_bn	Gets color

GetNodeVisStyle_bn

```
const char* GetNodeVisStyle_bn (node_bn* node, void* vis)
```

Returns the current "style descriptor" of *node* for any visual display (e.g., in Netica Application).

The returned string may be used as a parameter to SetNodeVisStyle_bn.

Pass NULL for *vis*; it is only for future expansion.

Version:

Versions 3.05 and later have this function.

See also:

SetNodeVisStyle_bn	Sets the style
GetNodeVisPosition_bn	Retrieves display coordinates

GetNthNet_bn

```
net_bn* GetNthNet_bn (int nth, environ_ns* env)
```

This function can be used to iterate through the nets that are currently in Netica's memory.

Start from 0, and pass successively higher values for *nth*, until it returns the net you are searching for.

When *nth* is the number of nets in memory, or larger, NULL will be returned.

Don't count on the ordering of nets to stay the same, as nets are added and removed.

Version:

Versions 2.07 and greater have this function.

See also:

NewNet_bn	Create a new net
ReadNet_bn	Read a net from file

Example:

The following function is available in NeticaEx.c:

```
// Returns the first net with 'newname', if there is one, otherwise NULL.
#include <string.h>

net_bn* NetNamed (const char* name){
    int nth = 0;
    net_bn* net;
    do {
        net = GetNthNet_bn (nth++, env);
    } while (net && strcmp (name, GetNetName_bn (net)) != 0);
    return net;
}
```

GetRelatedNodes_bn

```
void GetRelatedNodes_bn (nodelist_bn* related_nodes,
                         const char* relation,
                         const node_bn* node)
```

Finds all the Nodes that bear the relationship *relation* with *node* and puts them in *related_nodes*.

relation should be one of "parents", "children", "ancestors", "descendents", "connected", "markov_blanket", "d_connected", or the singular version of any of the above (which does the same thing - see IsNodeRelated_bn for singular versions), or any of the above with various modifiers appended.

Modifiers may be appended (with comma seperators) in any order to the string containing the relation. The allowed modifiers are:

append	add to the list that is passed in (otherwise, that list is first emptied).
union	add to the list that is passed in and remove all duplicates.
intersection	reduce the passed-in list to only the nodes that are in both the original passed-in list and the relation.
subtract	take the nodes that are in the relation away from the passed-in list.
include_evidence_nodes	Note: only relevant for "markov_boundary" and "d_connected". Without it the relation list will not contain any nodes with findings.
exclude_self	Note: only relevant for: "ancestors", "descendents", "connected", and "d_connected". Without it the relation list will also include *node* (it being generation 0).

Note: The definition of "ancestor", "descendent", "connected", and "d_connected" is such that a node is considered a level-0 "ancestor", "descendent", etc. of itself. Append "exclude_self" (e.g., "ancestor,exclude_self") if you only wish to start from level-1.

If you wish to pass a list of nodes, instead of a single node, for *node*, then use the function GetRelatedNodesMult_bn instead.

Version:

Versions 3.05 and later have this function.

See also:

GetRelatedNodesMult_bn	Same, but for all members of a nodelist
IsNodeRelated_bn	Tests relationship of two nodes
GetNodeParents_bn	Specialized form for 'parents'
GetNodeChildren_bn	Specialized form for 'children'
GetNetNodes_bn	Get all nodes in the net

Example:

```
//Find all the descendants of a node, excluding the node itself.
nodelist_bn* descendants = NewNodeList2_bn (0, GetNodeNet_bn (node));
GetRelatedNodes_bn (descendants, "descendants,exclude_self", node);
```

GetRelatedNodesMult_bn

```
void GetRelatedNodesMult_bn (nodelist_bn* related_nodes,
                             const char* relation,
                             const nodelist_bn* nodes)
```

Finds all the Nodes that bear the relationship *relation* with any member of *nodes* and puts them in

related_nodes.

Otherwise it works the same as GetRelatedNodes_bn; see that function for more information.

Note: It is okay if *related_nodes* = *nodes* (i.e., the list gets modified in-place).

On entry, *nodes* must not contain duplicates (but *related_nodes* may).

Tip: A handy and efficient way to remove the duplicates from any node list is to call this function with the node list as *related_nodes*, an empty list for *nodes*, and "parents,union" as the relation.

Version:

Versions 3.05 and later have this function.

See also:

GetRelatedNodes_bn	Same, but for a single node
IsNodeRelated_bn	Tests relationship of two nodes
GetNetNodes_bn	Get all nodes in the net

Example:

```
//find all the parents of all the children of a node
nodelist_bn* children =  GetNodeChildren_bn (node);
nodelist_bn* parentsOfChildren = NewNodeList2_bn (0, GetNodeNet_bn (node));
GetRelatedNodesMult_bn (parentsOfChildren, "parents", children);
```

Example 2:

```
//find all the descendants of the children of a node, excluding the children themselves.
//I.e., find grand-children, great-grandchildren, great-great-grandchildren, ...
nodelist_bn* children =  GetNodeChildren_bn (node);
nodelist_bn* descendants = NewNodeList2_bn (0, GetNodeNet_bn (node));
GetRelatedNodesMult_bn (descendants, "descendants,exclude_self", children);
```

GetStateNamed_bn

`state_bn GetStateNamed_bn (const char* name, const node_bn* node)`

Returns the index number of the state whose name is *name*, or UNDEF_STATE if there isn't one (case sensitive comparison).

The result is only valid for *node*; another node may have a state with the same name, but a different state number.

name can be any string; it need not be a legal IDname (of course if it isn't, UNDEF_STATE will be returned).

Netica won't modify or free the passed *name* string.

The returned value is a 'state_bn', which is just another name for an 'int', but used to indicate that the int stands for a state index.

Version:

This function is available in all versions. In versions previous to 3.05, this function was named **StateNamed_bn**.

See also:

GetNodeStateName_bn	(inverse function) Returns the state name given its index
GetNodeNumberStates_bn	The value returned will be between 0 and one less than this, inclusive

---

GetStreamContents_ns

`const char* GetStreamContents_ns (stream_ns* strm, long* length)`

Returns the memory buffer associated with memory stream *strm*, which will contain the results of any writing Netica has done to the stream.

The result returned is only valid until the next call using this memory stream. If the result is needed longer than that, it should be duplicated. Do not attempt to modify or free the result returned. Calling this function does not remove or alter the buffer's contents; a subsequent call will yield the same results.

**length* will be set to the number of bytes in the buffer, excluding any null terminator. It's value upon entry is ignored.

If *strm* is not a memory stream (i.e., created by NewMemoryStream_ns), an error will be generated.

Version:

Versions 2.26 and later have this function.

See also:

NewMemoryStream_ns	Create new memory stream
SetStreamContents_ns	Sets buffer

Example:

See NewMemoryStream_ns.

Example 2:

See SetStreamPassword_ns.

GetTestConfusion_bn

```
double GetTestConfusion_bn (tester_bn* test, node_bn* node,
                            int predictedState, int actualState)
```

Returns the number of times the Net predicted *predictedState* for *node*, but the case file actually held *actualState* as the value of that node, during the performance test of a net. These are the entries of a table traditionally called the "confusion matrix".

For each case, the "prediction" is formed by reading the values of the "observed nodes" of that case in the file, using them to update beliefs in the net, and then picking the state of *node* which has the highest resultant belief (posterior probability) to be the prediction. The set of "observed nodes" is specified when creating the `tester_bn`.

node is required to have been in the *test_nodes* list originally passed to NewNetTester_bn.

Version:

Versions 2.08 and later have this function.

See also:

GetTestErrorRate_bn	Get the fraction of test cases for which the prediction failed
GetTestLogLoss_bn	Get the "logarithmic loss" score of the test
GetTestQuadraticLoss_bn	Get the "quadratic loss" score of the test
NewNetTester_bn	Construct the `tester_bn` object

Example:

See NewNetTester_bn for a program that creates a `tester_bn`, and uses the below function.

The below function appears in `NeticaEx.c`:

```
// Prints a confusion matrix table.  Use after a tester_bn has run its tests.
// This function can be found in examples\TestNet.c
// that comes with this distribution.
//
void PrintConfusionMatrix (tester_bn* tester, node_bn* node){
   int i,a,p;
   int numstates = GetNodeNumberStates_bn (node);
   printf ("\nConfusion matrix for %s:\n", GetNodeName_bn (node));
   for (i=0;  i < numstates;  ++i)
       printf ("\t %s", GetNodeStateName_bn (node, i));
   printf ("\t Actual\n");
   for (a=0;  a < numstates;  ++a){
       for (p=0;  p < numstates;  ++p)
```

```
            printf ("\t %d", (int) GetTestConfusion_bn (tester, node, p, a));
        printf ("\t %s\n", GetNodeStateName_bn (node, a));
    }
    printf ("\n");
}

// Sample output:
Confusion matrix for Cancer:
       Present  Absent   Actual
       11       1        Present
       4        184      Absent
```

GetTestErrorRate_bn

`double GetTestErrorRate_bn (tester_bn* test, node_bn* node)`

Returns the accumulated "error rate" of *node* under the tests previously performed with *test*. This is the fraction of times the Net predicted or diagnosed states incorrectly for *node*, out of all the cases which provided a value for *node*.

For each case, the "prediction" is formed by reading the values of the "observed nodes" of that case in the file, using them to update beliefs in the net, and then picking the state of *node* which has the highest resultant belief (posterior probability) to be the prediction. The set of "observed nodes" is specified when creating the `tester_bn`.

A result of 0.0 means no prediction errors, whereas a result of 1.0 means all predictions were in error.

node is required to have been in the *test_nodes* list originally passed to NewNetTester_bn.

Version:

Versions 2.08 and later have this function.

See also:

GetTestConfusion_bn	Get elements of the confusion matrix
GetTestLogLoss_bn	Get the "logarithmic loss" score of the test
GetTestQuadraticLoss_bn	Get the "quadratic loss" score of the test
NewNetTester_bn	Construct the `tester_bn` object

Example:

See NewNetTester_bn.

GetTestLogLoss_bn

`double GetTestLogLoss_bn (tester_bn* test, node_bn* node)`

Returns the "logarithmic loss" of *node* under the tests previously performed with *test*.

The "logarithmic loss" is defined as: **MOAC [- log (p_c)]**

where **MOAC** stands for the mean (average) over all cases (i.e., all cases for which the case file provides a value for the node in question), and

where **log(p_c)** is the natural logarithm of the probability predicted for the state that turns out to be correct.

Values for logarithmic loss vary from 0 to infinity (inclusive), with 0 being a perfect score. If you must use a single number to grade the predictive/diagnostic quality of a net with respect to a certain discrete node, then we recommend the logarithmic loss.

node is required to have been in the *test_nodes* list originally passed to NewNetTester_bn.

Version:

Versions 2.08 and later have this function.

See also:

GetTestConfusion_bn	Get elements of the confusion matrix
GetTestErrorRate_bn	Get the fraction of test cases for which the prediction failed
GetTestQuadraticLoss_bn	Get the "quadratic loss" score of the test
NewNetTester_bn	Construct the `tester_bn` object
GetMutualInfo_bn	Find the mutual info (entropy reduction) between two nodes

Example:

See NewNetTester_bn.

GetTestQuadraticLoss_bn

```
double GetTestQuadraticLoss_bn (tester_bn* test, node_bn* node)
```

Returns the "quadratic loss" of *node* under the tests previously performed.

The "quadratic loss" (also known as "Brier score") is defined as: **MOAC [1 - 2 * p_c + sum[j=1 to n] (p_j^2)]**

where **MOAC** stands for the mean (average) over all cases (i.e., all cases for which the case file provides a value for the node in question),

where $\mathbf{p_c}$ is the probability predicted for the state that turns out to be correct,

where $\mathbf{p_j}$ is the probability predicted for state j, and

where **n** is the number of states of the node.

Values for quadratic loss vary from 0 to 2, with 0 being a perfect score.

node is required to have been in the *test_nodes* list originally passed to NewNetTester_bn.

Version:

Versions 2.08 and later have this function.

See also:

GetTestConfusion_bn	Get elements of the confusion matrix
GetTestErrorRate_bn	Get the fraction of test cases for which the prediction failed
GetTestLogLoss_bn	Get the "logarithmic loss" score of the test
NewNetTester_bn	Construct the `tester_bn` object

GetVarianceOfReal_bn

```
double GetVarianceOfReal_bn (sensv_bn* sens, const node_bn* Vnode)
```

Measures how much a finding at one node (called the "varying node") is expected to reduce the variance of another node (called the "query node").

The query node is set by the particular `sensv_bn` created (see NewSensvToFinding_bn). The varying node is passed as *Vnode*.

GetVarianceOfReal_bn can only be used with query nodes that are discretized continuous nodes, or that have a real numeric value associated with each state. It measures the expected change squared in the expected real value of the query node, due to a finding at the varying node. This turns out to be the same as the expected decrease in variance of the expected real value of the query node, due to a finding at the varying node. The varying nodes do not have to be continuous or have real numeric values attached to their states.

The maximum possible decrease in variance of the query node, is when variance goes to zero, i.e., all uncertainty is removed. That happens when a finding is obtained for the query node itself. So to find the variance of a node, measure the variance reduction between a node and itself.

To create a sensv_bn that can measure variance reduction, pass REAL_SENSV + VARIANCE_SENSV for *what_find* when calling NewSensvToFinding_bn. For its *Vnodes* argument, pass a list of all the nodes that might later be passed as *Vnode* to this function.

The variance reduction between two nodes can depend greatly on what findings are entered elsewhere in the net, and this function will properly take that into account.

The first time this function is called by some sensv_bn after the findings of a net have changed, it takes longer to return, but after that, for each *Vnode* passed, it returns quickly.

This function is available as "Network -> Sensitivity to Finding" in Netica Application. For more information on it, contact Norsys for the "Sensitivity" document.

Version:

Versions 2.03 and later have this function. In versions previous to 3.05, this function was named **VarianceOfReal_bn**.

See also:

NewSensvToFinding_bn	Create the sensv_bn required to measure variance reduction due to finding
GetMutualInfo_bn	Use a different measure of sensitivity: mutual info (entropy reduction)

HasNodeTable_bn

```
bool_ns HasNodeTable_bn (const node_bn* node, bool_ns* complete)
```

Returns TRUE if *node* has a function table or a CPT table, otherwise FALSE.

It ignores experience tables.

If *complete* is non-NULL, it is set to indicate whether *node* has a complete table (i.e., none of the entries are undefined).

Version:

In version 2.06 and earlier, this function was named **HasRelation_bn** (but it didn't have the *complete* parameter).

See also:

SetNodeProbs_bn	Create a probabilistic table (CPT)
SetNodeFuncState_bn	Create a function table for a discrete node
SetNodeFuncReal_bn	Create a function table for a continuous node
DeleteNodeTables_bn	Remove all the tables of a node

IndexOfNodeInList_bn

```
int IndexOfNodeInList_bn (const node_bn* node,
                          const nodelist_bn* nodes,
                          int start_index)
```

Returns the position (index) of *node* in the list *nodes*, or -1 if it was not found.

It starts searching at index *start_index*; pass 0 to search the whole list.

It is okay if *node* is NULL.

Version:

Versions 2.11 and later have this function.

See also:

NthNode_bn	(inverse function) Returns the index, given the node

GetNodeNamed_bn	Search for node by name
LengthNodeList_bn	Returns the maximum node index

Example:

The following function is available in `NeticaEx.c`:

```
// Like IndexOfNodeInList_bn, but generates an error if there is not exactly
//   one instance of node in list nodes.
//
int IndexOfNodeInList (const node_bn* node, const nodelist_bn* nodes){
    int i = IndexOfNodeInList_bn (node, nodes, 0);
    if (i == -1)
        NewError (env, 901, ERROR_ERR,
                  "IndexOfNodeInList: There is no node '%s' in the list",
                  node ? GetNodeName_bn (node) : "null");
    else {
        int second = IndexOfNodeInList_bn (node, nodes, i + 1);
        if (second != -1)
            NewError (env, 902, ERROR_ERR,
                      "IndexOfNodeInList: There is more than one instance of node '%s' in the
                       list",
                      node ? GetNodeName_bn (node) : "null");
    }
    return i;
}
```

Example 2:

See GetNodeNamed_bn example 'FindNodeNamed' to search a node list for a node with a given name.

InitNetica2_bn

`int InitNetica2_bn (environ_ns* env, char* mesg)`

This initializes the Netica system. Call it before any other Netica functions except NewNeticaEnviron_ns, GetNeticaVersion_bn, or one of the environment configuration functions, such as ArgumentChecking_ns.

`env` should point to an environment created by calling NewNeticaEnviron_ns.

`mesg` should be a pointer to a character array which is allocated at least `MESG_LEN_ns` characters long. A startup welcome message will be left in *`mesg`* if InitNetica2_bn is successful, or an error message if it isn't.

It will return 0 or greater on success, or a negative value on failure. If it fails, then no other Netica API functions should be called with *`env`* except CloseNetica_bn. Use the return value to check for an error, rather than the regular Netica error system (e.g., GetError_ns).

Version:

This function is available in all versions.

In versions previous to 2.26 this function was named **InitNetica_bn** and took the address of a pointer to an environ_ns structure instead of just the pointer to the environ_ns.

See also:

NewNeticaEnviron_ns	Creates the required `environ_ns` object
CloseNetica_bn	Reverses the effects of InitNetica2_bn

Example:

```
int main (void){
    char mesg[MESG_LEN_ns];
    environ_ns* env;
    int res;

    env = NewNeticaEnviron_ns (NULL, NULL, NULL);    // substitute your
                                                     license string for the
                                                     first NULL, if desired
    res = InitNetica2_bn (env, mesg);
    printf ("%s\n", mesg);
    if (res < 0)  exit (-1);
    ....
```

```
    .... [rest of program]
    ....
    res = CloseNetica_bn (env, mesg);
    printf ("%s\n", mesg);
    exit (res < 0 ? -1: 0);
}
```

InsertFindingsIntoDB_bn

```
void InsertFindingsIntoDB_bn (dbmgr_cs* dbmgr, nodelist_bn* nodes,
                              const char* column_names,
                              const char* tables, const char* control)
```

Takes the net's current findings and inserts them into the database managed by *dbmgr*.

This function corresponds to the basic SQL[1] INSERT statement:

INSERT INTO table1 (col1,col2,...,colN) VALUES (value1,value2,...,valueN) .

nodes represents the nodes (node1,...,nodeN) whose current finding values (value1,...,valueN) will be inserted.

column_names is a comma-delimited list of database column names. The names in this list must be in the exact same order as their corresponding nodes in *nodes*. If *column_names* is NULL, then for each Node, Netica will use that Node's title (or, if the title is not defined, then the name) as the corresponding column name.

tables is a comma-delimited list of database table names. If the database has only one conventional (non-system) table, then you can submit NULL for this parameter and Netica will find the implied table for you.

Thus, for the SQL command INSERT INTO table1 (col1,col2,...,colN) VALUES (value1,value2,...,valueN), *tables* should be "table1"; *column_names* should be "col1,col2,...,colN"; and *nodes* should be a list of nodes in the order node1, node2, ..., nodeN.

Pass NULL for *control*; it is only for future expansion.

What value is inserted? If a node does not have a value then "NULL" is used for the value inserted. For most databases, this has the result of inserting a "Missing Data" value, although check with your database vendor regarding this (e.g., MS-ACCESS-2000 generally follows this rule but will insert "false" in a boolean field, instead of "Missing Data"). Otherwise, if a node does have a value, then the behavior varies, depending on whether the node is discrete or continuous. For **discrete** nodes, if that node has a state title, then that state title is inserted. If it does not have a title, but it has a name, then the name is inserted. And if it has neither title nor name, then the numeric state index (0... nStates-1) is inserted. For **continuous** nodes, the number inserted is the same as that returned by GetNodeValueEntered_bn.

If there is a problem with the SQL INSERT command, an error will be generated explaining the nature of the problem.

[1] SQL is a standard query language for accessing databases. To properly use this function, you should have basic familiarity with the SQL INSERT statement.

Version:

Versions 2.26 and later have this function. In versions previous to 3.05, this function was named **FindingsToDB_bn**.

See also:

NewDBManager_cs	Creates the dbmgr_cs
ExecuteDBSql_cs	Execute an arbitrary SQL command
AddDBCasesToCaseset_cs	Retrieve a set of cases using SQL SELECT
AddNodesFromDB_bn	Add nodes to a net using cases from SQL SELECT

Example:

```
dbmgr_cs *dbmgr = NewDBManager_cs (
   "driver={Microsoft Access Driver (*.mdb)}; dbq=.\\myDB.mdb;UID=dba1;", "pooling", env);
net_bn*  net           = NewNet_bn ("odbcTestNet", env);
node_bn* sexNode       = NewNode_bn ("sex",       2, net );
node_bn* heightNode    = NewNode_bn ("height",    0, net );
node_bn* ownsHouseNode = NewNode_bn ("ownsHouse", 2, net );
node_bn* numDogsNode   = NewNode_bn ("numDogs",   0, net );
```

```
nodelist_bn* nodes;
SetNodeStateName_bn (sexNode, 0, "M");
SetNodeStateName_bn (sexNode, 1, "F");
nodes = (nodelist_bn*) GetNetNodes_bn (net);
EnterFinding_bn (sexNode, GetStateNamed_bn ("M",sexNode));
EnterNodeValue_bn ( heightNode, 2.222 );
EnterFinding_bn (ownsHouseNode,1);
//Insert the net's current findings into Table1
InsertFindingsIntoDB_bn (dbmgr,
                         nodes,
                         "Sex, Height, \"Owns a house\", \"Number of dogs\"",
                         "Table1",
                         NULL);
```

IsBeliefUpdated_bn

bool_ns IsBeliefUpdated_bn (const node_bn* node)

Returns TRUE if belief updating (i.e., computing the posterior probability of node) has been done so that the beliefs at *node* are consistent with the current state of the net and current findings entered into the net, otherwise it returns FALSE.

The main use of this function is to determine if Netica will do belief propagation (which may be time consuming) the next time GetNodeBeliefs_bn or GetNodeExpectedUtils_bn is called with this node.

Version:

This function is available in all versions.

See also:

GetNodeBeliefs_bn	Updates the beliefs (if they aren't already)

IsNodeDeterministic_bn

bool_ns IsNodeDeterministic_bn (const node_bn* node)

If this returns TRUE then *node* is a deterministic node, which means that: given values for its parents, its value is determined with certainty.

There is no API function to directly set whether a node is deterministic, but setting all its conditional probabilities (i.e., CPT entries) to 0 or 1 will make a node deterministic. Building its table just with SetNodeFuncState_bn or SetNodeFuncReal_bn also will. Note that a node with a deterministic equation can result in a non-deterministic CPT, due to uncertainties introduced in the discretization process.

Version:

This function is available in all versions.

See also:

HasNodeTable_bn	Determine if node has any table
SetNodeProbs_bn	To change whether a node is deterministic
GetNodeType_bn	To determine if a node is for a discrete or continuous variable
GetNodeKind_bn	To determine what kind of node it is

IsNodeInNodeset_bn

bool_ns IsNodeInNodeset_bn (const node_bn* node, const char* nodeset)

Returns whether *node* is a member of *nodeset*.

Returns FALSE if there is no node-set called *nodeset* in the net containing *node*.

Version:

Versions 3.22 and later have this function.

See also:

AddNodeToNodeset_bn	To add nodes
RemoveNodeFromNodeset_bn	To remove nodes
GetAllNodesets_bn	Returns string listing all node-sets defined

---

IsNodeRelated_bn

```
bool_ns IsNodeRelated_bn (const node_bn* node1, const char* relation,
                          const node_bn* node2)
```

Tests whether *node1* is in the relationship *relation* with *node2*.

Returns TRUE if-and-only-if the expression "*node1* is/is-a/is-in-the *relation* of/to/with *node2*" is true.

relation should be one of "parent", "child", "ancestor", "descendent", "connected", "markov_blanket", "d_connected".

Version:

Versions 3.05 and later have this function.

See also:

GetRelatedNodes_bn	Finds all nodes bearing relationship
GetRelatedNodesMult_bn	Same, but for all members in a nodelist
GetNodeParents_bn	Finds all parents of a node
GetNodeChildren_bn	Finds all children of a node
GetNetNodes_bn	Get all nodes in the net

Example:

```
//Test if node1 is a parent of node2
if (IsNodeRelated_bn (node1, "parent", node2)) ...
```

Example 2:

```
//Test if node1 is in the markov blanket of node2
if (IsNodeRelated_bn (node1, "markov_blanket", node2)) ...
```

JointProbability_bn

```
double JointProbability_bn (const nodelist_bn* nodes, const
                            state_bn* states)
```

Returns the joint probability that each node in *nodes* is in the corresponding state of *states*, given the findings currently entered in the Bayes net. The *states* array must provide a state for each node of *nodes*.

This function is designed to work fast when retrieving many joint probabilities from nodes that were put in the same clique (see below) during net compilation. The first time it is called it will take longer to return, but on subsequent calls it will return very fast if these conditions are met:

1. *nodes* is the same list for each call.
2. No calls to it with a different *nodes* list were made in between.
3. No new findings have been entered or retracted.
4. No change was made to the net requiring re-compilation.
5. Each of *nodes* was placed in the same clique during compiling.

If conditions 1 or 2 are violated, it will still be much faster than doing a new belief updating, but not as fast as if they aren't violated. If the other conditions are violated, then it will take the same time as 1 or 2 belief updatings.

All of *nodes* must come from the same Bayes net.

None of *nodes* should have a likelihood finding (but they may have other types of findings, and other nodes in the net may have likelihood findings).

You can be sure a set of nodes will be placed in the same clique if there is some "family" in the Bayes net which contains all of them.

A *family* consists of a node and its parents. The function FormCliqueWith (in the example below and in NeticaEx.c) can be used to ensure that all of *nodes* will be put in the same clique during the next compile.

Version:

Versions 1.18 and later have this function.
In versions previous to 2.10 this function was named **GetJointProb_bn**.

See also:

FindingsProbability_bn	Joint probability for current findings
GetNodeBeliefs_bn	Posterior probability for a single node
GetNodeProbs_bn	Gets CPT entries of a node

Example:

The following function is available in NeticaEx.c:

```
// Ensures that at the next compile all of nodes will be put in the same clique.
// It is useful for the JointProbability_bn function.
// It works by adding a dummy node with 1 state, and returning that node (or NULL if it
//    wasn't necessary to add one).
// Its effects can be completely undone by calling DeleteNode_bn on the node it returns.
//
node_bn* FormCliqueWith (const nodelist_bn* nodes){
    net_bn* net;
    node_bn* new_node;
    int i, num_nodes = LengthNodeList_bn (nodes);
    if (num_nodes <= 1)  return NULL;
    net = GetNodeNet_bn (NthNode_bn (nodes, 0));
    new_node = NewNode_bn (NULL, 1, net);
    for (i = 0;  i < num_nodes;  ++i)
        AddLink_bn (NthNode_bn (nodes, i), new_node);
    return new_node;
}
```

LearnCPTs_bn

```
void LearnCPTs_bn (learner_bn* learner, const nodelist_bn* nodes,
                                  const caseset_cs* cases, double degree)
```

Performs learning of CPT tables from data. For EM or gradient descent algorithms this is done until a termination condition is met.

learner is the learner object that performs the learning steps. Construct it with NewLearner_bn.

nodes is the list of nodes whose experience and conditional probability tables are to be updated by learning. They must all be from the same net. Other nodes in that net will not be modified.

cases is the set of cases to be used for learning.

degree is the frequency factor to apply to each case in the case set. It must be greater than zero. It gets multiplied by the "NumCases" (multiplicity number) which appears for each case in the file (if the number doesn't appear in the file, it is taken as 1).

When you create the learner_bn (see NewLearner_bn), you choose the algorithm you wish, which may be one of:

1. Counting Learning

This is traditional one-pass learning (see ReviseCPTsByFindings_bn) ... . It is the preferred learning method to use, if there are no hidden (also known as 'latent') nodes in the net and no missing values in the case data. If there are hidden variables, that is, variables for which you have no observations, but you suspect exist and can be useful for modeling your world, or if there are a substantial number of missing values in the case data, then the iterative learning algorithms may yield better results.
Because this learning method is not iterative, SetLearnerMaxIters_bn and SetLearnerMaxTol_bn have no affect on it.

2. EM Learning

EM learning optimizes the net's CPTs using the well known expectation maximization algorithm, in an attempt to maximize the probability of the data set given the net (i.e., minimize negative log likelihood of the data). If the nodes have CPT and experience tables before the learning starts, they will be considered as part of the data (properly weighted using the experience table), so the knowledge from the data set is combined with the knowledge already in the net. If you do not want this effect, be sure to delete the tables first (see DeleteNodeTables_bn). During EM learning, for each case in the case file, only the CPTs of nodes with findings and their ancestor nodes become modified, so only those nodes will have their experience tables incremented.

3. Gradient Descent Learning

Gradient descent learning works similar to EM learning, but it uses a very different algorithm internally. It uses a conjugate gradient descent to maximize the probability of the data, given the net, by adjusting the CPT table entries. Generally speaking, this algorithm converges faster than EM learning, but may be more susceptible to local maxima. It has similarities to the neural net back propagation algorithm.

After the Learner is created, you can set the termination conditions for it. For both EM learning and gradient descent learning, the two possible termination conditions are the maximum number of iterations of the whole batch of cases (see SetLearnerMaxIters_bn), and the minimum change in log likelihood from one pass through the batch to the next (see SetLearnerMaxTol_bn). Termination will occur when either of the two conditions are met. For Counting learning, there currently are no termination conditions to set.

Version:

Versions 2.26 and later have this function.

See also:

NewLearner_bn	Creates the learner
SetLearnerMaxIters_bn	Sets a learning termination parameter: the maximum number of batch iterations
SetLearnerMaxTol_bn	Sets a learning termination parameter: the minimum log likelihood increase
NewCaseset_cs	Creates the `caseset_cs`
ReviseCPTsByCaseFile_bn	Uses a different learning algorithm (better suited if there is little missing data)
DeleteNodeTables_bn	May want to do this before learning

Example:

See AddDBCasesToCaseset_cs

LengthNodeList_bn

```
int LengthNodeList_bn (const nodelist_bn* nodes)
```

Returns the number of nodes in the list *nodes*.

Remember that some of the 'nodes' may actually be `NULL` entries, if you set them to `NULL`, or you created the list using NewNodeList2_bn with nonzero length and never completely filled it. Netica functions that return node lists will never return a node list having `NULL` entries unless that possibility is specifically mentioned in the documentation for that function.

Version:

This function is available in all versions.

LimitMemoryUsage_ns

```
double LimitMemoryUsage_ns (double max_mem, environ_ns* env)
```

Call this function anytime to adjust the amount of memory that Netica is permitted to allocate for tables.

For *max_mem* pass the number of bytes allowed. For example, to limit memory usage to 100 Megabytes, pass 100e6.

If Netica does not have enough memory to complete an operation, it will gracefully limit its behavior, and generate an error of severity ERROR_ERR.

This function is especially useful when running on operating systems that will provide very large amounts of virtual memory. You may not want Netica to be allowed to claim such large amounts, since this can result in excessive hard-disk usage, and very slow behavior (also known as "system thrashing"). It is also useful in keeping Netica well-behaved, if it must share memory with other running programs.

The previous memory allocation limit is returned. If QUERY_ns is passed for *max_mem*, then the limit is returned without changing it.

Version:

Versions 2.15 and later have this function.
In versions previous to 2.26, this function was named **MaxMemoryUsage_ns**.

MapStateList_bn

```
void MapStateList_bn (const state_bn* src_states,
                      const nodelist_bn* src_nodes,
                      state_bn* dest_states,
                      const nodelist_bn* dest_nodes)
```

Puts into the *dest_states* array the same states that are in the *src_states* array, except in a different order.

The order of *src_states* is given by *src_nodes*, and the order of *dest_states* will be given by *dest_nodes*.

src_nodes may not contain duplicates, but *dest_nodes* may (the state values will be duplicated accordingly).Any *src_states* entries for nodes in *src_nodes* which don't appear in *dest_nodes* will be ignored. If there are nodes in *dest_nodes* that don't appear in *src_nodes*, then EVERY_STATE will be placed in the corresponding position of *dest_states*.

The idea is that each entry of *src_states* contains a value of the corresponding node in *src_nodes*, and now we want these values in the order given by *dest_nodes*.

Version:

Versions 1.18 and later have this function.
Some older versions had a function called **ReOrderStates_bn** which did the same as this one.

See also:

SetNodeProbs_bn	Requires a list of states in the correct order
GetNodeFuncState_bn	Also requires correctly ordered states
GetNodeParents_bn	For the above, the list of states must in parent order
GetNodeFinding_bn	Determine the current state finding of a node

NewCaseset_cs	For sets of node-value pairs
ReorderNodeStates_bn	For the states within a single node

Example:

```
// MapStateList_bn is equivalent to the below function, but it is much faster
// and it doesn't use the user-data pointers.

void MapStateList (const state_bn*  src_states, const nodelist_bn*  src_nodes,
                         state_bn* dest_states, const nodelist_bn* dest_nodes){
    int i, num_src = LengthNodeList_bn (src_nodes);
    int num_dest = LengthNodeList_bn (dest_nodes);
     state_bn every = EVERY_STATE;
    for (i = 0;  i < num_dest;  i++)
        SetNodeUserData_bn (NthNode_bn (dest_nodes, i), 0, (void*) &every);
    for (i = 0;  i < num_src;   i++)
        SetNodeUserData_bn (NthNode_bn (src_nodes, i), 0, (void*) &src_states[i]);
    for (i = 0;  i < num_dest;  i++)
        dest_states[i] = * (state_bn*) GetNodeUserData_bn (NthNode_bn (dest_nodes, i), 0);
}
```

MostProbableConfig_bn

```
void MostProbableConfig_bn (const nodelist_bn* nodes,
                                  state_bn* config, int nth)
```

Finds the most probable configuration, also known as the most probable explanation (MPE), for all the nodes in the net. This is the setting for each of the nodes with the highest overall joint probability, given the currently entered findings. Of course it is consistent with the findings.

For *nodes*, you must pass a list returned by GetNetNodes_bn.

Pass 0 for *nth*. It is only for future expansion.

For *config*, pass an array of state_bn at least as long as *nodes*. The initial contents of this array will be ignored.

Netica will fill the *config* array with the configuration of highest joint probability given the currently entered findings. Each element of *config* is the state for the corresponding node of *nodes* (i.e., they are in the same order, and have the same length).

The net must be compiled before calling this function.

After finding the most probable configuration, you can use JointProbability_bn to find its probability (see example below).

You can mix calls to this function with calls to GetNodeBeliefs_bn (which finds posterior probabilities).

This function does not work when likelihood findings are entered. In that case you must make child nodes corresponding to the observations, whose CPTs are the likelihoods, and enter a positive finding for them.

If you must have the MPE of a smaller set of nodes than all the nodes in the net, you can use AbsorbNodes_bn to remove the other nodes first.

Keep in mind that in the MPE, some nodes may be assigned states that are quite unlikely (i.e., the state won't be the one with the highest probability as returned by GetNodeBeliefs_bn). That may be necessary in order to achieve the highest overall joint probability, considering the assignment of states to the other nodes. Before using this function, consider carefully whether you really want the MPE, or rather just a list of the most probable state for each of the nodes.

The algorithm Netica uses to find the MPE is known as a "max propagation" in the junction tree.

Version:

Versions 1.07 and later have this function.

See also:

JointProbability_bn	Find the actual joint probability of a configuration
GetNodeBeliefs_bn	Can find the most probable state for a single node only
GetNetNodes_bn	Get the list of nodes

Example:

```
// The following puts the most probable configuration in config,
// and its probability in maxprob.
//
CompileNet_bn (net);
const nodelist_bn* allnodes = GetNetNodes_bn (net);
int num_nodes = LengthNodeList_bn (allnodes);
state_bn* config = new state_bn [num_nodes];
MostProbableConfig_bn (allnodes, config, 0);
double maxprob = JointProbability_bn (allnodes, config);
// ... use config and maxprob ...
delete config;
```

NewCaseset_cs

`caseset_cs* NewCaseset_cs (const char* name, environ_ns* env)`

Creates and returns a new `caseset_cs`, initially containing no cases.

`name` can be `NULL`, or a legal IDname (see IDname in the index), which means it must have `NAME_MAX_ns` (30) or fewer characters, all of which are letters, digits or underscores, and it must start with a letter.

`name` will be used in Netica error messages to identify the case-set, and in future versions of Netica it will have further uses.

Netica will make a copy of *`name`*; it won't modify or free the passed string.

Version:

Versions 2.26 and later have this function. In versions previous to 3.15 this function did not have the *`name`* parameter.

See also:

DeleteCaseset_cs	Release the resources (e.g., memory) used by the Caseset
AddFileToCaseset_cs	Add cases from text file
AddDBCasesToCaseset_cs	Add cases from a database
LearnCPTs_bn	Use the Caseset for batch learning

NewDBManager_cs

`dbmgr_cs* NewDBManager_cs (const char* connect_str,`
`const char* control, environ_ns* env)`

Creates and returns a new `dbmgr_cs` object, which manages all interaction with a database. The database must be ODBC compliant.

Note. This function is currently only available for Netica running on Microsoft Windows. For support on other platforms, please contact Norsys.

Connection String

The *`connect_str`* defines an ODBC data source that this database manager will communicate with. The *`connect_str`* is the standard ODBC resource definition string passed directly to the standard **SQLDriverConnect** ODBC command. The general syntactic form of a connection string is "param=value;param=value;...", where 'param' is not case sensitive. Here are some sample connection strings for some common types of database configuration:

Data Source	Connection String
A Microsoft Access database located on your hard disk at C:\MyProjectDir\myAccessDB.mdb	"driver={Microsoft Access Driver (*.mdb)}; dbq=C:\\MyProjectDir\\myAccessDB.mdb; UID=myUserAccount;PWD=myPassword"

A Microsoft Excel spreadsheet located on your hard disk at C:\MyProjectDir\myData.xls	"driver={Microsoft Excel Driver (*.xls)}; dbq=C:\\MyProjectDir\\myData.xls"
An arbitrary data source named "myDataSource" that is registered with the Windows ODBC Data Source Administrator[1]	"DSN=myDataSource;UID=myUserAccount;PWD=myPassword"
An ORACLE 8i data server running locally with database name "OraInstanceName"	"DRIVER={Microsoft ODBC for Oracle}; SERVER=OraInstanceName; UID=OraUser;PWD=OraPswd;"
A MySQL database called "myDB" running on a machine whose domain name address is "db1.abc.com" communicating via port 5432	"Driver={MySQL};Server=db1.abc.com; Port=5432;Option=131072;Stmt=;Database=myDB; Uid=myUsername;Pwd=myPassword"
A Microsoft SQL Server database called "myDB" running on a machine called "DB_server" on your Microsoft Network LAN	"Driver={SQL Server};Server=DB_server;DataBase=myDB"
A text file[2] located on your hard disk at C:\MyProjectDir\myFile.csv	"Driver={Microsoft Text Driver (*.txt; *.csv)}; Dbq=C:\MyProjectDir\;Extensions=asc,csv,tab,txt"

[1] To access the Windows ODBC Data Source Administrator, from the Start menu, select "Settings" then "Control Panel". Then for Windows 2000/XP, double-click on "Administrative Tools" and then "Data Sources (ODBC)", and for Windows 95/98/NT, double-click on "ODBC Sources".

[2] For text files, your SQL statements must use the file name as the TABLE name. For example: "SELECT * FROM myFile.csv". Furthermore, the first line of the text file is assumed to give the COLUMN names. If you prefer other options than these, then use the Windows ODBC Data Source Administrator[1] which has an excellent wizard for text-file databases.

If the database does not have user accounts, then you do not need to specify a UID and PWD.

There are a great many parameters that can be specified within *connect_str*. Some are generic and apply to all database vendors, like "DSN" and "UID", while others are vendor specific. See your database vendor's documentation and the documentation for **SQLDriverConnect** to see what other parameters may be available. A good on-line source of documentation for ODBC is available at http://www.microsoft.com/data/odbc/.

Hint: If you are having difficulty getting your connection string to work, then use the Windows ODBC Data Source Administrator to connect to the database, give it a data source name (DSN), "myName", and then use "DSN=myName" as the simplified connection string. The Administrator has powerful wizards to facilitate both finding the database and defining precisely how you wish to access it.

WARNING: Because the connection string may contain a UserID and Password to your database, it represents a security risk. You may want to take extra precautions by securing your source code, dynamically fetching the password from the user, or asking your database administrator to place extra restrictions on the database user account.

Connection Pooling

The ODBC 3.0 standard allows for the caching of database connections, also known as "connection pooling". Every ODBC call involves requesting a connection and releasing it when done. Making a connection is expensive and can often take longer than an actual query to the database. For this reason, you will typically want to enable connection pooling so that your connections are opened only once, and thereafter are taken from and released back to the pool, rather than being really fully initialized and released with each ODBC call. You may wish to disable connection pooling if there are only a limited number of connections available and your process must be a good citizen and share its database connectivity with other processes.

Pass "pooling" for *control* to enable connection pooling. Pass "no_pooling" to disable it. Other control parameters may be added in future.

Connections in the pool may expire after a time. Adjustments to this time limit and to other properties of the connection pool can be made sometimes via the connection string (see above), and sometimes via the ODBC Data Source Administrator control panel, depending on the ODBC driver that is available.

When the `dbmgr_cs` object is deleted (see DeleteDBManager_cs), any connections it may have are released.

Version:

Versions 2.26 and later have this function.

See also:

DeleteDBManager_cs	Discard the database manager
ExecuteDBSql_cs	Execute an arbitrary SQL command
InsertFindingsIntoDB_bn	Insert net findings using SQL INSERT
AddDBCasesToCaseset_cs	Retrieve a set of cases using SQL SELECT
AddNodesFromDB_bn	Add nodes to a net using cases from SQL SELECT

Example:

```
// Create a new table called "Table2" in our MS-Access database, myDb.mdb,
// and define four columns in the table.
dbmgr_cs *dbmgr = NewDBManager_cs (
   "driver={Microsoft Access Driver (*.mdb)};dbq=.\\myDB.mdb;UID=dba1;", "pooling", env);
ExecuteDBSql_cs (dbmgr, "CREATE TABLE Table2 (TxtFld1 CHAR(10),
                                              IntFld1 INTEGER,
                                              FloatFld1 FLOAT,
                                              DateFld1 date)",
                                              NULL);
```

NewError_ns

```
report_ns* NewError_ns (environ_ns* env, int number,
                        errseverity_ns severity,
                        const char* mesg)
```

Generates a new error report, with its error number being *`number`*, severity level *`severity`*, and error message *`mesg`*. The new report is returned and registered with environment *`env`*. There is no need to worry about deleting the report object created, since this will be done automatically when the error is removed with either ClearError_ns or ClearErrors_ns.

The Netica API communicates to you every error condition it detects via a `report_ns` registered with the environment. This function is provided as a convenience, so that you can make your own `report_ns` registered with the environment, which can later be recovered with GetError_ns, allowing for simpler and more consistent error handling when extending Netica API.

`severity` must be one of `NOTHING_ERR`, `REPORT_ERR`, `NOTICE_ERR`, `WARNING_ERR`, `ERROR_ERR`, `XXX_ERR` (for a description of these see ErrorSeverity_ns).

`number` is an error number of your choice, but it must be between 1 and 999, inclusive. Netica will never generate errors with these numbers on its own.

To distinguish between an error generated by this function, or internally by Netica, see ErrorCategory_ns (passing `FROM_DEVELOPER_CND`).

There is no need to delete the report object created, since this will be done automatically when the error is removed with either ClearError_ns or ClearErrors_ns.

Version:

In versions previous to 2.10, this function was named **ReportError_ns**.

See also:

ClearError_ns	(reverse operation) Removes an error report from the system
GetError_ns	Obtain the error report from the environment
ErrorMessage_ns	Retrieve the error message from the report
ErrorNumber_ns	Retrieve the error number from the report
ErrorSeverity_ns	Retrieve the severity level of the error from the report

Example:

The following function is available in `NeticaEx.c`:

```
/*
 * Like NewError_ns, but with printf style arguments for the message.
 * You must be careful that your error message length is limited,
 *   so that it doesn't run over the declared buffer size, which you may
 *   want to make a little bigger or smaller.
 * For an example of its use, see the code for the "GetNode" function, in NeticaEx.c.
 */
report_ns* NewError (environ_ns* env, int number, errseverity_ns sev, const char* mesg, ...){
    va_list ap;
    char buf[1000];
    va_start (ap, mesg);
    vsprintf (buf, mesg, ap);
    va_end (ap);
    return  NewError_ns (env, number, sev, buf);
}
```

NewFileStream_ns

```
stream_ns* NewFileStream_ns (const char* filename, environ_ns* env,
                             const char* access)
```

Returns a Norsys stream for the file with name *`filename`*.

This stream can then be passed as an argument to functions which read or write a file, to identify which file to read or write.

Pass `NULL` for *`access`*; it is only for future expansion.

When finished with the `stream_ns` created, delete it with DeleteStream_ns.

`filename` may contain a path to indicate in which directory the file is located. It should be a string in the format normally understood by the operating system currently being used (see the examples below).

`filename` does not have to indicate a file which already exists.

Netica won't modify or free the passed *`filename`* string.

Version:

In versions previous to 2.10, this function was named **FileNamed_ns**, and between versions 2.10 and 2.26 it was called **NewStreamFile_ns**.

See also:

DeleteStream_ns	Delete it when done
WriteNet_bn	Saves a net to a file with name specified by the passed stream
WriteNetFindings_bn	
ReadNetFindings_bn	
ReadNet_bn	Reads a net from the file identified by the passed stream

Example:

See WriteNet_bn

```
//
// Examples for UNIX / Linux
//
file = NewFileStream_ns ("temp3", env, NULL);
file = NewFileStream_ns ("/local/project1/configure.bn.txt", env, NULL);
file = NewFileStream_ns ("../nets/Umbrella.dne", env, NULL);
//
// Examples for MS Windows
//
file = NewFileStream_ns ("temp3", env, NULL);
file = NewFileStream_ns ("C:local\project1\configur.txt", env, NULL);
file = NewFileStream_ns ("..\nets\Umbrella.dne", env, NULL);
//
// Examples for MacOS
//
```

```
file = NewFileStream_ns ("temp3", env, NULL);
file = NewFileStream_ns ("local:project1:configure.bn.txt", env, NULL);
file = NewFileStream_ns ("::nets:Umbrella.dne", env, NULL);
```

NewLearner_bn

```
learner_bn* NewLearner_bn (learn_method_bn method, const char* info,
                           environ_ns* env)
```

Creates and returns a new `learner_bn` object for use in learning of CPTs from case data, and associates it with a given Netica environment.

After creating this object, you use it to set the learning parameters you want, and then you pass it to a learning function, such as LearnCPTs_bn, to actually perform the learning on some net using some data file.

Pass `NULL` for *`info`*; it is only for future expansion.

`method` must be one of `COUNTING_LEARNING`, `EM_LEARNING`, or `GRADIENT_DESCENT_LEARNING`. See LearnCPTs_bn for a description of how each learning algorithm operates.

Version:

Versions 2.26 and later have this function.

See also:

SetLearnerMaxIters_bn	Set the maximum number of iterations (if applicable) it will do when learning
SetLearnerMaxTol_bn	Set the maximum tolerance (if applicable) it will allow before termination
LearnCPTs_bn	Performs the learning
DeleteLearner_bn	Discard the `learner_bn`

Example:

See AddDBCasesToCaseset_cs

NewMemoryStream_ns

```
stream_ns* NewMemoryStream_ns (const char* name, environ_ns* env,
                               const char* access)
```

Returns a Norsys stream for reading and writing to buffers in memory, in the same format reading/writing is done to disk files.

`name` is a symbolic name for identifying the stream (e.g., in error messages), and providing information on the expected format of the buffer, in the same way that a file extension can be used to indicate the format of disk files.

The stream can be used for input or output. Use SetStreamContents_ns to have Netica read from a text buffer that you supply. Use GetStreamContents_ns to retrieve a buffer that Netica has written to.

`name` is not necessarily the name of a file on the OS filesystem. If the end of name consists of a dot and then a few characters, it will be interpreted like a filetype extension, which can be useful to Netica in deciding the data format of the stream. For example, writing a net to a memory stream with a ".dnet" or ".dne" extension will result in a text file Netica format ("DNET format"), whereas using a ".neta" extension would result in a binary Netica file.

Netica will make a copy of *`filename`*; it won't modify or free the passed string.

Pass `NULL` for *`access`*; it is only for future expansion.

When done with the `stream_ns` produced, call DeleteStream_ns.

Version:

Versions 2.26 and later have this function.

See also:

SetStreamContents_ns	Sets memory buffer
GetStreamContents_ns	Retrieves memory buffer
SetStreamPassword_ns	Prepare the stream to do encryption or decryption
DeleteStream_ns	Releases all resources (e.g., memory) used by this stream

Example:

```
long length;
stream_ns* strm = NewMemoryStream_ns ("myBuf.dne", env, NULL);
WriteNet_bn (net, strm);     // write a net into the memory buffer
const char* buf = GetStreamContents_ns (strm, &length);
// ... [examine and use the buffer contents] ...
DeleteStream_ns (strm);
```

Example 2:

```
stream_ns* strm = NewMemoryStream_ns ("myBuf.cas", env, NULL);
const char* casedata = "A B C \n yes no 1 \n yes yes 3 \n no no 1";
SetStreamContents_ns (strm, casedata, strlen (casedata), true);
ReviseCPTsByCaseFile_bn (strm, nodes, 0, 1.0);
DeleteStream_ns (strm);
```

NewNet_bn

```
net_bn* NewNet_bn (const char* name, environ_ns* env)
```

Creates and returns a new net, initially having no nodes (use NewNode_bn to add nodes). If just nature nodes are added to this net, it will be a Bayes net, while if any decision nodes are also added, it will be a decision net.

The name of the new net will be *name*. It must be a legal IDname (see IDname in the index), which means it must have NAME_MAX_ns (30) or fewer characters, all of which are letters, digits or underscores, and it must start with a letter.

Netica will make a copy of *name*; it won't modify or free the passed string.

When you are done with the net, you should pass it to DeleteNet_bn.

See also:

DeleteNet_bn	(reverse operation) Frees the memory used by a net
ReadNet_bn	Create a net, by reading it from a file
GetNthNet_bn	Retrieve a net already created
SetNetName_bn	Later change the name
SetNetTitle_bn	Label the net without the IDname restriction
GetNetName_bn	Retrieve name
NewNode_bn	Add nodes to the new net
SetNetAutoUpdate_bn	Good to set this immediately after creating net, since the default value varies between Netica versions

---

NewNeticaEnviron_ns

```
environ_ns* NewNeticaEnviron_ns (const char* license,
                                 environ_ns* subenv, const char* locn)
```

This is always the first function to call when using Netica API. It creates a new environ_ns structure that is suitable for passing to InitNetica2_bn.

For *license* pass the license string provided to you when you purchased Netica, or NULL if you don't have one. The behavior of the Netica system may be limited by the license you use.

Pass NULL (i.e., 0) for *subenv* and *locn*; they are just for future expansion.

Most applications will have only one global environment, and in that case the best approach is to make a global variable of type `environ_ns*`, set it to the value returned by this function, and then pass it to any function that requires it.

This function does not generate any error conditions. The subsequent call to InitNetica2_bn will report if anything went wrong.

Netica will make a copy of *`license`*; it won't modify or free the passed string.

Version:

Versions 2.09 and earlier had a function called **NewNeticaEnviron_bn** (_bn instead of _ns) instead, which didn't take the *`locn`* or *`subenv`* arguments.

See also:

InitNetica2_bn	Initialize the `environ_ns` created
CloseNetica_bn	Delete the `environ_ns` created
LimitMemoryUsage_ns	To limit the amount of memory that can be allocated for this `environ_ns`

Example:

See InitNetica2_bn

NewNetTester_bn

```
tester_bn* NewNetTester_bn (nodelist_bn* test_nodes,
                            nodelist_bn* unobsv_nodes, int tests)
```

Creates a `tester_bn` which is a tool for grading a Bayes net, using a set of real cases to see how well the predictions or diagnosis of the net match the actual cases. It is not for decision networks.

`test_nodes` are the nodes that the Bayes net will predict and get rated on. Their values in the case file are all hidden from the Bayes net (i.e., unobserved) whenever a case is read. For each such case, the Bayes net does a prediction and compares that prediction with the true value from the case file, accumulating statistics as it goes.

If *`unobsv_nodes`* is non-`NULL`, then the nodes it contains will also be unobserved. It is okay if it repeats nodes already in *`test_nodes`*.

Pass -1 for *`tests`*.

After creating the `tester_bn` object, you run the tests using TestWithCaseset_bn, and then read out the results of the tests with the GetTest... functions. When done, you discard the `tester_bn` with DeleteNetTester_bn.

IMPORTANT: Before calling TestWithCaseset_bn, you may want to call RetractNetFindings_bn to remove any findings entered, because otherwise those findings will be considered while testing each case in the file.

The same net-testing capability is available as "Cases -> Test With Cases" in Netica Application.

Version:

Versions 2.08 and later have this function.

See also:

TestWithCaseset_bn	Accumulate case data into the test
GetTestConfusion_bn	Get elements of the confusion matrix
GetTestErrorRate_bn	Get fraction of test cases where prediction failed
GetTestLogLoss_bn	Get the "logarithmic loss" score of the test
GetTestQuadraticLoss_bn	Get the "quadratic loss" score of the test
DeleteNetTester_bn	Free up tester and all its resources
NewNodeList2_bn	Create the node lists

Example:

```
net_bn* net = ReadNet_bn (NewFileStream_ns ("ChestClinic.dne", env, NULL), NO_VISUAL_INFO);
nodelist_bn* unobsv_nodes = NewNodeList2_bn (0, net);
nodelist_bn*   test_nodes = NewNodeList2_bn (0, net);
```

```
node_bn* test_node = GetNodeNamed_bn ("Cancer", net);
AddNodeToList_bn (test_node, test_nodes, LAST_ENTRY);
// Now make the unobserved nodes list out of other factors not known during diagnosis:
AddNodeToList_bn (GetNodeNamed_bn ("Tuberculosis", net), unobsv_nodes, LAST_ENTRY);
AddNodeToList_bn (GetNodeNamed_bn ("Bronchitis", net), unobsv_nodes, LAST_ENTRY);
AddNodeToList_bn (GetNodeNamed_bn ("TbOrCa", net), unobsv_nodes, LAST_ENTRY);
RetractNetFindings_bn (net);   // IMPORTANT: Otherwise any findings will be part of tests !!
CompileNet_bn (net);

tester_bn* tester = NewNetTester_bn (test_nodes, unobsv_nodes, -1);

stream_ns* casefile = NewFileStream_ns ("ChestClinic.cas", env, NULL);
caseset_cs* caseset = NewCaseset_cs ("ChestClinicCases", env);
AddFileToCaseset_cs (caseset, casefile, 1.0, NULL);
TestWithCaseset_bn (tester, caseset);

PrintConfusionMatrix (tester, test_node); // defined in example for GetTestConfusion_bn
printf ("Error rate = %f %\n", 100 * GetTestErrorRate_bn (tester, test_node));
printf ("Logarithmic loss = %f %\n", GetTestLogLoss_bn (tester, test_node));

DeleteNetTester_bn (tester);
DeleteCaseset_cs (caseset);
//================================================
Confusion matrix for Cancer:
        Present  Absent  Actual
        6        1       Present
        1        192     Absent

Error rate = 1 %

Logarithmic loss = 0.02794
```

NewNode_bn

node_bn* NewNode_bn (const char* name, int num_states, net_bn* net)

Creates and returns a new node for *net*.

If the node is for a discrete variable, pass the number of states it has for *num_states*.

If the node is for a variable which is continuous in the real world, pass 0 for *num_states*, even if you plan to later discretize it to a certain number of states (see SetNodeLevels_bn for more details).

name will be the name of the new node. It must be different from the names of all other nodes in *net* (by case-sensitive comparison), and it must be a legal IDname, which means it must have NAME_MAX_ns (30) or fewer characters, all of which are letters, digits or underscores, and it must start with a letter. If name is NULL, then Netica will pick a unique name for the node; you can discover what name was picked using GetNodeName_bn after the node is formed.

The node will start off as a nature node (kind = NATURE_NODE), but it may be changed by calling SetNodeKind_bn.

Netica will make a copy of name; it won't modify or free the passed string.

See also:

DeleteNode_bn	(reverse operation) Removes the node from its net and frees memory it uses
CopyNodes_bn	Creates nodes by duplicating them, even from another net
SetNodeKind_bn	Set what kind of node it is (nature, decision, utility, etc.)
SetNodeLevels_bn	The way to set the number of states if the node is for a continuous variable being discretized
SetNodeName_bn	Later change the name
SetNodeTitle_bn	Label the node without the IDname restriction
GetNodeType_bn	Determine if it was created as a continuous variable node
GetNodeNumberStates_bn	Retrieve *num_states*
GetNodeName_bn	Retrieve name
AddLink_bn	Link the new node with others

NewNet_bn	Create a net for adding nodes

Example:

See SetNodeLevels_bn for creating a discretized node for a continuous variable.

NewNodeList2_bn

```
nodelist_bn* NewNodeList2_bn (int length, const net_bn* net)
```

Creates and returns a new node list with *length* entries, each filled with NULL.

You can then fill the entries using SetNthNode_bn. A safer way to build node lists is to call this function with *length* = 0, and then add nodes to the end of the list by calling AddNodeToList_bn with an index of LAST_ENTRY.

For *net*, pass the net which contains the nodes that will later be placed in this list. Nodes from other nets will not be allowed.

Use DeleteNodeList_bn to free the list when you are done with it (do not try to use the Standard C/ C++ 'free' or 'delete').

Version:

Prior to version 3.15, this function was called **NewNodeList_bn** and took an *env* parameter instead of *net*.

See also:

DeleteNodeList_bn	(reverse operation) Free the new list made
DupNodeList_bn	Make a copy of an existing list
AddNodeToList_bn	Increases a list's length by adding a node
SetNthNode_bn	

NewSensvToFinding_bn

```
sensv_bn* NewSensvToFinding_bn (const node_bn* Qnode,
                                const nodelist_bn* Vnodes, int what_find)
```

Creates a sensitivity measuring object, which measures how much the beliefs at one node (called the "query node" or "target node") will change if a finding is entered at another node (called the "varying node"). This is sometimes called "utility-free value of information".

For *Qnode*, pass the query node, and for *Vnodes*, pass a list of all nodes that might later be used as varying nodes.

There are two different measures available: variance reduction and entropy reduction (i.e., mutual information).

For *what_find*, pass the bitwise-OR of which measures you want the created object capable of measuring. To measure variance reduction, pass REAL_SENSV + VARIANCE_SENSV, and to measure mutual information, pass ENTROPY_SENSV.

After the object is created, to measure variance reduction, pass the new object to GetVarianceOfReal_bn along with a particular findings node. To measure mutual information between two nodes, pass it to GetMutualInfo_bn.

When you are finished with the sensitivity object, free the resources it uses by calling DeleteSensvToFinding_bn.

Netica uses an efficient algorithm that takes the current findings into account, and requires only a few belief updatings no matter how many nodes are contained in *Vnodes*. When you request the first sensitivity measure of the query node relative to one of the varying nodes (by calling GetMutualInfo_bn or GetVarianceOfReal_bn), the belief updatings are done and the results cached for subsequent calls involving other varying nodes.

These functions are available as "Network -> Sensitivity to Finding" in Netica Application. For more information on them, contact Norsys for the "Sensitivity" document.

Version:

Versions 2.03 and later have this function.

See also:

DeleteSensvToFinding_bn	(reverse operation) Delete the sensv_bn when finished with it
GetVarianceOfReal_bn	Use the sensv_bn to find the variance reduction due to finding
GetMutualInfo_bn	Use the sensv_bn to find the mutual info (entropy reduction)

Example:

```
net_bn* net = ReadNet_bn (NewFileStream_ns ("ChestClinic.dne", env, NULL), NO_WINDOW);
sensv_bn* svCancer = NewSensvToFinding_bn (GetNodeNamed_bn ("Cancer", net),
                                           GetNetNodes_bn (net), ENTROPY_SENSV);
double mutinfo = GetMutualInfo_bn (svCancer, GetNodeNamed_bn ("Dyspnea", net));
double entropy = GetMutualInfo_bn (svCancer, GetNodeNamed_bn ("Cancer", net));
DeleteSensvToFinding_bn (svCancer);
```

NthNode_bn

node_bn* NthNode_bn (const nodelist_bn* nodes, int index)

Returns the node at position *index* within list *nodes*.

The numbering starts with 0, and goes to LengthNodeList_bn (*nodes*) - 1.

If *index* is less than 0, or greater than LengthNodeList_bn (*nodes*) - 1, NULL will be returned, and an error will be generated (unless it is LAST_ENTRY, which indicates the last node).

To recover the node by name, rather than index, see the suggestion in GetNodeNamed_bn.

For the inverse function (finding the index given the node) see the function IndexOfNodeInList_bn.

Version:

This function is available in all versions.

See also:

IndexOfNodeInList_bn	(inverse function) Return the index, given the node
SetNthNode_bn	Set the node at the given index
RemoveNthNode_bn	Remove the node from the list (also returns it)
LengthNodeList_bn	Find maximum node index

Example:

The following function is available in NeticaEx.c:

```
// Prints out the names of the nodes in the list passed to it.
// You may need to print a newline ('\n') before the writing appears.
//
#include <stdio.h>

void PrintNodeList (nodelist_bn* nodes){
    int i, numnodes = LengthNodeList_bn (nodes);
    for (i = 0;  i < numnodes;  ++i){
    if (i != 0)  printf (", ");
        printf ("%s", GetNodeName_bn (NthNode_bn (nodes, i)));
    }
}
```

ReadNet_bn

net_bn ReadNet_bn (stream_ns* file, int control)

Reads a net from *file*, and returns the new net read, or NULL if reading was impossible. Even if this function returns a non-NULL value, you should check if it generated any errors, since it may report on a problem but return

the best net it can..

If *control* is NO_VISUAL_INFO, then any information about the visual display of the net for use by the graphical editor is ignored. You may want to read the visual information even if you won't be displaying the net, so that when it is written to file again, the information will not be lost for Netica Application, in which case pass NO_WINDOW for *control*.

If there were findings entered when the net was written to file, they will be present after reading, so you may want to do a RetractNetFindings_bn right after reading the net.

The Net will be created in the same environ_ns as *file* is in.

Version:

This function is available in all versions. Versions previous to 2.27 could not read files in .neta format. Some other Netica API programming environments which have a visual display allow the *control* argument to be MINIMIZED_WINDOW or REGULAR_WINDOW.

See also:

NewFileStream_ns	Generates the required stream_ns from the file name
WriteNet_bn	Saves a net to file in a format understood by ReadNet_bn
GetNetFileName_bn	Later retrieve the name of the file that net was read from

Example:

See WriteNet_bn

ReadNetFindings_bn

```
void ReadNetFindings_bn (caseposn_bn* case_posn, stream_ns* file,
                         const nodelist_bn* nodes, long* ID_num,
                         double* freq)
```

Reads a set of findings (i.e., a case) from a file containing one or more cases.

The case file is an ascii text file with each case on one row, and the first row being the list of nodes as column headings. Each entry is separated by a comma, space or tab. Such a format is quite common; it can be produced by a spreadsheet program like Excel, or by the Netica function WriteNetFindings_bn.

It only reads findings into the nodes listed in *nodes*. Other nodes in the net will not have any new findings entered, even if findings for them appear in the file. All the nodes of *nodes* must be from the same net. It is okay if *nodes* contains some nodes not mentioned in the file. If *nodes* is empty, no new findings will be entered.

WARNING: It does not retract findings that are already in *nodes*, and will even generate errors if the findings in the file are inconsistent with findings already in *nodes*. So you probably want to call RetractNetFindings_bn first.

In general it reads from *file* the case at **case_posn*, or if **case_posn* is NEXT_CASE it reads the next case after the last one read from *file*, and sets **case_posn* to the position of the case it just read. In detail:

Called with:	File condition:	Action taken:
case_posn = NULL	- file has no cases	generates error
	- file has 1 or more cases	reads first case
*case_posn = FIRST_CASE	- file has no cases	returns with *case_posn = NO_MORE_CASES
	- otherwise	reads first case & sets *case_posn to it
*case_posn = NEXT_CASE	- all cases read	returns with *case_posn = NO_MORE_CASES
	- otherwise	reads next case & sets *case_posn to it
*case_posn = NO_MORE_CASES		generates error

*case_posn = case	- indicated case is in file	reads indicated case
	- indicated case isn't in file	generates error

Make sure *`case_posn` is initialized on entry. If you want to read cases by random access, *`case_posn` should be set to a value previously returned by WriteNetFindings_bn or ReadNetFindings_bn (not the case `ID_num`).

When reading multiple sequential cases from the same file using `NEXT_CASE`, the `stream_ns` structure keeps track of the current file position. So different parts of your program, or different threads, can read from the same file in an interleaved way without interference, provided they each have their own `stream_ns`. But each sequential series of reads must use a single `stream_ns` (so the example below wouldn't work if the ReadNetFindings_bn call was replaced with: ReadNetFindings_bn (..., NewFileStream_ns (filename, env), ...); because that would make a new `stream_ns` each time it was called).

If `ID_num` is non-`NULL`, then on return *`ID_num` will be set to the ID number of the case, or -1 if it doesn't have one. If `freq` is non-`NULL`, then on return *`freq` will be set to the frequency (i.e., multiplicity) of the case stored with that case, or 1.0 if it doesn't have one.

This function doesn't modify or free the `nodes` list passed to it.

Version:

This function is available in all versions.
In versions previous to 2.26, this function was named **ReadCase_bn**.

See also:

WriteNetFindings_bn	Save it so that ReadNetFindings_bn can read it back
RetractNetFindings_bn	You may want to call this before reading a case
GetNetNodes_bn	Usually use this for the nodes argument
NewFileStream_ns	To create the `stream_ns` for the file argument

Example:

```
// Usage of ReadNetFindings_bn usually follows a pattern like that below.
//
// This example is meant as a template for functions that scan through
// a case file.
//
stream_ns* casefile = NewFileStream_ns (filename, env, NULL); // create fresh local stream_ns
const nodelist_bn* all_nodes = GetNetNodes_bn (net);
caseposn_bn caseposn = FIRST_CASE;
while(1){
    RetractNetFindings_bn (net);                          // must retract old case first
    ReadNetFindings_bn (&caseposn, casefile, all_nodes, NULL, NULL);
    if (caseposn == NO_MORE_CASES)  break;
    if (GetError_ns (env, ERROR_ERR, NULL))  break;
    //  ... do stuff with the case now entered ...
    caseposn = NEXT_CASE;                          // set it back to NEXT_CASE each time
}
DeleteStream_ns (casefile);
```

RemoveNodeState_bn

`void RemoveNodeState_bn (node_bn* node, state_bn state)`

Removes `state` from the states of `node`, so that `node` has one fewer state.

This function is for discrete nodes only. It is not for continuous nodes, even if they have been discretized (use SetNodeLevels_bn instead).

The CPTable will be renormalized to account for the missing state. If the probability for the missing state was 1.0 anywhere in the table, an error will be generated.

WARNING: You may want to remove any finding for *node* before calling this function, since if *node* has a finding, and it is for the statebeing removed, an error will be generated. If it is for another state, then that state's index will be properly modified so that after the removal operation it will correspond to the same state as it did before.

Version:

Since version 3.

See also:

AddNodeStates_bn	Add one or more states instead
ReorderNodeStates_bn	Assign a new order to the states
GetStateNamed_bn	Retrieve the new indexes of the states
GetNodeNumberStates_bn	*state* must be between 0 and one less than this, inclusive
SetNodeLevels_bn	For continuous nodes
RetractNodeFindings_bn	May want to call this first

RemoveNthNode_bn

`node_bn* RemoveNthNode_bn (nodelist_bn* nodes, int index)`

Removes (and returns) the node at position *index* from the list *nodes*, making the list one shorter, and maintaining the order of the rest of the nodes.

index can range from zero (the first node) to LengthNodeList_bn(*nodes*) - 1 (the last node), or it can be LAST_ENTRY which also indicates the last node.

If *index* is outside these bounds, the list will not be changed and an error will be generated.

Removing nodes from the end of the list executes the fastest.

Version:

In versions previous to 2.10, INT_MAX was used instead of LAST_ENTRY.

See also:

AddNodeToList_bn	(reverse operation) Adds a node to the list, lengthening it
NthNode_bn	Get a node from the list without removing it
LengthNodeList_bn	Find maximum node index
DupNodeList_bn	To duplicate a list before modifying it

Example:

The following function is available in NeticaEx.c:

```
// Removes node from the list nodes.
// node must be in the list, and appear only once, or an error is generated.
//
void RemoveOneNodeFromList (node_bn* node, nodelist_bn* nodes){
    int i = IndexOfNodeInList (node, nodes);
    RemoveNthNode_bn (nodes, i);
}
```

Example 2:

The following function is available in NeticaEx.c:

```
// Removes the first occurrence of node from the list.
// If node doesn't appear in the list, it does nothing.
//
void RemoveNodeFromListIfThere (node_bn* node, nodelist_bn* nodes){
    int i = IndexOfNodeInList_bn (node, nodes, 0);
    if (i != -1) RemoveNthNode_bn (nodes, i);
}
```

Example 3:

The following function is available in NeticaEx.c:

```
// This achieves the same purpose as RemoveNthNode_bn.
// Since removing the last node is fastest, this will execute
// more quickly (for long lists), but the order won't be maintained.
//
void RemoveNthNodeFast (int index, nodelist_bn* nodes){
    node_bn* lastnode = RemoveNthNode_bn (nodes, LAST_ENTRY);
    SetNthNode_bn (nodes, index, lastnode);
}
```

RedoNetOper_bn

`int RedoNetOper_bn (net_bn* net, double to_when)`

Call this to redo an operation that was undone by UndoNetLastOper_bn.

After N calls of UndoNetLastOper_bn and then N calls of RedoNetOper_bn, the net will be in the same state as it was before the calls.

Returns 0 or greater if it succeeded, otherwise negative. The most common reason for failing is that all the operation that were undone have already been redone.

Pass -1 for *to_when*; it is only for future expansion.

Version:

Versions 3.22 and later have this function.

See also:

UndoNetLastOper_bn	Call this first

RemoveNodeFromNodeset_bn

`void RemoveNodeFromNodeset_bn (node_bn* node, const char* nodeset)`

Removes *node* from the node-set named *nodeset*.

It is okay if *node* isn't in *nodeset* when this is called (then no action is taken).

Version:

Versions 3.22 and later have this function.

See also:

AddNodeToNodeset_bn	(inverse operation) To add the nodes
IsNodeInNodeset_bn	Determines if a node is in a node-set
ReorderNodesets_bn	To change the priority order of a net's node-sets
GetAllNodesets_bn	Returns string listing all node-sets defined

---

ReorderNodesets_bn

`void ReorderNodesets_bn (net_bn* net, const char* nodeset_order, void* vis)`

This rearranges the priority order of the node-sets of *net*.

Any node-sets contained in the comma-separated string *nodeset_order* will become the highest priority, with the nodes earlier in that list being higher priority. The priority of nodes not mentioned in *nodeset_order* will not be modified.

The purpose of the node-set priority order is just to determine which node-set to use for coloring a node in Netica Application.

Pass NULL for *vis*; it is only for future expansion.

Version:

Versions 3.22 and later have this function.

See also:

AddNodeToNodeset_bn	To create node-sets
SetNodesetColor_bn	How the node-set is displayed in Netica Application
GetAllNodesets_bn	Returns string listing all node-sets defined

---

ReorderNodeStates_bn

```
void ReorderNodeStates_bn (node_bn* node, const state_bn* new_order)
```

Rearranges the order of the states so that state *i* is moved to position*new_order[i]*. The length of *new_order* must be the number of states of*node*, all its entries must be between 0 and numstates-1, and it must not contain any duplicates.

All relevant parts of *node* will be modified to reflect the change. State names, titles, and comments will be moved, and the tables (CPT, experience, and function) will be adusted.

This function is for discrete nodes only. It is not for continuousnodes, even if they have been discretized.

Version:

Since version 3.

See also:

AddNodeStates_bn	Adds one or more new states
RemoveNodeState_bn	Removes a single state
GetNodeNumberStates_bn	*new_order* must have this many elements
GetStateNamed_bn	Retrieve the new indexes of the states

ReportJunctionTree_bn

```
const char* ReportJunctionTree_bn (net_bn* net)
```

Returns a null terminated C string containing a report of the junction tree for *net*, similar to that produced by the Netica Application operation "Report -> Junction Tree".

The report consists of one line for each clique, consisting of the clique's index, a list of cliques (i.e., their indexes) which the clique is connected to, and finally a list of nodes in the clique. At the end is the total statespace size of all the cliques, then the total size (with sepsets) added, and finally the total size with sepsets reduced by simplifications due to any findings currently entered.

net must already be compiled before calling this.

Version:

Versions 2.10 and later have this function.

See also:

CompileNet_bn	Need to compile the net first
SizeCompiledNet_bn	Just gets the overall size of the junction tree

Example:

```
// Below is example output from ReportJunctionTree_bn for the "ChestClinic" Bayes net.

Clique  [Joined To]  Size   Member nodes (* means home)
 0  [1]                4  (*VisitAsia, *Tuberculosis)
 1  [0 2]              8  (Tuberculosis, Cancer, *TbOrCa)
 2  [1 3 4]            8  (Cancer, TbOrCa, Bronchitis)
 3  [2 5]              8  (TbOrCa, Bronchitis, *Dyspnea)
 4  [2]                8  (*Smoking, *Cancer, *Bronchitis)
 5  [3]                4  (*XRay, TbOrCa)
Sum of clique sizes = 40  (with sepsets = 56)
```

RetractNetFindings_bn

`void RetractNetFindings_bn (net_bn* net)`

Retracts all findings (i.e., the current case) from all the nodes in *net*, except "constant" nodes (use RetractNodeFindings_bn for that).

This includes positive findings (state and real value), negative findings, and likelihood findings.

If *net* does not have any findings, calling this will have no effect.

If the net is an auto-update net (see SetNetAutoUpdate_bn), then a belief updating will be done to reflect the removal of findings, before this function returns (otherwise it will just be done when needed).

Version:

In versions previous to 2.10 this function was named **RetractAllFindings_bn**.

See also:

RetractNodeFindings_bn	To remove the findings for just one node

---

RetractNodeFindings_bn

`void RetractNodeFindings_bn (node_bn* node)`

Retracts all findings from *node*.

This includes positive findings (state and real value), negative findings, and likelihood findings. It removes them from any kind of node, including "constant" nodes.

If *node* does not have any findings, calling this will have no effect.

If the net is an auto-update net (see SetNetAutoUpdate_bn), then a belief updating will be done to reflect the removal of findings, before this function returns (otherwise it will just be done when needed). If you are going to be retracting a finding for a node, and then entering a new one, sometimes very significant performance gains can be made by ensuring auto-updating is turned off during the retraction (see example of EnterFinding_bn).

Version:

This function is available in all versions.

See also:

RetractNetFindings_bn	To remove the findings from all nodes in the net
EnterFinding_bn	To enter a finding for a node
GetNodeFinding_bn	To determine if a node has a finding

ReverseLink_bn

```
void ReverseLink_bn (node_bn* parent, node_bn* child)
```

Reverses the link from *parent* to *child*, so that instead it goes from *child* to *parent*.

This is a special function which maintains the joint probability represented by the net, which means any subsequent inference will yield the same results. To do so, Netica may have to add or remove links which go to *parent* from the parents of *child*, or which go to *child* from the parents of *parent*. If this is not desired then use DeleteLink_bn, followed by an AddLink_bn in the reverse direction. That will change the overall joint probability, and even change the independence information represented by the net.

If links are added, the CPT tables may become very large, possibly resulting in slow behavior or an out-of-memory condition.

If it is not possible to do the reversal, an error will be generated, and the net will not be changed. Reasons it might not be possible include: reversing the link would create a directed cycle, the child or parent node is not a nature node, the link is a time-delay link, the link is disconnected, or the child node has some other disconnected link.

Version:

This function is available in all versions.

See also:

GetNodeParents_bn	See what links Netica has added or removed
DeleteLink_bn	Followed by AddLink_bn the other way, will result in a reversed link and a net with different independence information
DeleteNodeTables_bn	Avoid overly large CPT tables caused by the reversal
AbsorbNodes_bn	Removes nodes, also maintaining the overall joint probability
LimitMemoryUsage_ns	In case this function is consuming too much memory

ReviseCPTsByCaseFile_bn

```
void ReviseCPTsByCaseFile_bn (stream_ns* file,
                              const nodelist_bn* nodes, int updating,
                              double degree)
```

Reads a file of cases from *file* and uses them to revise the experience and conditional probability tables (CPT) of each node in *nodes*. This function does the same thing as ReviseCPTsByFindings_bn, for each of the cases in *file*, but is more efficient than multiple calls to ReviseCPTsByFindings_bn. See the description of ReviseCPTsByFindings_bn for more information on the arguments passed, and how this function revises the probabilities.

All the nodes of *nodes* must be in the same net.

Pass 0 for *updating*. It is only for future expansion.

It is okay if the case file has missing data, or has data on nodes not included in *nodes*, or even has data on nodes not in the net containing *nodes*. However the probabilities of a node are only modified by cases supplying a value for the node and for all of its parents.

Version:

In versions previous to 2.10, this function was named **CaseFileRevisesProbs_bn**.

In versions 2.11 through 2.14, this function was named **CaseFileRevisesCPTs_bn**.

See also:

ReviseCPTsByFindings_bn	Revise probabilities with a single case
LearnCPTs_bn	Revise probabilities with a `caseset_cs`

NewFileStream_ns	Create the stream
NewNodeList2_bn	Create the node list

ReviseCPTsByFindings_bn

```
void ReviseCPTsByFindings_bn (const nodelist_bn* nodes, int updating,
                              double degree)
```

The current case (i.e., findings entered) is used to revise each node's conditional probabilities. This is different from *belief updating*, which finds the beliefs for nodes (i.e., posterior probabilities), given conditional probability relations between them and the findings that have been entered. Instead, *revising* the probabilities changes the conditional probability tables (CPTs) between the nodes to account for the current case.

The first few times this is called for a node, the probabilities will change considerably, because the node has little experience, but after many cases have been entered, each new case will result in only a small change.

All the nodes of *nodes* must be in the same net.

Pass 0 for *updating*. It is only for future expansion.

degree indicates how the case should be weighted. The normal value for *degree* is 1. If a positive integer n is passed, it will have the same effect as calling this function n times to tally up n identical cases. If *degree* is 0, the call will have no effect. If the case is learned by calling with *degree* = 1, it can later be "unlearned" by calling with *degree* = -1.

In general, if it is called with *degree* = d at one point in time, and then with the same case and *degree* = c at another time, the overall effect will be the same as a single call with *degree* = d + c, even if there were many intervening calls with other cases and other degrees, and even if d or c or both are negative. If a call to FadeCPTable_bn was made in between, then d will be weighted by the degree passed to FadeCPTable_bn.

The order in which cases are presented has no effect.

If a node already has CPT and experience tables, this function uses the experience table to provide a "confidence" for each of the probabilities in the CPT table. The higher the experience of a probability, the less it will be altered. It is okay if a node starts with no CPT or experience tables, since then Netica will start it off with a uniform distribution having the minimum experience. However, when calling this function, a node cannot have a CPT table and no experience table, since then Netica will not know what confidence to assign the existing probabilities of the CPT table, and an error will be generated.

Version:

In versions previous to 2.10, this function was named **CaseRevisesProbs_bn**.

See also:

ReviseCPTsByCaseFile_bn	Batch version, more efficient than one at a time
FadeCPTable_bn	Use between calls to ReviseCPTsByFindings_bn when the world is changing during learning
NewNodeList2_bn	Create the node list

SetCaseFileDelimChar_ns

```
int SetCaseFileDelimChar_ns (int newchar, environ_ns* env)
```

Sets the symbol used to separate data fields in a case file being created by Netica.

For *newchar*, pass the ascii character code. It must be one of tab (9), space (32) or comma (44).

Whole cases are always separated by a line end (i.e., a carriage return, a newline, or both).

newchar will only be used by Netica for creating case files; while reading them it will understand any of the above choices.

It returns the old symbol being used for this purpose. If QUERY_ns is passed for *newchar*, then the old value is returned without changing it.

Version:

Versions 1.18 and later have this function.

See also:

SetMissingDataChar_ns	Set the character used to indicate missing data
WriteNetFindings_bn	The function that uses the file delimiter character

Example:

```
int old_delim = SetCaseFileDelimChar_ns (',', env);
int old_miss  = SetMissingDataChar_ns (0, env);   // 0 allowed only if delim char is comma
// ... WriteNetFindings_bn ...
SetCaseFileDelimChar_ns (old_delim, env);         // restore (only do if necessary)
SetMissingDataChar_ns (old_miss, env);
```

SetLearnerMaxIters_bn

```
int SetLearnerMaxIters_bn (learner_bn* learner, int max_iters)
```

Sets the maximum number of learning-step iterations (i.e., complete passes through the data) which will be done when *learner* is used, after which learning will be automatically terminated. This applies to EM_LEARNING and GRADIENT_DESCENT_LEARNING only, since they are iterative by nature. Learning by the COUNTING_LEARNING method is not affected by this function.

Learning may be terminated earlier, if it first reaches another limit, such as *learner*'s maximum tolerance limit (see SetLearnerMaxTol_bn).

max_iters must be greater than 0 (or QUERY_ns). The default is 1000.

It returns the previous value of this limit (always 1 for COUNTING_LEARNING). If QUERY_ns is passed for *max_iters*, it just returns the previous value without changing it.

Version:

Versions 2.26 and later have this function.

See also:

NewLearner_bn	Creates the learner_bn
SetLearnerMaxTol_bn	Sets another termination parameter
LearnCPTs_bn	Performs the learning using this parameter

SetLearnerMaxTol_bn

```
double SetLearnerMaxTol_bn (learner_bn* learner,
                            double log_likeli_tol)
```

Sets the tolerance for the minimum change in data log likelihood between consecutive passes through the data, as a termination condition for any learning to be done by *learner*. This applies to EM_LEARNING and GRADIENT_DESCENT_LEARNING only, since they are iterative by nature. Learning by the COUNTING_LEARNING method is not affected by this function.

When learning is performed, with each iteration (i.e., pass through the complete data set), the "log likelihood" of the data given the net is computed. The log likelihood is the per-case average of the negative of the logarithm of the probability of the case given the current Bayes net (structure + CPTs). When the difference between the computed

log-likelihoods for two consecutive passes falls below this tolerance, the algorithm is halted. So, the closer this tolerance is to zero, the longer the algorithm may take.

The algorithm may terminate earlier if another termination condition is met, such as the maximum number of iterations (see SetLearnerMaxIters_bn).

log_likeli_tol must be greater than 0.0 (or QUERY_ns). The default is 1.0e-5.

It returns the previous value of this limit. If QUERY_ns is passed for *log_likeli_tol*, it just returns the previous value without changing it.

Version:

Versions 2.26 and later have this function.

See also:

NewLearner_bn	Creates the learner_bn
SetLearnerMaxIters_bn	Sets another termination parameter
LearnCPTs_bn	Performs the learning using this parameter

SetMissingDataChar_ns

```
int SetMissingDataChar_ns (int newchar, environ_ns* env)
```

Sets the symbol to be used for indicating missing data fields in a case file created by Netica. Data is "missing" when Netica has to provide the value for a node, and that node doesn't have a finding entered.

For *newchar*, pass the ascii character code. It must be one of asterisk * (42), question mark ? (63), space (32) or absent (0). It cannot be space or absent unless the delimiter symbol is a comma (see SetCaseFileDelimChar_ns).

newchar will only be used by Netica for creating case files; while reading them it will understand any of the above choices.

It returns the old symbol being used for this purpose. If QUERY_ns is passed for *newchar*, then the old value is returned without changing it.

Version:

Versions 1.18 and later have this function.

See also:

SetCaseFileDelimChar_ns	Set the character used to separate data entries
WriteNetFindings_bn	The function that uses the missing data character

Example:

See SetCaseFileDelimChar_ns

SetNetAutoUpdate_bn

```
int SetNetAutoUpdate_bn (net_bn* net, int autoupdate)
```

Pass BELIEF_UPDATE for *autoupdate* to have the new beliefs of a compiled net calculated immediately whenever new findings are entered, or 0 to inhibit this (in which case they will be calculated when needed, e.g., by GetNodeBeliefs_bn). The old auto-update value of *net* is returned.

A reason for inhibiting automatic updating is because updating (also known as "propagation") is time and memory consuming, and you may want to enter many findings before doing it. However, an advantage to having updating done after each finding is entered, is that each new finding will be checked for consistency with the findings already entered.

If you are going to be retracting a finding for a node, and then entering a new one, sometimes very significant performance gains can be made by ensuring auto-updating is turned off during the retraction (see example of EnterFinding_bn).

If the net is auto-updating, and you make a call to a single Netica function which enters findings for several nodes at once (e.g., reading a case), then Netica will use just a single updating to account for them all.

If you are turning auto-updating on, and the net is compiled but not updated, then updating will be done before this function returns, which may be time consuming.

It is best to always set auto-updating one way or the other after creating a new net, since the default value may vary between Netica versions.

When a net is written to file, the auto-update value is included.

Version:

This function is available in all versions.
Versions previous to 2.11 expected the autoupdate argument to be 1 instead of `BELIEF_UPDATE`, and they didn't return anything.

See also:

GetNetAutoUpdate_bn	Retrieves value
CompileNet_bn	Auto-updating doesn't occur until net is compiled
GetNodeBeliefs_bn	Forces a belief update if one is required

Example:

See EnterFinding_bn for an example of saving and restoring auto-update.

SetNetComment_bn

`void SetNetComment_bn (net_bn* net, const char* comment)`

This associates the null terminated C character string *`comment`* with *`net`* to help document it.

The comment may contain anything, but is usually used to store such things as the origin of the net, its purpose or applicability, background information on the problem domain, a copyright notice, how to use the net, notes for future changes, etc. It is best if the comment consists only of that sort of descriptive information (and as ascii characters), in order to meet expectations in case you share this net with other people or Netica Application. If you wish to attach other data, use SetNetUserField_bn.

Information that pertains only to a particular node should not be placed here, but rather in that node's comment field.

To remove a comment, pass `NULL` or the empty string for *`comment`*.

Netica will make a copy of *`comment`*; it won't modify or free the passed string.

Version:

This function is available in all versions.

See also:

GetNetComment_bn	Retrieves value
SetNodeComment_bn	Set a comment for a particular node
SetNetUserField_bn	To attach other types of information, and have it saved to file with the net

Example:

```
// Put string addon at the end of the existing comment for net
//
const char* origcomment = GetNetComment_bn (net);
int origlength = strlen (origcomment);
char* comment = malloc (origlength + strlen (addon) + 1);
strcpy (comment, origcomment);
strcpy (comment + origlength, addon);
SetNetComment_bn (comment, net);
```

```
free (comment);                 // but don't free origcomment
```

SetNetElimOrder_bn

void SetNetElimOrder_bn (net_bn* net, const nodelist_bn* elim_order)

Associates the list of nodes *elim_order* with *net* to be used as its "elimination order" the next time *net* is compiled.

elim_order must include all the nodes of *net* without any duplication (except it should not include any nodes whose kind is UTILITY_NODE or CONSTANT_NODE). Alternately, *elim_order* can be NULL, in which case any elimination order currently associated with *net* will be removed.

The elimination order guides the process of triangulation during the compilation of *net*, and can effect both the time and memory efficiency of belief updating considerably.

If no elimination order is supplied, Netica finds one automatically as the first step of compiling. When a net is written to file, the elimination order is included. Whenever the structure of a net changes, Netica removes the existing elimination order.

Calling this function has no effect on the current compilation; it only takes action during the next compilation. It doesn't matter if the net is compiled or not when this function is called.

Netica will make a copy of *elim_order*; it won't modify or free the passed list.

Version:

This function is available in all versions.

See also:

GetNetElimOrder_bn	Retrieves the elimination order currently being used
CompileNet_bn	Do or redo the compilation to use the new elimination order
AddNodeToList_bn	Can use with NewNodeList2_bn to build the list
SizeCompiledNet_bn	See how good the current ordering is
ReportJunctionTree_bn	Analyze the effect of the current order

---

SetNetName_bn

void SetNetName_bn (net_bn* net, const char* name)

Changes the name of *net* to be *name*.

name must be a legal IDname (see IDname in the index), which means it must have NAME_MAX_ns (30) or fewer characters, all of which are letters, digits or underscores, and it must start with a letter.

Netica will make a copy of *name*; it won't modify or free the passed string.

Version:

This function is available in all versions.

See also:

GetNetName_bn	Retrieves value
SetNetTitle_bn	Doesn't have the restriction of an IDname
NewNet_bn	Gives the net its original name

SetNetTitle_bn

```
void SetNetTitle_bn (net_bn* net, const char* title)
```

Sets the title of *net* to *title*, which can be any C character string to be used for titling the net. There are no restrictions on its length or what characters it may contain (unlike the 'name' of the net).

It is advised not to put too much information in the title, since the 'comment' field is available for that.

To remove a net's title, pass NULL or the empty string for *title*.

Netica will make a copy of *title*; it won't modify or free the passed string.

Version:

This function is available in all versions.

See also:

GetNetTitle_bn	Retrieves value
SetNetName_bn	The short, restricted name
SetNetComment_bn	For longer descriptions
SetNodeTitle_bn	Set the title for a particular node

SetNetUserData_bn

```
void SetNetUserData_bn (net_bn* net, int kind, void* data)
```

Attaches to *net* the data pointed to by *data*. Only your program needs to be able to understand this data. It may point to whatever is desired, possibly a large structure with many fields. This information may later be recovered using GetNetUserData_bn.

Pass 0 for *kind*. It is only for future expansion.

Only one user pointer may be attached to *net* at a time. When *net* is written to file, this user data is not included. For user-defined field-by-field data that gets saved to file, see SetNetUserField_bn.

Likewise, when the net is duplicated (CopyNet_bn), this user data is **not** included.

Netica will not modify, free or duplicate the data, even if the net is freed.

Version:

This function is available in all versions.

See also:

GetNetUserData_bn	Retrieves value
SetNetUserField_bn	Attach information field-by-field, and have it saved to file
SetNodeUserData_bn	Attach a user pointer to a particular node

SetNetUserField_bn

```
void SetNetUserField_bn (net_bn* net, const char* name,
                         const void* data, int length, int kind)
```

This associates user-defined data with net on a field-by-field basis. When net is written to file, this data will be saved to the file with it, and will be available when the net is read back from file.

It works exactly like SetNodeUserField_bn; see that function for usage information.

Version:

Versions 2.00 and later have this function.

See also:

GetNetUserField_bn	Retrieves value, by its name
GetNetNthUserField_bn	Iterate through the user fields of this net
SetNetUserData_bn	To attach completely user-managed data (not saved to file)
SetNodeUserField_bn	Attach field-by-field data to a particular node

---

SetNodeComment_bn

```
void SetNodeComment_bn (node_bn* node, const char* comment)
```

This associates the null terminated C character string *comment* with *node* to help document it.

The comment may contain anything, but is usually used to store such things as information on the variable represented by the node, its real-world significance, the meaning of its states, how the relation with its parents was determined, notes for future changes, etc. It is best if the comment consists only of that sort of descriptive information (and as ascii characters), in order to meet expectations in case you share this net with other people or Netica Application. If you wish to attach other data, use SetNodeUserField_bn.

Information that pertains to the net as a whole should not be placed here, but rather in the net's comment field.

To remove a node's comment, pass `NULL` or the empty string for *comment*.

Netica will make a copy of *comment*; it won't modify or free the passed string.

Version:

This function is available in all versions.

See also:

GetNodeComment_bn	Retrieves value
SetNetComment_bn	Set a comment for the whole net
SetNodeUserField_bn	To attach other types of information, and have it saved to file with the net

Example:

To add to an existing comment, see SetNetComment_bn.

SetNodeEquation_bn

```
void SetNodeEquation_bn (node_bn* node, const char* eqn)
```

This associates the equation *eqn* (a null terminated C character string) as the equation of *node*.

The equation can be deterministic, so that it specifies a value for *node*, given values for its parents (i.e., it expresses *node* as a function of its parents). Or, it can be probabilistic, so that it provides a probability for each of *node*'s values (i.e., a probability distribution), as a function of its parents.

For information on Netica equations, see the "Equation" chapter of Netica Application's onscreen help.

WARNING: Setting a node's equation does not modify its CPT table (which is what is used for inference in a compiled net). To modify the CPT table so that it reflects the new equation, use EquationToTable_bn.

To remove a node's equation, pass `NULL` or the empty string for *eqn*.

Netica will make a copy of *eqn*; it won't modify or free the passed string.

There is no restriction on the length or complexity of the equation.

Version:

Versions 1.30 and later have this function.

See also:

GetNodeEquation_bn	Retrieves value
EquationToTable_bn	Required to convert the equation to a CPT table for inference

---

SetNodeExperience_bn

```
void SetNodeExperience_bn (node_bn* node,
                           const state_bn* parent_states,
                           double experience)
```

This is to associate a degree of experience with each belief vector of *node*'s conditional probability table (see the chapter on Learning Nets). It sets the amount of experience for the condition described by *parent_states* (which provides a value for each parent) to *experience*.

The order of the states in *parent_states* should match the order of the nodes in the list returned by GetNodeParents_bn (this will be the same order that parents were added using AddLink_bn). MapStateList_bn may be useful for that. *parent_states* can be NULL if *node* has no parents.

If any entry of *parent_states* is EVERY_STATE then it applies to all possible values of the corresponding parent (see SetNodeProbs_bn).

To cycle through all the possibilities of *parent_states*, see the NeticaEx function NextStates.

Version:

This function is available in all versions.
In versions 1.33 and earlier, "EVERY_STATE" was called "WILDCARD_STATE".

See also:

GetNodeExperience_bn	Retrieves values
ReviseCPTsByFindings_bn	Increments experience
ReviseCPTsByCaseFile_bn	Sets experience to measure the number of relevant cases
FadeCPTable_bn	Decreases experience, and smoothes the probabilities
SetNodeProbs_bn	Sets corresponding CPT table entry
MapStateList_bn	To create the state list passed in

SetNodeFuncReal_bn

```
void SetNodeFuncReal_bn (node_bn* node, const state_bn* parent_states,
                         double func_value)
```

This is for deterministic nodes that are continuous or have been given real levels (e.g., by SetNodeLevels_bn). Deterministic nodes can be expressed as a function of their parent nodes, and that function can be in the form of a table. The purpose of SetNodeFuncReal_bn is to build that table. It adds an entry to the table by telling Netica that when each parent has the state indicated in the vector *parent_states*, the value of *node* is *func_value*.

The order of the states in *parent_states* should match the order of the nodes in the list returned by GetNodeParents_bn (this will be the same order that parents were added using AddLink_bn). MapStateList_bn may be useful for that. *parent_states* can be NULL if *node* has no parents.

If any entry of *parent_states* is EVERY_STATE then it applies to all possible values of the corresponding parent (see SetNodeProbs_bn).

If node has many parents (i.e., the product of their number of states is large) then the function table will be large, and your system may run out of memory. You can use GetError_ns after calling this to see if the table was successfully built.

To cycle through all the possibilities of *parent_states*, see the NeticaEx function NextStates.

Version:

Versions 2.06 and earlier didn't have this function, but had one called **SetNodeFuncValue_bn**, which worked almost the same, but took both discrete and continuous nodes (i.e., combined this and SetNodeFuncState_bn).
In versions 1.33 and earlier, "EVERY_STATE" was called "WILDCARD_STATE".

See also:

GetNodeFuncReal_bn	Retrieves values
SetNodeFuncState_bn	Same, but builds state integer tables instead of real-valued tables
SetNodeProbs_bn	To use instead if *node* isn't deterministic
MapStateList_bn	To create the state list passed in

Example:

The following function is available in NeticaEx.c:

```
void SetNodeFuncReal (node_bn* node, double value, ...);
// The first example of SetNodeFuncState_bn can be adapted for real values
// by just passing a double instead of an int for value, and calling
// SetNodeFuncReal_bn instead of SetNodeFuncState_bn.
```

SetNodeFuncState_bn

```
void SetNodeFuncState_bn (node_bn* node,
                          const state_bn* parent_states,
                          int func_state)
```

For deterministic nodes that are discrete or discretized. Deterministic nodes can be expressed as a function of their parent nodes, and that function can be in the form of a table. The purpose of SetNodeFuncState_bn is to build that table. It adds an entry to the table by telling Netica that when each parent has the state indicated in the vector *parent_states*, the state of *node* is *func_state*.

The order of the states in *parent_states* should match the order of the nodes in the list returned by GetNodeParents_bn (this will be the same order that parents were added using AddLink_bn). MapStateList_bn may be useful for that. *parent_states* can be NULL if *node* has no parents.

If any entry of *parent_states* is EVERY_STATE then it applies to all possible values of the corresponding parent (see SetNodeProbs_bn).

If *node* has many parents (i.e., the product of their number of states is large) then the function table will be large, and your system may run out of memory. You can use GetError_ns after calling this to see if the table was successfully built.

To cycle through all the possibilities of *parent_states*, see the NeticaEx function NextStates.

Version:

Versions 2.06 and earlier didn't have this function, but had one called **SetNodeFuncValue_bn**, which worked almost the same, but took both discrete and continuous nodes (i.e., combined this and SetNodeFuncReal_bn).
In versions 1.33 and earlier, "EVERY_STATE" was called "WILDCARD_STATE".

See also:

GetNodeFuncState_bn	Retrieves values
SetNodeFuncReal_bn	Same, but builds real-valued tables instead of discrete tables
SetNodeProbs_bn	To use instead if *node* isn't deterministic

Example:

The following function is available in NeticaEx.c:

```
// This function is similar to SetNodeProbs; see the comment for it.
//

#include <stdarg.h>
#define ARR_SIZE 20

void SetNodeFuncState (node_bn* node, int value, ...){
    char* statename;
    state_bn parent_states[ARR_SIZE];
    const nodelist_bn* parents = GetNodeParents_bn (node);
    int pn, numparents = LengthNodeList_bn (parents);
    va_list ap;
    if (numparents > ARR_SIZE){
        NewError_ns (env, 0, XXX_ERR, "SetNodeFuncState: Array too small");
        return;
    }
    va_start (ap, value);
    for (pn = 0;  pn < numparents;  ++pn){
        statename = va_arg (ap, char*);
        if (statename[0] == '*')  parent_states[pn] = EVERY_STATE;
        else parent_states[pn] = GetStateNamed_bn (statename, NthNode_bn (parents, pn));
    }
    va_end (ap);
    SetNodeFuncState_bn (node, parent_states, value);
}
#undef ARR_SIZE
```

Example 2:

```
// This doesn't use SetNodeFuncState_bn, but it is useful for setting
// parentStates.
// It cycles through all possible configurations (i.e., elements of the cartesian
//   product) of states, odometer style, with the last state changing fastest.
// states is a list of node states, one for each node of nodes.
// It returns TRUE when all the configurations have been examined (i.e., when it
//   "rolls over" to all zeros again).
// Don't forget to initialize states before calling it the first time (usually
//   to all zeros).

bool_ns NextStates (state_bn* states, const nodelist_bn* nodes){
    int nn;
    for (nn = LengthNodeList_bn (nodes) - 1;  nn >= 0;  nn--){
        if (++states[nn] < GetNodeNumberStates_bn (NthNode_bn (nodes, nn)))
            return FALSE;
        states[nn] = 0;
    }
    return TRUE;
}
```

SetNodeInputName_bn

```
void SetNodeInputName_bn (node_bn* node, int input_index,
                          const char* input_name)
```

Names the link entering *node* from its *input_index*th parent to be *input_name*.

input_index corresponds to the ordering of the parents obtained by GetNodeParents_bn (with the first parent having *input_index* = 0). The reason that an index number is passed instead of the actual parent, is because the link may not have a parent node (i.e., it is "disconnected"), or there may be more than 1 link from the same parent to *node*.

input_name must be a legal IDname (see IDname in the index), which means it must have NAME_MAX_ns (30) or fewer characters, all of which are letters, digits or underscores, and it must start with a letter.

To remove the name of a link, pass NULL (not the empty string) for *input_name*.

input_name must be different from the name of any other links entering *node* (by case-sensitive comparison, and must be different from the names of any parents of *node* which are connected to *node* by an unnamed link.

Input names are used to document what each link means, local to the node, which is especially important if the link is disconnected, or if its parents are continuously being switched. They are also useful as local parameters in equations, instead of using the names of parent nodes, so the equation stays valid even if the parents change.

When links are first created, they are unnamed, and remain so until this function is called, or until they are disconnected from the parent node (in which case they take on the name of the parent). It is possible to name some of the links entering a node, and leave the rest unnamed. All disconnected links are named.

Netica will make a copy of *input_name*; it won't modify or free the passed string.

Version:

In versions 1.17 and earlier, this function was named **SetLinkName_bn**.

See also:

GetNodeInputName_bn	Retrieves value
GetInputNamed_bn	Retrieves the index given the name
GetNodeParents_bn	Gets the actual parents of the links (e.g., to find their names or determine their numbering)
SwitchNodeParent_bn	Connects the "input" to a parent node

SetNodeKind_bn

```
void SetNodeKind_bn (node_bn* node, nodekind_bn kind)
```

Sets whether *node* is a nature, decision, utility or constant node.

kind must be one of:

NATURE_NODE	Bayes nets are composed only of this type (and constant nodes) This is a "chance" or "deterministic" node of an influence diagram
DECISION_NODE	Indicates a variable that can be controlled This is a "decision" node of an influence diagram
UTILITY_NODE	A variable to maximize the expected value of This is a "value" node of an influence diagram
CONSTANT_NODE	A fixed parameter, useful as an equation constant When its value changes, equations should be reconverted to CPT tables, and maybe the net recompiled

Nodes of one kind can usually be transformed to another at any time, but certain transformations are not allowed. Calling SetNodeKind_bn with a disallowed transformation will result in no changes, and a suitable error report will be generated. An example of a disallowed transformation is a non-deterministic node being transformed into a utility node.

You cannot use SetNodeKind_bn to change a node to kind DISCONNECTED_NODE. Nodes of that kind are created automatically when SwitchNodeParent_bn is called with NULL for the new parent.

Version:

In versions 1.09 and earlier, CONSTANT_NODE was called ASSUME_NODE.

See also:

GetNodeKind_bn	Retrieves value
NewNode_bn	Sets whether node is for a discrete or continuous variable
SwitchNodeParent_bn	To change a node to kind DISCONNECTED_NODE

SetNodeLevels_bn

```
void SetNodeLevels_bn (node_bn* node, int num_states,
                       const level_bn* levels)
```

Sets the levels list of *node* to *levels*.

The levels list is a list of real numbers used by Netica to translate from a real value of a continuous node to a discrete state, or from a state of a discrete node to a real value. That way a continuous node can act discrete (called "discretization"), or a discrete node can provide real-valued numbers. 'level_bn' is just defined as a 'double'.

If the underlying variable is continuous, we may want to discretize it for some operations. For example, we may divide all possible masses of some object into 3 ranges: 0 to 0.1 kg, 0.1 to 10 kg, and greater than 10 kg. For that we would use the levels list: [0, 0.1, 10, INFINITY_ns].

Conversely, if it is discrete, we may want a mapping from its state (represented as an integer), to a measurable value. For example, a drill may have 3 speeds (2.5 rps, 5 rps and 10 rps) as well as off. We could use a 4-state discrete node to represent the drill speed, with the levels list [0, 2.5, 5, 10]. Or milk may be available in containers of size 0.375, l, and 2 liters.

Netica will make a copy of *levels*; it won't modify or free the passed array.

Since the usage of levels is a little different for each type of node, each is discussed separately:

***node* is continuous:** (GetNodeType_bn would return CONTINUOUS_TYPE)

For *num_states* pass the number of ranges to discretize the node into. It must be zero or greater (if it is zero, then *levels* must be NULL, and any current discretization will be removed).

levels must contain *num_states* + 1 entries, and must monotonically ascend or descend (it is okay to have adjacent entries equal to create point-valued "ranges").

The first and last entries of the levels list provide a bound on the lowest and highest values the node can take on, but they may be INFINITY_ns or -INFINITY_ns (note: do not use the INFINITY_ns macro until after InitNetica2_bn has been called).

Once *node* has been given the levels list, Netica can translate a continuous value *val* for the node to a discrete state *st*, by choosing *st* so that:

levels [*st*] <= *val* < levels [*st* + 1] (if levels ascends) or
levels [*st*] > *val* >= levels [*st* + 1] (if levels descend)

A discrete state *st* can also be translated to the range:

[levels[*st*], levels[*st* + 1]) (if levels ascends) or
[levels[*st* + 1], levels[*st*]) (if levels descend)

***node* is discrete:** (GetNodeType_bn would return DISCRETE_TYPE)

There must be one entry in *levels* for each state of *node*. The value passed for *num_states* must be the number of states of the node (i.e., the value returned by GetNodeNumberStates_bn). There is no constraint on the ordering of *levels*.

Once *node* has been given the levels list, Netica can convert a discrete state *st* to a real-valued number *val*, using:

val = levels [*st*]

A real-number value *val* can also be translated to a discrete state *st* by choosing *st* so that: *val* = levels [*st*]. If there is no such *st*, then a legal translation cannot be made, but sometimes you can request Netica to approximate by choosing *st* so that:

| *val* - levels [*st*] | is minimized.

Version:

This function is available in all versions.

See also:

GetNodeLevels_bn	Retrieves values
NewNode_bn	Must be called with *num_states* = 0 to make continuous node
EnterNodeValue_bn	Uses discretization to convert to state finding
EquationToTable_bn	Uses discretization to handle continuous values in the table
GetNodeExpectedValue_bn	Uses discretization or real values to calculate mean and standard deviation

Example:

```
// Here we make a continuous node and then discretize it into 3 states.
//
node_bn* node = NewNode_bn ("n1", 0, net);  // must pass 0 for num_states to create a node
                                               for a continuous variable
level_bn levels[4];                         // 1 more than the number of states
levels[0] = 0;                              // the first range is 0 to 0.1
levels[1] = 0.1;
levels[2] = 10;
levels[3] = INFINITY_ns;
SetNodeLevels_bn (node, 3, levels);      // discretizes to 3 states
SetNodeStateNames_bn (node, "low, medium, high");   // naming the states is optional
```

Example 2:

```
// Here we make a 3-state discrete node and then give it
// levels to provide real values to its children.
//
node_bn* node = NewNode_bn ("volt_switch", 3, net);  // discrete, with 3 states
level_bn levels[3];                      // 1 element for each state
levels[0] =  0.0;
levels[1] = -3.5;                        // state 1 gives -3.5
levels[2] =  5.5;
SetNodeLevels_bn (node, 3, levels);      // set the levels
SetNodeStateName_bn (node, "off, reverse, forward");   // naming the states is optional
```

SetNodeName_bn

`void SetNodeName_bn (node_bn* node, const char* name)`

Changes the name of *node* to be *name*.

name must be a legal IDname (see IDname in the index), which means it must have NAME_MAX_ns (30) or fewer characters, all of which are letters, digits or underscores, and it must start with a letter.

name must be different from the name of every other node already in the same net (by case-sensitive comparison).

Netica will make a copy of *name*; it won't modify or free the passed string.

Version:

This function is available in all versions.

See also:

GetNodeName_bn	Retrieves value
SetNodeTitle_bn	Not restricted by IDname criteria
NewNode_bn	Gives the node its original name

SetNodeProbs_bn

```
void SetNodeProbs_bn (node_bn* node, const state_bn* parent_states,
                      const prob_bn* probs)
```

The purpose of this function is to build the conditional probability table (CPT) of *node*, which provides a probability distribution over the states of *node* for each possible configuration of parent states (i.e., parent condition). Each call sets the conditional probabilities of *node* for the situation where its parents have the states indicated by the vector *parent_states*. The length of *parent_states* must be the number of parents of *node*, and each of its entries provides a state for the corresponding parent. The length of the *probs* array must be the number of states of *node*, each entry is a prob_bn (i.e. 'float'), and consist of the conditional probabilities:

P (*node* = state0 | parents take on *parent_states*)

P (*node* = state1 | parents take on *parent_states*)

...

P (*node* = stateN | parents take on *parent_states*)

The order of the states in *parent_states* should match the order of the nodes in the list returned by GetNodeParents_bn (this will be the same order that parents were added using AddLink_bn). MapStateList_bn may be useful for that. *parent_states* can be NULL if *node* has no parents.

If any entry of *parent_states* is EVERY_STATE then it applies to all possible values of the corresponding parent. More than one entry of *parent_states* may be EVERY_STATE, in which case all the probabilities of their cartesian product will be set to *probs*, as you would expect (e.g., see the MakeProbsUniform example below).

Netica will make a copy of the *probs* array; it won't modify or free the passed array.

If *node* has many parents (i.e., the product of their number of states is large) then the probability table will be large, and your system may run out of memory. You can use GetError_ns after one or more calls to SetNodeProbs_bn to see if there was a problem.

After changing a node's probabilities, its net must be (re)compiled before calling GetNodeBeliefs_bn on any node in the net (although a full recompile isn't necessary, so it will proceed very quickly).

To set all the conditional probabilities of *node* at once, pass NULL for *parent_states*.

To cycle through all the possibilities of *parent_states*, see the NeticaEx function NextStates.

Version:

This function is available in all versions.
In versions 1.33 and earlier, "EVERY_STATE" was called "WILDCARD_STATE".

See also:

GetNodeProbs_bn	Retrieve values
SetNodeFuncState_bn	Build the table for a deterministic node
SetNodeExperience_bn	Associate a degree of experience with the probabilities
ReviseCPTsByFindings_bn	Revise the probabilities using the case currently entered
ReviseCPTsByCaseFile_bn	Revise the probabilities using a file of cases
FadeCPTable_bn	Adjust the probabilities for a changing world
MapStateList_bn	To create the state list passed in

Example:

The following function is available in NeticaEx.c:

```
// Gives the passed node a uniform conditional probability distribution
// (i.e., all the probabilities the same).
//
void MakeProbsUniform (node_bn* node){
    int st, numstates = GetNodeNumberStates_bn (node);
    int pn, numparents = LengthNodeList_bn (GetNodeParents_bn (node));
    prob_bn* uniform = malloc (numstates * sizeof (prob_bn));
```

```
    state_bn* pstates = malloc (numparents * sizeof (state_bn));
    for (st = 0;  st < numstates;  ++st)  uniform[st] = 1.0 / numstates;
    for (pn = 0;  pn < numparents;  ++pn)  pstates[pn] = EVERY_STATE;
    SetNodeProbs_bn (node, pstates, uniform);
    free (uniform);  free (pstates);
}
```

Example 2:

The following function is available in `NeticaEx.c`:

```
/*_____________________________________________________________ SetNodeProbs
This function is meant to be a more convenient (but slower) version of
  SetNodeProbs_bn.  Its first argument is the node whose probabilities we are
  setting.  This is followed by the names of the conditioning states of its
  parents as C strings.  Finally comes a list of doubles, being the probabilities
  for each of the states of the node.
For example:   SetNodeProbs (Temperature, "Windy", "Low", 0.6, 0.3, 0.1);
  means that the probability that Temperature is in its first state given that
  its first parent is in state "Windy" and its second parent is in state "Low"
  is 0.6, the probability its in its second state is 0.3, and that its in its
  third state is 0.1.
Passing "*" for a state names means it applies to all values of the state.
Since the function prototype uses "...", you must be very careful to pass doubles
  for the probabilities (e.g., passing 0 instead of 0.0 will get you in trouble).
If time efficiency is critical, and you must set large probability tables,
  use SetNodeProbs_bn directly instead of this function.
__________*/

#include <stdarg.h>
#define ARR_SIZE 20

void SetNodeProbs (node_bn* node, ...){
    state_bn parent_states[ARR_SIZE];
    prob_bn probs[ARR_SIZE];
    char* statename;
    state_bn state, numstates = GetNodeNumberStates_bn (node);
    const nodelist_bn* parents = GetNodeParents_bn (node);
    int pn, numparents = LengthNodeList_bn (parents);
    va_list ap;
    if (numstates > ARR_SIZE || numparents > ARR_SIZE){
        NewError_ns (env, 0, XXX_ERR, "SetNodeProbs: Array size too small");
        return;
    }
    va_start (ap, node);
    for (pn = 0;  pn < numparents;  ++pn){
        statename = va_arg (ap, char*);
        if (statename[0] == '*')  parent_states[pn] = EVERY_STATE;
        else parent_states[pn] = GetStateNamed_bn (statename, NthNode_bn (parents, pn));
    }
    for (state = 0;  state < numstates;  ++state)
        probs[state] = (prob_bn) va_arg (ap, double);
    va_end (ap);
    SetNodeProbs_bn (node, parent_states, probs);
}

#undef ARR_SIZE
```

Example 3:

The following function is available in `NeticaEx.c`:

```
// Sets all the conditional probabilities of node based on the array probs.
// You could use this function in combination with GetNodeAllProbs (see GetNodeProbs_bn
// to temporarily save probability tables.
//

void SetNodeAllProbs (node_bn* node, const prob_bn* probs){
    int num_states  = GetNodeNumberStates_bn (node);
    int num_parents = LengthNodeList_bn (GetNodeParents_bn (node));
    state_bn* parent_states = calloc (num_parents, sizeof (state_bn));
    while (1){
        SetNodeProbs_bn (node, parent_states, probs);
        if (NextStates (parent_states, GetNodeParents_bn (node)))  break;
        probs += num_states;
        if (GetError_ns (env, ERROR_ERR, NULL))  break;
    }
    free (parent_states);
```

```
}
```

SetNodesetColor_bn

```
color_ns SetNodesetColor_bn (const char* nodeset, color_ns color,
                             net_bn* net, void* vis)
```

This sets the color of *nodeset* to *color* and returns the old color of *nodeset*.

To indicate that nodes should not be colored based on this node-set, but rather to go on to the next lower priority node-set to determine the color, pass -2 for *color* (which can also be a return value of this function).

To just return the existing color (without changing it), pass QUERY_ns for *color*.

Colors are represented by ints, with the most significant byte(s) being 0, and last 3 bytes being red, green, blue (similar to colors in HTML documents).

The purpose of color is only for display purposes in Netica Application. If the color display is not as you expect, perhaps it is due to the node-set priority order (see ReorderNodesets_bn).

Pass NULL for *vis*; it is only for future expansion.

Version:

Versions 3.22 and later have this function.

See also:

AddNodeToNodeset_bn	To create node-sets, and add nodes to them
IsNodeInNodeset_bn	Determines if a node is in a node-set
ReorderNodesets_bn	To change the priority order of a net's node-sets
GetAllNodesets_bn	Returns string listing all node-sets defined

---

SetNodeStateComment_bn

```
void SetNodeStateComment_bn (node_bn* node, state_bn state,
                             const char* comment)
```

Gives the comment *comment* to that state of node whose index is *state*.

state must be between 0 and one less than the number that would be returned by GetNodeNumberStates_bn, inclusive.

There is no restriction on the length of the comment, or on what characters it might contain. *node* may have some states commented, and others not.

Discretized continuous nodes may have their states commented, as well as regular discrete nodes.

Passing NULL for *comment* will remove the comment of that state only.

Netica will make a copy of *comment*; it won't modify or free the passed string.

Version:

Versions 2.26 and later have this function.

See also:

GetNodeStateComment_bn	Get the existing state comment if there is one
SetNodeStateTitle_bn	Sets the state's title
GetNodeNumberStates_bn	*state* must be between 0 and one less than this, inclusive

SetNodeStateName_bn

```
void SetNodeStateName_bn (node_bn* node, state_bn state,
                          const char* state_name)
```

Gives the name *state_name* to that state of *node* whose index is *state*.

state must be between 0 and one less than the number that would be returned by GetNodeNumberStates_bn, inclusive. The argument type is a 'state_bn', whichis just another name for an 'int', but used to indicate that the int stands for a state index.

state_name must be a legal IDname (see IDname in the index), which means it must have NAME_MAX_ns (30) or fewer characters, all of which are letters, digits or underscores, and it must start with a letter. No two states of a node may have the same name. To avoid these restrictions, you can give the states titles instead; see SetNodeStateTitle_bn.

It is not required that a node's states be named, but if you give names to some of the sates of a node, you should name them all.

It may be more convenient to set the names of all the states at the same time; for that, see SetNodeStateNames_bn.

To remove all the state names of a node, you should use SetNodeStateNames_bn (notice theplural), but the same will also be done by a single call to this function ifNULL is passed for *state_name*.

Discretized continuous nodes may have their states named, as well as regular discrete nodes.

Netica will make a copy of *state_name*; it won't modify or free the passed string.

Version:

This function is available in all versions.

See also:

GetNodeStateName_bn	Retrieves values
SetNodeStateNames_bn	Sets names of all states at the same time
SetNodeStateTitle_bn	Doesn't have the restrictions of a name
NewNode_bn	Sets the number of states to start with
GetNodeNumberStates_bn	*state* must be between 0 and one less than this, inclusive

SetNodeStateNames_bn

```
void SetNodeStateNames_bn (node_bn* node, const char* state_names)
```

Names the states of *node* using the list of names *state_names*.

The names must be separated by commas and/or whitespace (i.e., spaces, tabs or newlines). It is okay if there is an extra separator at the end. The number of names must be the number that would be returned by GetNodeNumberStates_bn.

Each name must be a legal IDname (see IDname in the index), which means it must have NAME_MAX_ns (30) or fewer characters, all of which are letters, digits or underscores, and it must start with a letter. No two states of a node may have the same name. To avoid these restrictions, you can give the states titles instead; see SetNodeStateTitle_bn.

It is not required that a node's states be named. If you pass NULL (not the empty string) for *state_names*, it will remove all the state names for *node*.

Discretized continuous nodes may have their states named, as well as regular discrete nodes.

Netica will copy from *state_names*; it won't modify or free the passed string.

Version:

Versions 2.10 and later have this function.

See also:

SetNodeStateName_bn	Sets name of one state at a time
GetNodeStateName_bn	Retrieves a single state name
SetNodeStateTitle_bn	Doesn't have the restrictions of a name
NewNode_bn	Sets the number of states to start with

Example:

```
// Here is how you would make a new node with the 2 states
// "true" and "false".
//
node_bn* node = NewNode_bn ("n1", 2, net);  // make new node with 2 states
SetNodeStateNames_bn (node, "true, false");
```

SetNodeStateTitle_bn

```
void SetNodeStateTitle_bn (node_bn* node, state_bn state,
                           const char* state_title)
```

Gives the title *state_title* to that state of node whose index is *state*.

state must be between 0 and one less than the number that would be returned by GetNodeNumberStates_bn, inclusive.

State titles provide an alternative to state names for labeling the states of a node. Unlike a state's name, there is no restriction on the length of the title, or on what characters it might contain. *node* may have some states titled, and others not.

Discretized continuous nodes may have their states titled, as well as regular discrete nodes.

Passing NULL for *state_title* will remove the title of that state only.

Netica will make a copy of *state_title*; it won't modify or free the passed string.

Version:

Versions 1.18 and later have this function.

See also:

GetNodeStateTitle_bn	Get the existing state title if there is one
SetNodeStateName_bn	
SetNodeStateComment_bn	Sets the state's comment
GetNodeNumberStates_bn	*state* must be between 0 and one less than this, inclusive

---

SetNodeTitle_bn

```
void SetNodeTitle_bn (node_bn* node, const char* title)
```

Sets the title of node to *title*, which can be any C character string to be used for titling the node. There are no restrictions on its length or what characters it may contain (unlike the 'name' of the node).

It is advised not to put too much information in the title, since the 'comment' field is available for that.

To remove a node's title, pass NULL or the empty string for *title*.

Netica will make a copy of *title*; it won't modify or free the passed string.

Version:

This function is available in all versions.

See also:

GetNodeTitle_bn	Retrieve value
SetNodeName_bn	The short, restricted name
SetNodeComment_bn	For longer descriptions
SetNetTitle_bn	Set the title for the whole net

SetNodeUserData_bn

```
void SetNodeUserData_bn (node_bn* node, int kind, void* data)
```

Attaches to *node* the data pointed to by *data*. Only your program needs to be able to understand this data. It may point to whatever is desired, possibly a large structure with many fields. This information may later be recovered using GetNodeUserData_bn.

Pass 0 for *kind*. It is only for future expansion.

Only one user pointer may be attached to *node* at a time. When the net is written to file, the user pointer data is not included. For user-defined field-by-field data that gets saved to file, see SetNodeUserField_bn.

Likewise, when a node is duplicated (e.g., CopyNodes_bn and CopyNet_bn), this user data is **not** included.

Netica will not modify, free or duplicate the data, even if the node is freed or duplicated (although the duplicated node will contain the same pointer).

Version:

This function is available in all versions.

See also:

GetNodeUserData_bn	Retrieve it
SetNodeUserField_bn	Attach information field-by-field, and have it saved to file
SetNetUserData_bn	Attach a user pointer to the whole net

SetNodeUserField_bn

```
void SetNodeUserField_bn (node_bn* node, const char* name,
                          const void* data, int length, int kind)
```

This associates user-defined data with *node* on a field-by-field basis using attribute-value pairs. When the net is written to file, this data will be saved in the file with it, and will be available when the net is read back from file.

For *name* pass the name of the field to be set. Field names may be any ascii C string which meets the requirements of an IDname (max 30 chars, alphanumeric, underscores okay). The number of different field names is only limited by available memory.

For *data* pass a pointer to the data to associate, and for *length* pass the number of bytes of data to save. When you later retrieve the data with GetNodeUserField_bn that function will return the same length, and a pointer to a byte-for-byte copy of the same data.

Pass 0 for *kind*. It is only for future expansion.

If you have already set a field with the same name, Netica will overwrite that. To remove a field, call this function passing `NULL` for *data*.

The data may be of any type, but if you wish your Bayes net files to be portable across different operating systems, or if people or other programs may directly read your Bayes net files, it is best for the data to be an ascii string. Netica Application can also read and set user fields if they are ascii strings (use the multi-purpose selector at the bottom of the node properties dialog box of version 2.00 or later). Some helpful functions to set user fields to integers, real numbers and strings are: SetNodeUserInt, SetNodeUserNumber and SetNodeUserString, which are

provided in NeticaEx.c, and in the examples below. See GetNodeUserField_bn for the matching functions that do retrieval.

SetNodeUserField_bn will just copy the data from the location pointed to by *data*, and will never modify it or try to free that memory.

All memory management for the internal representation of user-defined fields is managed by Netica. They are duplicated when nodes are duplicated, and freed when they are freed.

If you wish to associate user data with a node, and not have that data saved to file, use SetNodeUserData_bn instead.

Version:

Versions 2.00 and later have this function.

See also:

GetNodeUserField_bn	Retrieve it, by its name
GetNodeNthUserField_bn	Iterate through the user fields of this node
SetNodeUserData_bn	To attach completely user-managed data (not saved to file)
SetNetUserField_bn	Attach field-by-field data to the whole net

Example:

The following function is available in NeticaEx.c:

```
// To set a user field to an ascii string
//
#include <string.h>

void SetNodeUserString (node_bn* node, const char* fieldname, const char* str){
    SetNodeUserField_bn (node, fieldname, str, strlen (str), 0);
}
```

Example 2:

The following function is available in NeticaEx.c:

```
// To set a user field to an integer
//
#include <string.h>
#include <stdio.h>

void SetNodeUserInt (node_bn* node, const char* fieldname, int num){
    char buf[22];
    sprintf (buf, "%d", (int)num);
    SetNodeUserField_bn (node, fieldname, buf, strlen (buf), 0);
}
```

Example 3:

The following function is available in NeticaEx.c:

```
// To set a user field to a real number
//
#include <string.h>
#include <stdio.h>

void SetNodeUserNumber (node_bn* node, const char* fieldname, double num){
    char buf[65];
    sprintf (buf, "%g", (double)num);
    SetNodeUserField_bn (node, fieldname, buf, strlen (buf), 0);
}
```

SetNodeVisPosition_bn

```
void SetNodeVisPosition_bn (node_bn* node, void* vis, double x,
                            double y)
```

Moves *node* so that its center is at coordinates (x, y), for any visual display (e.g., in Netica Application).

This is useful when directly programming Netica Application (see NewNeticaEnviron_ns), or before writing a net to a file that will later be read by Netica Application.

Pass `NULL` for *`vis`*; it is only for future expansion.

Version:

Versions 2.07 and later have this function, and versions 1.15 to 2.06 have an equivalent function called **SetNodeCenter_bn** that took ints instead of doubles.

See also:

GetNodeVisPosition_bn	Retrieves coordinates
SetNodesetColor_bn	Sets color

SetNodeVisStyle_bn

```
void SetNodeVisStyle_bn (node_bn* node, void* vis, const char* style)
```

Sets the style of *`node`* for any visual display (e.g., in Netica Application).

This is useful when directly programming Netica Application (see NewNeticaEnviron_ns), or before writing a net to a file that will later be read by Netica Application.

`style` must be one of: "Default", "Absent", "Shape", "LabeledBox", "BeliefBars", "BeliefLine", or "Meter". In future, other qualifiers may be added to this style parameter (e.g., "LabeledBox,CornerRoundingRadius=3,LineThickness=2").

Note that setting the style does not guarantee that a particular display application will be able to display the node in that style. Some applications may be limited in their ability and may interpret a particular display style differently or even ignore it.

Pass `NULL` for *`vis`*; it is only for future expansion.

Version:

Versions 3.05 and later have this function.

See also:

GetNodeVisStyle_bn	Gets style
SetNodeVisPosition_bn	Sets coordinates

SetNthNode_bn

```
void SetNthNode_bn (nodelist_bn* nodes, int index, node_bn* node)
```

Puts *`node`* at position *`index`* of list *`nodes`* without changing the length of the list.

If *`index`* is 0 the node is put at the beginning of the list, and if it is LengthNodeList_bn(nodes) - 1, the node will be put at the end.

If *`index`* is outside of these bounds, nodes will not be modified, and an error will be generated.

Version:

This function is available in all versions.

See also:

AddNodeToList_bn	Insert a node in a list, increasing its length
NthNode_bn	Retrieve the node at the given index
LengthNodeList_bn	Find maximum node index for SetNthNode_bn
DupNodeList_bn	To duplicate a list before modifying it

SetStreamContents_ns

```
void SetStreamContents_ns (stream_ns* strm, const char* buffer,
                           long length, bool_ns* copy)
```

Sets *strm*'s memory buffer to be *buffer*, so that future reading from *strm* will take place from *buffer*.

strm must be a memory stream (created by a call to NewMemoryStream_ns) or an error will be generated.

buffer is a pointer (possibly NULL) to the new memory buffer. If you wish Netica to later interpret the contents of this buffer as an ascii string (e.g., if the buffer contains case data that will be parsed into a Caseset), then this buffer must be null terminated. Once passed into this function, *buffer* should not be freed or modified until either DeleteStream_ns is called on *strm*, or a new buffer is assigned to *strm* with this function (possibly NULL to empty it). Netica will not free or modify *buffer*, even when DeleteStream_ns is called.

length is the number of bytes in *buffer*, excluding any terminating null.

Normally you will pass TRUE for *copy*, so that Netica copies the string in *buffer* for its own use. However, if the string is very large, for efficiency you can pass FALSE, in which case you must ensure that the contents of *buffer* are not modified or deallocated until the stream_ns is destroyed or has its contents set to something else.

Version:

Versions 2.26 and later have this function. In versions previous to 3.05, this function was named **SetStreamBuffer_ns**.

See also:

NewMemoryStream_ns	Create new memory stream
GetStreamContents_ns	Retrieves buffer

Example:

See NewMemoryStream_ns.

Example 2:

See SetStreamPassword_ns.

SetStreamPassword_ns

```
void SetStreamPassword_ns (stream_ns* strm, const char* password)
```

Sets the password that Netica will use for either encrypting an output stream, or for decrypting an input stream.

Encryption/decryption is only possible for certain file formats (e.g., ".neta"). The file format is specified when the stream is created (see NewFileStream_ns or NewMemoryStream_ns). If *strm* is not for a format that allows encryption/decryption (such as .dne, .cas, .xml, or .txt), then an error will be generated.

If the password supplied for reading an encrypted source is not the same password that was used by Netica to encrypt that source, then an error will be generated when you attempt to read from that source.

Netica will make a copy of *password*; it won't modify or free the passed string.

Pass NULL for *password* to remove it, so that subsequent reading/writing from this stream will be without any encryption/decryption.

Version:

Versions 2.26 and later have this function.

See also:

NewFileStream_ns	Create new file stream

NewMemoryStream_ns	Create new memory stream

Example:

```
stream_ns* stream = NewMemoryStream_ns ("myNet.neta", env, NULL);
SetStreamPassword_ns (stream, "MyPassword123");
WriteNet_bn (net, stream);   // writes an encrypted file
long length;
const char* buf = GetStreamContents_ns (stream, &length);  // buf now holds the encrypted net

stream_ns* stream2 = NewMemoryStream_ns ("myNet.neta", env, NULL);
SetStreamContents_ns (stream2, buf, length);
SetStreamPassword_ns (stream2, "MyPassword123");
net_bn* net2 = ReadNet_bn (stream2, NO_VISUAL_INFO);   // reads the encrypted file

stream_ns* stream3 = NewMemoryStream_ns ("myNet.neta", env, NULL);
SetStreamContents_ns (stream3, buf, length);
SetStreamPassword_ns (stream3, "WrongPassword456");
net_bn* net3 = ReadNet_bn (stream3, NO_VISUAL_INFO);  // generates error - password is wrong
```

Size Compiled Net_bn

```
double SizeCompiledNet_bn (net_bn* net, int method)
```

Returns the total size of the internal structure created by compiling a net (i.e., the junction tree, including sepsets), considering the findings currently entered. The size is measured as the number of state space entries (i.e., the number of probabilities that must be stored).

Pass 0 for *method*. It is only for future expansion.

net must already be compiled before calling this (see CompileNet_bn).

Maximum inference time for belief updating, and memory required for compiling and updating, are both linearly related to the quantity returned (the number of bytes required is 4 times the number returned). They are maximum, providing *net* does not have any positive findings entered which are later removed.

The value returned will be at its maximum before any findings are entered, and with each new positive finding entered, it will decrease or remain constant. Any likelihood or negative findings entered will not alter the value returned, unless they are equivalent to a positive finding.

Version:

Versions 2.06 and later have this function.

See also:

ReportJunctionTree_bn	Provides more information on junction tree
CompileNet_bn	Need to compile the net first
SetNetElimOrder_bn	Elimination order can have a major effect on the compiled size

---

Switch Node Parent_bn

```
void SwitchNodeParent_bn (int link_index, node_bn* node,
                          node_bn* new_parent)
```

Makes node *new_parent* a parent of *node* by replacing the existing parent at the *link_index*th position, without modifying *node*'s equation, or any of *node*'s tables (such as CPT table or function table).

The new parent must be compatible with the old (e.g., same number of states), or an explanatory error will be generated, and no action taken.

NULL can be passed for *new_parent*, in which case the corresponding link will not be removed, but will become disconnected. If that link was not already named, then its name will become the name of the parent it was disconnected from. To determine whether a link is disconnected, see GetNodeKind_bn.

If the link was disconnected, this function may be used to re-connect it, by passing non-NULL for *new_parent*.

The parents of *node* are numbered from 0 to one less than the number of parents, and the ordering can be obtained using GetNodeParents_bn. Sometimes it is more useful to be able to pass a parent node instead of *link_index*, if you know there is exactly one link from the parent node to *child*. This can be accomplished with the SwitchNodeParent example below.

Version:

This function is available in all versions.

See also:

GetNodeParents_bn	Can be used to determine a suitable value for *link_index*
AddLink_bn	Adds a link between two nodes
DeleteLink_bn	Removes a link between two nodes
GetNodeKind_bn	To determine if a link is disconnected (returns DISCONNECTED_NODE)

Example:

The following function is available in NeticaEx.c:

```
// Switches the link from parent -> child to go from new_parent -> child.
// Assumes there is already exactly one link from parent to child.
//
void SwitchNodeParent (node_bn* parent, node_bn* child, node_bn* new_parent){
    int link_index = IndexOfNodeInList (parent, GetNodeParents_bn (child));
    SwitchNodeParent_bn (link_index, child, new_parent);
}
```

TestWithCaseset_bn

void TestWithCaseset_bn (tester_bn* test, caseset_cs* cases)

Scans through the case data in *cases* to do a number of performance tests on a Bayes net (specified when creating the tester_bn).

Netica will pass through the caseset, processing the cases one-by-one. Netica first reads in the case, except for any findings for the unobserved nodes (specified when creating the tester_bn). It then does belief updating to generate beliefs for each of the unobserved 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 (which may be retrieved by the various GetTest... functions).

IMPORTANT: Before calling TestWithCaseset_bn, you may want to call RetractNetFindings_bn to remove any findings entered, because otherwise those findings will be considered while testing each case in the file.

The net must be compiled (see CompileNet_bn) before calling this.

This function can be called multiple times with different files to accumulate the results of all the cases.

Calls to this function can be intermingled with calls to GetTestConfusion_bn, GetTestErrorRate_bn, GetTestLogLoss_bn, and GetTestQuadraticLoss_bn.

This function will properly support a 'NumCases' column in any case file used to create the caseset, if such a column was present.

Version:

Versions 2.08 and later have this function. In versions previous to 3.15, this function was named **TestWithFile_bn**.

See also:

NewNetTester_bn	Construct the tester_bn object

Example:

See NewNetTester_bn.

UncompileNet_bn

```
void UncompileNet_bn (net_bn* net)
```

Releases the resources (e.g., memory) used by a compiled net.

It doesn't change the elimination ordering.

Calling UncompileNet_bn when the net is not compiled has no effect.

SizeCompiledNet_bn can be used to determine how much memory will be released.

Version:

Versions 2.09 and later have this function.

See also:

CompileNet_bn	(reverse operation)
SizeCompiledNet_bn	To determine how much memory will be released
DeleteNet_bn	Discard the whole net

UndoNetLastOper_bn

```
int UndoNetLastOper_bn (net_bn* net, double to_when)
```

Undoes the last operation done to a Bayes net (or any node in it), leaving the net in the same state as it was before the operation was done.

It may be called repeatedly to undo multiple operations.

Returns 0 or greater if it succeeded, otherwise negative. The most common reason for failing is that there were no (more) operations to undo.

Pass -1 for *to_when*; it is only for future expansion.

Version:

Versions 3.22 and later have this function.

See also:

RedoNetOper_bn	Re-does the operation just undone

WriteCaseset_cs

```
void WriteCaseset_cs (caseset_cs* cases, stream_ns* file,
                      const char* control)
```

Writes all the cases within the caseset *cases* to *file*.

They are written in the standard Netica case file format.

In future, *control* will allow you to control what gets copied. For now, pass NULL.

Version:

Versions 2.28 and later have this function.

See also:

AddFileToCaseset_cs	Reverse function.

NewCaseset_cs	Create a new Caseset.
DeleteCaseset_cs	Free the resources (e.g., memory) used by the Caseset.
LearnCPTs_bn	Use the Caseset for learning.

WriteNet_bn

```
void WriteNet_bn (const net_bn* net, stream_ns* file)
```

Writes *net* to a new file specified by *file*.

The file format that the net is written in depends on the file extension (i.e., the ending of the file name passed to NewFileStream_ns or NewMemoryStream_ns). If the extension is ".neta", a binary format producing much smaller files and allowing for encryption is used. Otherwise the DNET file format is used, which is a text file format that may be useful in examining/editing the files produced, or exporting them to another program (for more information, see http://www.norsys.com/dl/DNET_File_Format.txt). It is advised to end the file names with either ".neta" or ".dne", so that way they can be more easily identified by other people and other programs, such as Netica Application.

All versions of Netica API and Netica Application can read/write ".dne" files (which are the same as ".dnet" files), and all versions of them after 2.27 can read/write ".neta" files.

If *file* already exists, it is overwritten. The net is always saved using a "safe-save", which writes it to a new file, and then if there was no problem, it deletes the old file and changes the name of the new file to that of the old. That way there is no risk of data loss in case of an interruption due to a software error or hardware failure.

If there are findings entered in *net*, you may want to retract them with RetractNetFindings_bn before writing *net*, since otherwise they will be saved in the file.

If the file size is very large, it may be because of large tables (such as CPTs). If these are defined by equations, it may be worthwhile to delete them with DeleteNodeTables_bn before writing the net to file, and restoring them with EquationToTable_bn after reading the net back in.

Version:

This function is available in all versions. Versions previous to 2.27 could not read/write files in .neta format.

See also:

NewFileStream_ns	Generates the required `stream_ns`
ReadNet_bn	Reads back the net saved
RetractNetFindings_bn	May want to retract findings before saving net
WriteNetFindings_bn	Just save the findings currently entered as a case
GetNetFileName_bn	Later retrieve the name of the file written to

Example:

```
// Sets net2 to a copy of net1, but without its visual information
//
file = NewFileStream_ns ("temp.dne", env, NULL);
WriteNet_bn (net1, file);
net2 = ReadNet_bn (file, NO_VISUAL_INFO);
if ((error = GetError_ns (env, WARNING_ERR, NULL)) != NULL)
    fprintf (stderr, "%s\n", ErrorMessage_ns (error));
```

WriteNetFindings_bn

```
caseposn_bn WriteNetFindings_bn (const nodelist_bn* nodes,
                                 stream_ns* file, long ID_num,
                                 double freq)
```

Saves in *file* the set of findings currently entered in *nodes*, so that later they can be read back with

ReadNetFindings_bn.

It saves findings of discrete nodes and values of continuous nodes, but not likelihood findings, or negative findings (i.e., findings which say that a node is not in some state).

If *file* already exists, this will add the case to it (unless it is not a case file, in which case an error will be generated). If you wish to write over the existing file, delete it before calling this.

The first case determines what columns will be included in the file. Each node in *nodes* will become one column.

Pass -1 for *ID_num* and/or *freq* if you do not want columns for them to appear in the case file. If any cases will need them, they must be included in the first case written to the file.

It returns the file position of the new case (which can later be passed to ReadNetFindings_bn).

It only saves findings from the nodes of *nodes*, and if the file already exists, it won't save findings from any of *nodes* that were not included in the node list used to first construct the file.

It is advised to give case files the extension ".cas" (i.e., the file name passed to NewFileStream_ns ends with ".cas"). That way they can be more easily identified by the Netica Application program.

You can control the characters Netica uses to separate findings, and to indicate a finding is absent, with the functions SetCaseFileDelimChar_ns and SetMissingDataChar_ns, respectively.

Netica won't modify or free the passed *nodes* list.

Version:

This function is available in all versions.
In versions previous to 2.26, this function was named **WriteCase_bn**.

See also:

ReadNetFindings_bn	Reads back the case that WriteNetFindings_bn saves
WriteNet_bn	Saves the whole net, including findings
SetCaseFileDelimChar_ns	Controls which character Netica uses to separate findings
SetMissingDataChar_ns	Controls which character Netica uses to indicate a node has no finding
NewNodeList2_bn	Creates the node list

15 Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X