Netica™ API

Programmer's Library

Reference Manual

Version 1.01

Netica API Programmer's Library

Reference Manual

Version 1.01

November 10, 1995

Published by:

Norsys Software Corp.
2315 Dunbar St.
Vancouver, BC, Canada
V6R 3N1
http://www.norsys.com

Contents

1 Introduction

This reference manual is for the Netica Programmer's Library, also known as the "Netica API" (Application Program Interface), which is a module designed for 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, but not embeddable within another program.

This manual assumes that you are familiar with the C programming language and your development system (including its C foreign language interface, if you are programming in some other language). It also assumes familiarity with Bayesian belief networks or influence diagrams, although it has a little introductory material on issues that are new or generally not well understood. Questions and comments about material in this manual may be sent to norsys_info@norsys.com.

1.1 Netica Programmer's Library

The Netica API is a complete library of C-callable functions for working with Bayesian belief networks and influence diagrams. It contains functions to build, learn, modify, transform, save and read networks, as well as a powerful inference engine. It may be embedded in programs written in any language (such as C++), as long as the language can call C functions.

Programs that use the Netica API completely control it. For example, Netica functions will not take any action until called, Netica will not do any I/O unless requested to, and its functions will not take an unpredictable amount of time before returning.

It may be used in conjunction with other C or C++ libraries (and it won't interfere with them), but it doesn't require any other library except the standard C library. Versions of the Netica API are available for MS Windows, Sun Sparc and Macintosh, 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.

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.

Characteristics:

- Has operations to build and modify networks in memory, as well as to save them to file and read them back.
- Compiles belief (Bayesian) networks and influence diagrams into a junction tree of cliques for fast probabilistic reasoning.
- Can solve decision networks to find optimal sequences of decisions and conditional plans.
- Can learn probabilistic relations from data.
- Networks can learn from experience while they are being queried to do probabilistic inference.
- Can reverse individual links (while maintaining the same joint probability distribution) and "sum out" nodes of an influence diagram or belief net.
- Accepts likelihood findings (i.e., virtual evidence), and findings of the form that some variable is not is some state.
- Supports disconnected links, which makes possible libraries of probabilistic relationships.
- Has facilities for the easy discretization of continuous variables.
- Every node and network has a "user-settable/readable" field which can contain a pointer to anything you wish.
- Has a simple but powerful method for handling errors, which can generate very detailed error messages if desired.
- The programmer can 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.
- Fast and not too large (about 200K to 2M depending on platform/usage).
- Can work hand-in-hand with the Netica Application standalone product (for example, sharing the same files).

1.2 Getting Started

The easiest way to get started is to copy all the files supplied (see the "Files Included" section) into one directory, and then build the supplied project called "Demo". For each different development system, a project or makefile for "Demo" is provided, along with installation notes for that particular system. Assuming this compiles, links, and runs without errors, you can simply substitute your program files for the Demo.c file.

Below is the smallest program which properly uses Netica (sort of a "Hello World" example). If this compiles, links and runs without getting any error messages, then your Netica installation is probably successful.

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

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

    /*    Region A ...    */

    env = NewNeticaEnviron_bn (NULL);     /* replace NULL with your license string if desired */
    res = InitNetica_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 text of the program above in a file called "myapp.c". If you have purchased a Netica API license, you should change the argument passed to NewNeticaEnviron_bn from `NULL` to the password (called the "license number" on the invoice) provided to you, so the line looks something like:

```
env = NewNeticaEnviron_bn ("+SmithJ/UCS/310-1S4/12345");
```

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 the one that includes file I/O and floating point math. If you are using a Unix command line, you might use commands like:

```
cc -c myapp.c

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

The first line compiles myapp.c and puts the result 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. When you run the executable (in the above example, by typing "myapp"), it should print out something similar to this:

```
Netica (AtC) 1.01 M68, (c) 1990-1995 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_bn or InitNetica_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_bn and InitNetica_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.).

1.3 Namespace Conflicts

Whenever a C library is used, namespaces 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. Older libraries almost always restricted the prefix because old linkers were inconsistent with the number of characters they considered significant. Newer libraries sometimes restrict the prefix (to be compatible with the greatest number of other libraries, which also use prefix), or sometimes the suffix (because it creates more readable code).

All Netica global symbols end in "_ns" or "_bn". The "_ns" symbols are those used in all Norsys products, while the "_bn" symbols are those particular to belief networks or influence diagrams. Your program, or other libraries it includes, should not define any global symbols that end with these letters, or with _cs, _sn, _fc, _pl, _co or _lr (which may be used in future versions of Netica, or libraries which work with Netica). If this is a problem (for example, some symbols conflict with another library you must use), contact Norsys.

If you just have a conflict with a Netica type name or an enumeration constant, say `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 be *very* careful to use `NEXT_CASE_ns` and not `NEXT_CASE` when calling Netica functions.

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

1.4 Types

There are two kinds of data types involved in using Netica API: scalar types and opaque pointers. 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, errdanger_ns and nodekind_bn. In your source code, use the names for the enumeration constants defined in the NeticaAPI.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 (e.g. C structs) defined by Netica, you will do so with an *opaque pointer*. These are defined in the Netica.h header file so that complete type checking can be done, 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 so that you don't have to change your software to work with future versions of Netica.

1.5 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 Standard Library function 'free' to free any array, string, or structure that Netica originally created. In other words, each system 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 FreeNet_bn and DeleteNode_bn.

Whenever this manual says that Netica returns a "nonmodifiable" structure (including a struct, array, or character string), it is really returning a pointer to some structure that is being maintained within the Netica system. Netica doesn't return a duplicated copy of the structure in order to be more efficient, and this difference in efficiency can be major in many situations (especially if your code just looks quickly at some small part 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, then try to read the string you originally obtained, its contents will likely have been destroyed.

1.6 Files Included

Also provided with the Netica API is a file of example source code called NeticaEx.c. Nothing in NeticaEx.c is required by the Netica API; it is all optional. You can modify the functions in this file to precisely suit your needs, and place them in your program. All functions of the Netica API end in _ns or _bn, but functions defined in this file 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. If you modify them then you should document it, 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 make the appropriate modifications again.

The following is a complete list of files that make up the Netica API system:

Netica.lib - The actual object code library that you will link in to your program. Make sure you get the right version for your platform (Windows, Mac, SunOS, etc.).

Netica.h - The header file to #include in your source code in order to use the Netica API. It contains enumeration constants, type definitions and function prototypes for Netica.lib. A listing of this file appears as a chapter of this manual.

NeticaEx.c - Example, extra, and external C language source code that you can customize and put in your program to make working with Netica easier. Many functions from it appear as examples in the "Function Reference" chapter, and documentation for them can be found by looking them up in the index of this manual.

NeticaEx.h - The header file for NeticaEx.c.

Demo.c - A simple example program to test your Netica installation. Depending on your development system, there may be other files called Demo.* or NeticaDemo.* (with various extensions) which supply a project or makefile suitable for your development system to get you started.

ManualAPI.ps - A postscript file which can be printed out or viewed with GhostScript or a similar program. It is this manual.

LicenseAPI.txt - An ASCII text file containing the license agreement for users of Netica.

These files available from Norsys may also be of value:

AsiaEx.dnet - An example DNET file that is used by the example programs in this manual.

DNET_File_Format.txt - An ASCII text document describing the file format for DNET files.

netlib - This is a directory containing a number of DNET files that you may use as examples. They are belief networks and decision networks that have become classics in the literature.

You can download any of these files from the Norsys WWW site at: http://www.norsys.com.

2 Probabilistic Inference

2.1 Belief Networks and Probabilistic Inference

A belief network (also known as a Bayesian network or probabilistic causal network) 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 belief networks are designed to be used only once for a single world situation. More often, belief networks are designed for repetitively occurring situations. They may be constructed using expert knowledge provided by some person (which is based on a deep knowledge of structure, and on the experience of previous cases), by an automatic learning process which examines many previous cases, or by a combination of the two. If the network is to be used repetitively, then it may be considered as a *knowledge base*. Sometimes networks that are built to be used only once are constructed automatically on-the-fly, perhaps by pasting together pieces of networks 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 belief networks 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 unmeasurable 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 network, and provide initial conditional probabilities, based on his medical training and experience with previous cases. Then the network probabilities may be fine-tuned by using statistics from previous cases, and from new cases as they arrive.

When the belief network 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 belief network.

The nodes are then connected up with directed *links*. Usually a link from node A (the *parent*) to node B (the *child*) indicates that A causes B, that A partially causes or predisposes B, that B is an imperfect observation of A, that A and B are functionally related, or that A and B are statistically correlated. The precise definition of a link is based on conditional independence, and is explained in detail in an introductory work like Neapolitan90 or Pearl88. Finally, probabilistic relations are provided for each node, which express the probability of that node having different values conditioned on the values of its parent nodes.

After the belief network 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 belief network is called *belief updating*.

Probabilistic inference only results in a set of beliefs at each node; it does not change the network (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 Cases" chapter.

2.2 Netica's Probabilistic Inference

There are two ways that Netica can do regular probabilistic inference: by link reversals and by clique tree message passing. For most applications you will want to use clique tree message passing, because usually it is much faster, but you may want to use link reversals 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). This section deals with join trees; see the "Network Transforms" chapter for information on link reversals.

Netica uses the fastest known algorithm for exact general probabilistic inference in a compiled belief network, which is message passing in a *join tree* (or "junction tree") of cliques. This is based upon the work of LauritzenSpiegelhalter88, which is described in much simpler and more extensive terms in SpiegelhalterDLC93 and Neapolitan90.

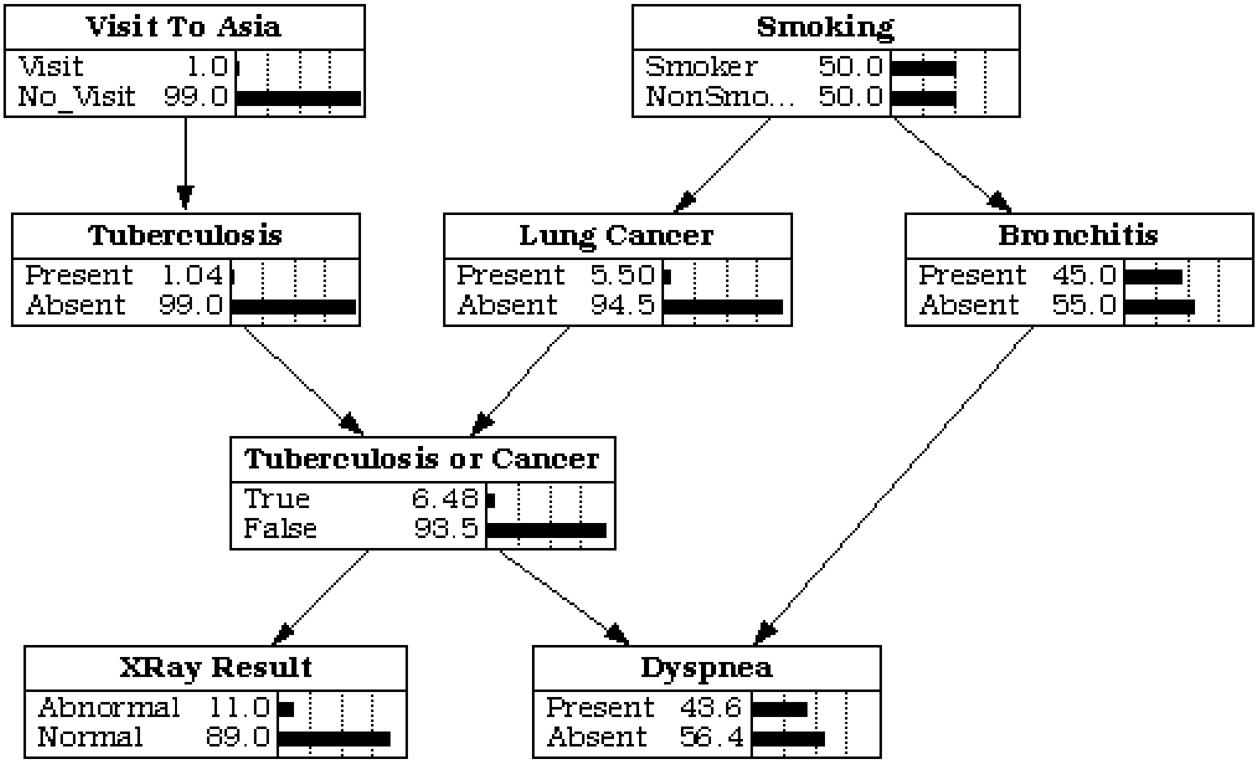
In this process the belief network is first "compiled" into a join tree. The join tree is implemented as a large set of data structures connected up with the original belief network, but invisible to you as a user of Netica. You enter findings for one or more nodes of the original belief network, and 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 join tree. It determines the resultant beliefs for each of the nodes of the original belief network, 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 join 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 network. 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 by writing a small program to try several), or just let Netica find a good one itself.

2.3 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 network 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 network we will use 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 network 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 standalone program, and just show the probabilities of each state before any findings have arrived.



Before the example code below will work, the file containing the network must exist. If you do not already have it, the next chapter shows how to construct it. At the end of that chapter is a listing of the file for the above network; you may wish to see the file in order to see exactly what the probabilistic relations are (it is missing the Bronchitis and Dyspnea nodes, but that will not matter for now).

```
#include "Netica.h"
#include "NeticaEx.c"

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

environ_ns *env;

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

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

    net = ReadNet_bn (FileNamed_ns ("AsiaEx.dnet", env), FALSE);
    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);

    EnterFinding ("Cancer", "Present", net);
    belief = GetNodeBelief ("Tuberculosis", "Present", net);
    CHKERR

    printf ("Given abnormal X-ray, Asia visit, and lung cancer, \n\
             the probability of tuberculosis is %g\n\n", belief);
end:
    FreeNet_bn (net);
    res = CloseNetica_bn (env, mesg);
    printf ("%s\n", mesg);
    exit (res < 0 ? -1 : 0);

error:
    fprintf (stderr, "API_Demo1: Error in %s\n",
             ErrorMessage_ns (GetError_ns (env, ERROR_ERR, NULL)));
    goto end;
}
```

The program starts by using NewNeticaEnviron_bn and InitNetica_bn to initialize the system as described in the Introduction. Next, ReadNet_bn is used to read the file and create the network 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. Later you can obtain the error number or an error message from this report. It is not imperative 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.

Next, CompileNet_bn builds the join tree of cliques and attaches it to the data structure of the belief network, but does not discard any of the information from the original belief network. We can now use this network 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 network. This step can be time consuming if the network 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 network 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 belief network 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 network 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%.

```
Netica (AtC) 1.01 M68, (C) 1990-1995 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.

3 Building and Saving Networks

In the previous chapter we loaded a network into memory from a file and then did probabilistic inference using it. Now we consider how to obtain the network file in the first place. There are 5 possibilities:

- Obtain it from Norsys, another company or a colleague (by email, disk, downloading from the Internet, etc.). The transfer is easy, since it is a machine independent ASCII text file.
- Create the file using a text editor, according to the DNET file specification.
- Write a program that creates the DNET file, containing the network.
- Use the Netica Application standalone program to construct the network 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 network in memory. Once the network 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 AsiaEx network used in the previous chapter (except, to be more brief, it doesn't include the two nodes Bronchitis and Dyspnea, which are not actually required for the inference examples of that chapter).

```
#include "Netica.h"
#include "NeticaEx.c"

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

environ_ns *env;

void main (void){
    net_bn *net;
    node_bn *VisitAsia, *Tuberculosis, *Smoking, *Cancer, *TbOrCa, *XRay;
    char mesg[MESG_LEN_ns];
    int res;
```

```
    env = NewNeticaEnviron_bn (NULL);
    res = InitNetica_bn (&env, mesg);
    printf ("%s\n", mesg);
    if (res < 0)  exit (-1);

    net = NewNet_bn ("AsiaEx", 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 (VisitAsia,   "Visit",   "No_Visit");
    SetNodeStateNames (Tuberculosis,"Present", "Absent");
    SetNodeStateNames (Smoking,     "Smoker",  "NonSmoker");
    SetNodeStateNames (Cancer,      "Present", "Absent");
    SetNodeStateNames (TbOrCa,      "True",    "False");
    SetNodeStateNames (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

    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, FileNamed_ns ("AsiaEx.dnet",env));
    CHKERR

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

```
    error:
        fprintf (stderr, "API_Demo2a: Error in %s\n",
                 ErrorMessage_ns (GetError_ns (env, ERROR_ERR, NULL)));
        goto end;
    }
```

First a new net is created with NewNet_bn, and each of the nodes are added 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 2 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 usually 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.

Next, the program sets the state names of the nodes using SetNodeStateNames which is defined in NeticaEx.c as a convenient way to call SetNodeStateName_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 relations 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

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);
```

Now the network is fully constructed in memory, and we could use it for inference, do network 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 standalone, 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 AsiaEx {

    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
    };
};
```

There are a number of other functions that may be used when constructing a network. For a list of them, see the "Low-Level Network Modification" section of the "Function Groups" chapter, and for detailed descriptions of each one, look it up in the "Function Reference" chapter.

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

4 Findings and Cases

In the "Probabilistic Inference" chapter we saw how to enter positive findings into a belief network 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 network, 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 network), 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 did 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
```

4.1 Cases and Case Files

The set of all findings entered into the nodes of a single belief network 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. There are three main purposes for case files: to act like databases, for future restoration of cases into a network (perhaps to present to a user, or to add new findings and find new beliefs), and to use for future learning of a belief network.

There are 3 different ways you can make a case file:

- Use a text editor to manually construct it, according to the specification below.
- Take an existing tab delimited text data file from a spreadsheet or database output, and add the two header lines described below (usually when you copy from a spreadsheet and then paste into a text file, this will be the format you get).
- Call Netica API functions to enter the case as findings into a belief network, write the case to a file, and repeat for each case to be put in the file.

A case file starts off with "// ~->[CASE-1]->~".

Then it contains a line of headings for the columns. Each heading corresponds to one variable of the case. They are ASCII strings separated by tabs, and each is the name of the node used to represent the variable.

The case data is next, with one case per line. The values of the variables are in the same order as the heading line, and are separated by tabs. The value of a discrete variable is given by its state name, or if it doesn't have a state name, then by its state number. 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 variables is given by a number in integer, decimal, or scientific notation (e.g. -3.21e-7). If it 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, so that the case file can be used for different discretizations of that variable in the future.

Missing data is represented by a single asterisk ("*"). 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 at the end of the line.

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, and does not have to be ordered in the file. The heading for this column is "IDnum". The other 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 "temp.cases" which is produced by the example program in the "Learning From Cases" chapter (the case file you obtain may be a little different, since random numbers are involved). It has an IDnum column, but no frequency column.

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

IDnum   VisitAsia   Tuberculosis   Smoking     Cancer    TbOrCa   XRay       Bronchitis   Dyspnea
0       No_Visit    Present        Smoker      Absent    True     Abnormal   Absent       Present
1       No_Visit    Absent         Smoker      Absent    False    Normal     Present      Present
2       No_Visit    Absent         Smoker      Present   True     Abnormal   Present      Present
3       No_Visit    Absent         NonSmoker   Absent    False    Normal     Absent       Absent
4       No_Visit    Absent         Smoker      Present   True     Abnormal   Present      Present
5       No_Visit    Absent         Smoker      Absent    False    Abnormal   Present      Present
....
118     No_Visit    Absent         Smoker      Absent    False    Normal     Present      Present
119     No_Visit    Absent         Smoker      Present   True     Abnormal   Present      Present
```

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]->~

IDnum  Starts  BatAge  Cranks  Lights  StMotor  SpPlug  MFuse  Alter  BatVolt  Dist  PlugV   Timing
1      False   5.9     False   off     *        fouled  okay   *      dead     *     *       good
2      False   1.3     False   off     *        okay    okay   *      dead     *     none    bad
3      False   5.2     False   off     Okay     okay    okay   Okay   dead     Okay  none    good
4      True    4.1     True    bright  *        okay    okay   *      strong   Okay  strong  *
5      True    2.7     *       bright  *        wide    okay   *      strong   Okay  *       *
6      *       1.3     True    bright  *        fouled  okay   *      *        Okay  strong  good
7      False   1.7     True    off     Okay     okay    okay   Okay   dead     *     none    good
8      True    2.9     True    bright  *        *       *      *      strong   Okay  strong  *
```

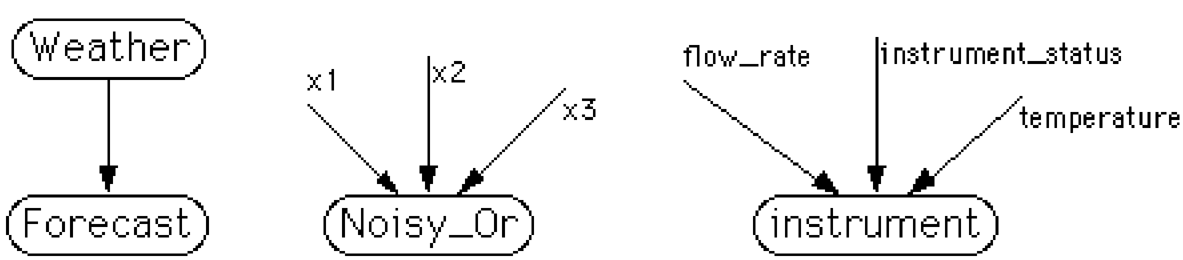
Future versions of Netica will support more advanced operations with cases, including a more efficient file representation, and a way of using belief networks as "indexing functions" to do the kind of lookup common in case-based reasoning. However, the above described type of file format will always be supported.

5 Network Transforms

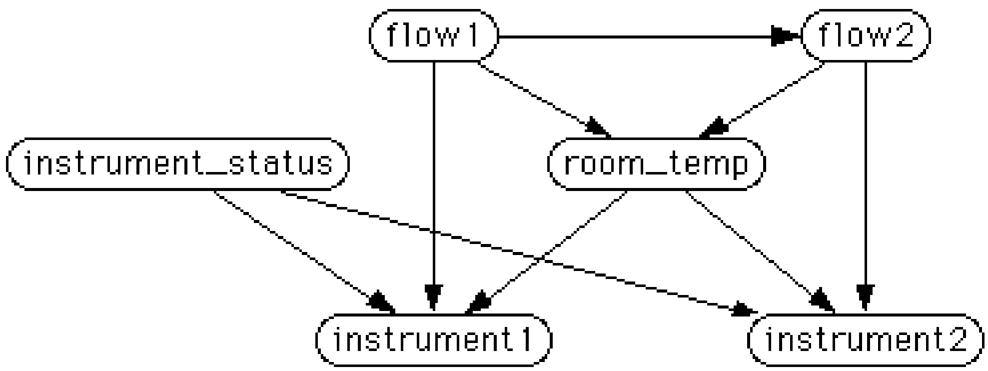
5.1 Node Libraries

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

For example, suppose that you made a simple network 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 you put the link from weather to forecast because often its 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 networks for inference involving the weather and its forecast, such as the decision problem discussed in the "Decision Problems" chapter.



As another example, suppose you have a device for measuring the flow rate in a pipe. It can produce biased readings depending on the temperature, and it can break in a few different ways, each of them producing wrong or inaccurate readings. You can model the device with a 4 node network, 1 for the reading on the device, and 3 parents corresponding to: actual flow rate, ambient temperature, and device status (okay, broken-1, broken-2, etc.). You enter the probabilistic relationships, and then you disconnect the node from its parents and place it in a library (see diagram, disconnection and grafting are explained below). Later, if you have a network to model a situation in which you have made 4 measurements with the device, you just duplicate the device characteristics node from the library 4 time into the new network, and graft it to the appropriate nodes in that network (see diagram). Note that if the ambient temperature could be different between measurements, or if the device could have broken along the way, then you would use multiple connected nodes for these variables. Automating the process of network construction for new situations is an area of active research, with dynamic belief networks, templates and graph grammars being some of the methods used.



Netica makes it easy to maintain libraries of disconnected nodes and subnetworks. To make a new library, just use NewNet_bn. Nodes and subnetworks can be copied to it using DuplicateNodes_bn, which can transfer material from one network to another, and also copies all the links between nodes in a subnetwork. When a node is being duplicated, but one of its parents isn't, then DuplicateNodes_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 of node 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.

When you want to use something in the library, you call DuplicateNodes_bn again, this time to duplicate from the library into the new network. 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);
WriteNet_bn (libnet, FileNamed_ns ("Library.dnet", env));

FreeNet_bn (net);
FreeNet_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 network 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 network.>
// <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 network.

libnet = ReadNet_bn (FileNamed_ns ("Library.dnet", env), FALSE);
instrument1 = DuplicateNode (NodeNamed_bn ("instrument", libnet), appnet);
instrument2 = DuplicateNode (NodeNamed_bn ("instrument", libnet), appnet);

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

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

SwitchNodeParent_bn (LinkNamed_bn ("flow_rate", instrument2), instrument2, flow2);
SwitchNodeParent_bn (LinkNamed_bn ("temperature", instrument2), instrument2, rtemp);
SwitchNodeParent_bn (LinkNamed_bn ("instrument_status", instrument2), instrument2, status);
```

Now the application network *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 in a way that properly accounts for random (uncorrelated) and systematic (correlated) errors, as well as all the background knowledge about the situation.

5.2 Network Reduction

Suppose you have a large network 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 network might by Gender, and in the restricted application the network 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 *evidence nodes*, and those whose beliefs you will want are called *query nodes*. The hundreds of other nodes might be involved in intermediate calculations, but you don't care about their values explicitly.

You can simplify the large network 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 evidence nodes and the query nodes, and pass it to AbsorbNodes_bn. The resulting 6 node network 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 evidence node, then you can then further simplify things by removing any links that go from evidence node P to evidence 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 evidence nodes ancestors of all the query nodes, then you can remove all the links between the evidence nodes. Any evidence 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 network 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 evidence nodes is *very* small, the resulting network must be simpler, although the transformations to generate it might temporarily require a lot of memory.

5.3 Probabilistic Inference by Node Absorption

From the previous section you may have realized it is possible to do probabilistic inference using node absorption, by entering all the findings, and then absorbing all the nodes except for a single 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 belief network, doing belief updating, and obtaining the beliefs 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 network each time, since it is destroyed in the process), and so it will probably be slower. If you have a number of queries, the compiling method will also be faster because you have to compile only once. But if you only need the beliefs of one node, for one set of findings, and there are many nodes in the network that are irrelevant to the particular query, then the node absorption method can be much faster.

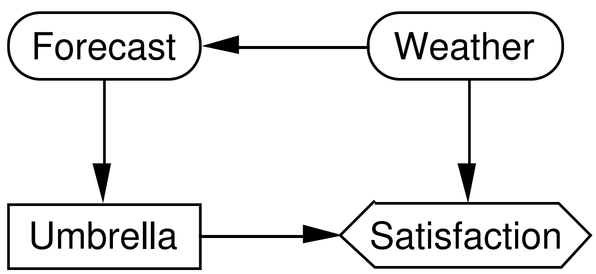
The next version of Netica will have a new inference method that will be ideal for one-time inferences in a network built on the fly, or inference for which many of the nodes in the network are not relevant. It will always be faster than the node absorption method, and will be faster than the compilation method in the above-mentioned situations.

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

When using AbsorbNodes_bn for decision problems, the decision nodes must have no-forgetting links, and if the list of nodes to absorb does not include all the nodes in the network, it must consist of a descendant subnetwork (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.

6 Decision Problems

Chapter 1 discussed probabilistic inference using a belief network. That is used to determine new beliefs (in the form of probabilities) as observations are made or facts gathered. It is composed only of *nature nodes* (also known as "chance" nodes). By adding *decision nodes* and *utility nodes* to a belief network, we obtain a *decision network* (also known as an influence diagram). Decision networks can be used to find the optimal decisions which will maximize expected utility.



As an example, let's consider a very tiny decision network from Ross Shachter known as "Umbrella". It has 2 *nature nodes* representing what the weather forecast is in the morning (sunny, cloudy or rainy), and what the weather actually is during the day (sunshine or rain), a *decision node* of whether or not to take an umbrella, and a *utility node* (also known as "value" nodes) 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. 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).

Below is a program to build this decision network in memory, and then solve it (i.e., find the optimal decisions). Much of it is very similar to building a belief network (see the chapter "Building and Saving Networks" 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 SetNodeFuncValue to build up the relations of a deterministic node instead of SetNodeProbs, but it works in a similar fashion.

```
#include "Netica.h"
#include "NeticaEx.c"

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

environ_ns *env;

void main (void){
    net_bn *net;
    node_bn *weather, *forecast, *umbrella, *satisfaction;
    state_bn forecast_state, decision;
    char mesg[MESG_LEN_ns];
    int res;

    env = NewNeticaEnviron_bn (NULL);
    res = InitNetica_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 (forecast, "sunny",          "cloudy",    "rainy");
    SetNodeStateNames (weather,  "sunshine",       "rain");
    SetNodeStateNames (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
    SetNodeFuncValue (satisfaction,  20, "sunshine", "take_umbrella");
    SetNodeFuncValue (satisfaction, 100, "sunshine", "dont_take_umbrella");
    SetNodeFuncValue (satisfaction,  70, "rain",     "take_umbrella");
    SetNodeFuncValue (satisfaction,   0, "rain",     "dont_take_umbrella");
    CHKERR

    OptimizeDecisions_bn (net);
    CHKERR

    for (forecast_state = 0;  forecast_state < 3;  forecast_state++){
        decision = GetNodeFuncValue_bn (umbrella, &forecast_state);
        pprintf ("If the forecast is '%s', the best decision is %s.\n",
                 GetNodeStateName_bn (forecast, forecast_state),
                 GetNodeStateName_bn (umbrella, decision));
    }

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

error:
    fprintf (stderr, "API_Demo2: Error in %s\n",
             ErrorMessage_ns (GetError_ns (env, ERROR_ERR, NULL)));
    goto end;
}
```

Once the network is built, the program calls OptimizeDecisions_bn, which attaches to each decision node a deterministic function. This function 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. These decisions are the ones that maximize the expected value of the sum of the utility nodes. The program uses GetNodeFuncValue_bn to access this decision function, and prints out the following.

```
Netica (AtC) 1.01 M68, (C) 1990-1995 Norsys Software Corp.

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.
```

7 Learning From Cases

Belief network learning is the automatic process of determining a suitable belief network given data in the form of cases. Each case represents an example, event, object or situation in the world (presumably that exists or has occurred), and it supplies values for a set of variables which describes the event, object, etc. Each variable will become a node in the learned network (unless we want to ignore some of them). Some cases may not have values for some variables that other cases do (this is known as *missing data*), but there should be a significant amount of overlap between the variables of the cases.

The learned network can be used to analyze a new case drawn from the same population as the cases used for learning. Typically the values for some variables of the new case will be known. These are entered as findings, and then probabilistic inference is done 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 network correspond to a causal structure, and the query nodes are ancestors of the nodes with findings, then you could say that the network has learned to do diagnosis. If the query nodes are descendants, then the network has learned to do prediction, and if the query node corresponds to a "class" variable, then the network has learned to do classification. Of course the same network could do all three, even at the same time.

The belief network learning task has traditionally been divided into 2 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 network. *Parameter learning* determines the conditional probability relationship 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).

While learning, Netica assumes independence between each of the conditional probabilities of a node's relation to its parents. This works well when there is lots of data (or nodes have few parents and probabilities are not very close to zero), but can result in under-confidence when there is little data. It may not have seen some parent configuration, so its probability is uniform for it, but it may be better off if it assumed it behaved similarly to another similar parent configuration (i.e., assuming a little more independence between variables, and a little less independence between conditional probabilities).

Future versions of Netica will make it possible to not assume independence of the conditional probabilities, if that is desired. Netica will be able to learn relations from a set of special models. It will also be able to suggest changes to the structure, thereby partially doing structure learning.

7.1 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, with probability having a much sounder theoretical basis. A deficiency of using just 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, if you are going to sequentially draw a number of balls from the bag, then it is 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.

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 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 belief network for this. That approach works fine for this simple problem, but you can imagine that if you had many interrelated variables, that it would become too complicated.

If during the learning we consider the conditional probabilities being learned to be independent of each other, and the prior distribution to be Dirichilet, 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 is also known as "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.

Before learning begins (providing there has been no previous learning or entry of probabilities by an expert) the network starts off in a state of ignorance. All probabilities start as uniform, and experience starts off as `BaseExperience_bn`. There is some controversy as to the best value of `BaseExperience_bn` but usually it doesn't matter much. 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.

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 is the normalization constant).

7.2 How To Do It

There are two ways to learn probabilities 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 belief network, you enter it into the network as findings (see the "Findings and Cases" chapter). Then CaseRevisesProbs_bn is called with a list of nodes. Only nodes in the passed list will have their probabilities revised, so normally it will be a list of all the nodes in the network. Each of these nodes 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 CaseFileRevisesProbs_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 CaseRevisesProbs_bn each time. See the "Findings and Cases" chapter for more information on creating a file of cases.

7.3 Example of Learning

The program below will demonstrate making a file of cases randomly sampled from a belief network, and then using them to learn a new network. First it reads from file a very simple example network (the network that was constructed in the "Building and Saving Networks" chapter), and then duplicates it by making a new network and duplicating all the nodes into it.

Next it removes the probabilities and experience from the duplicated nodes with DeleteNodeRelation_bn. The idea is to relearn them later.

It deletes a file named "temp.cases" if there is one (otherwise it would try to add the cases to this file). Then, in a loop repeated 120 times it generates a random case in the original belief network. These cases will be distributed according to the probability distribution of that network. Each case is saved to the case file named "temp.cases". For a listing of this file, see the "Cases and Case Files" section of the "Findings and Cases" chapter.

Then, in a loop it reads each case from the file into the new network, where CaseRevisesProbs_bn uses it to learn the probabilities for the new network. Instead of the loop, we could have used the single line:

```
CaseFileRevisesProbs_bn (casefile, learned_nodes, 0, 1.0);
```

which would execute faster, but would not have demonstrated the case-by-case way, which you may want to use sometime if you have some cases to filter out, or some such thing.

Finally the new network is saved 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 network will be to the original. Of course, in a real application there would be no point in relearning a network 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 network comes out similar to the old.

```
#include "Netica.h"
#include "NeticaEx.c"

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

environ_ns *env;

void main (void){
    net_bn *orig_net, *learned_net;
    const nodelist_bn *orig_nodes;
    nodelist_bn *learned_nodes;
    caseposn_bn case_posn;
    int numnodes;
    const int numcases = 120;
    stream_ns *casefile;
    char mesg[MESG_LEN_ns];
    int nn, res;

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

    orig_net = ReadNet_bn (FileNamed_ns ("AsiaEx.dnet", env), FALSE);
    orig_nodes = GetNetNodes_bn (orig_net);
    SetNetAutoUpdate_bn (orig_net, FALSE);
    CHKERR
```

```
    learned_net = NewNet_bn ("Learned_Asia", env);
    learned_nodes = DuplicateNodes_bn (orig_nodes, learned_net);
    numnodes = LengthNodeList_bn (learned_nodes);
    for (nn = 0;  nn < numnodes;  nn++)
        DeleteNodeRelation_bn (NthNode_bn (learned_nodes, nn));
    CHKERR

    remove ("temp.cases");
    casefile = FileNamed_ns ("temp.cases", env);
    for (nn = 0;  nn < numcases;  nn++){
        RetractAllFindings_bn (orig_net);
        res= RandomCase_bn (orig_nodes, 0, 20);
        if (res >= 0)
            WriteCase_bn (orig_nodes, casefile, nn, -1);
        CHKERR
    }

    case_posn = FIRST_CASE;
    while(1){
        RetractAllFindings_bn (learned_net);
        ReadCase_bn (&case_posn, casefile, learned_nodes, NULL, NULL);
        if (case_posn == NO_MORE_CASES)  break;
        CaseRevisesProbs_bn (learned_nodes, 0, 1.0);
        case_posn = NEXT_CASE;
        CHKERR
        }

    WriteNet_bn (learned_net, FileNamed_ns ("Learned_AsiaEx.dnet", env));
    CHKERR

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

error:
    fprintf (stderr, "API_Demo3: Error in %s\n",
             ErrorMessage_ns (GetError_ns (env, ERROR_ERR, NULL)));
    goto end;
}
```

7.4 Fading

When a belief network 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 belief network 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 FadeProbs_bn with a node and a degree from 0 to 1, and it will reduce the experience and smooth the probabilities of the node by an amount dictated by the degree, with 0 having no effect and 1 creating uniform distributions with no experience. Calling FadeProbs_bn once with `degree` = 1-d, and again with `degree` = 1-f, is equivalent to a single call with `degree` = 1-df.

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*

The global variable *BaseExperience_bn* should have the same value as it did when the learning from cases was done, if learning from cases was done (see section 7.1 for more information).

When learning in a changing environment, you would normally call FadeProbs_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 FadeProbs_bn in the "Function Reference" chapter.

8 Function Reference

AbsorbNodes_bn

`void AbsorbNodes_bn (nodelist_bn *nodes)`

Absorb out (sum or max) all of *`nodes`* from their network. This removes and frees the nodes while maintaining the joint distribution of the remaining nodes.

In order to maintain the joint distribution it may have to add links. Absorbing a nature node which has no evidence will only add links from the parents of the removed node and its children's parents, to the removed nodes children. However, if it has evidence, many links between the ancestors of the removed node may be added (possibly resulting in an out-of-memory error). 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.

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).

If it is impossible to absorb out all of *`nodes`*, as many as possible will be absorbed out and then an error message generated.

Each of the nodes in *`nodes`* must be in the same network.

WARNING: The *`nodes`* structure passed to AbsorbNodes_bn will be freed by this routine, so it should not be used subsequent to calling this routine (it would contain invalid data anyway, since each of the nodes it refers to will have been freed).

See also:

DeleteNode_bn	Removes a node without maintaining joint distribution

Example:

```
// Handy function to absorb a single node
//
void AbsorbNode (node_bn *node){
    nodelist_bn *nodes = NewNodeList_bn (1, env);
    SetNthNode_bn (nodes, 0, node);
    AbsorbNodes_bn (nodes);
}
```

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 conditional probability 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 probability table is changed.

See also:

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

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 `INT_MAX`, 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.

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`

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 which is to be done 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_ns`	Return the current setting without changing it

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 (or even NO_CHECK)`, but they may occasionally temporarily change it to `REGULAR_CHECK` in order to do user input checking.

In order to determine the current degree of checking without changing it, call this with *`setting`* = QUERY_ns, and the checking degree will be returned.

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

---

CaseFileRevisesProbs_bn

```
void  CaseFileRevisesProbs_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 conditional probabilities of each node in *`nodes`*. This function does the same thing as CaseRevisesProbs_bn, for each of the cases in *`file`*, but is more efficient than multiple calls to CaseRevisesProbs_bn. See the description of CaseRevisesProbs_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 network.

Pass 0 for *`updating`*. It is there only for future expandability.

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 network 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.

See also:

CaseRevisesProbs_bn	Revise probabilities with a single case

CaseProbability_bn

```
double  CaseProbability_bn  (net_bn *net)
```

Returns the joint probability of the findings entered into network *`net`* so far.

If the computations for belief updating haven't been done since the last findings were entered, or the last network modifications made, it will be done before this routine returns, which can be quite time consuming (you can use IsBeliefUpdated_bn before calling this to find out if belief updating will be done).

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

See also:

IsBeliefUpdated_bn	Find out if CaseProbability_bn is going to trigger belief updating

---

CaseRevisesProbs_bn

```
void  CaseRevisesProbs_bn  (const nodelist_bn *nodes,
                             int updating,  double degree)
```

The current case 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 relations between the nodes to include 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 network.

Pass 0 for `updating`. It is there only for future expandability.

`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 routine 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 FadeProbs_bn was made in between, then d will be weighted by the degree passed to FadeProbs_bn.

See also:

CaseFileRevisesProbs_bn	Batch version, more efficient than one at a time
FadeProbs_bn	Use between calls to CaseRevisesProbs_bn when the world is changing during learning

---

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 freed as well.

The name of this function is very similar to ClearErrors_ns, but hopefully there won't be any confusion since its 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!

See also:

ClearErrors_ns	Clears all the errors

Example:

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

Example:

See GetError_ns

ClearErrors_ns

`void ClearErrors_ns (environ_ns *env, errdanger_ns danger)`

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

danger must be one of `NOTHING_ERR`, `REPORT_ERR`, `NOTICE_ERR`, `WARNING_ERR`, `ERROR_ERR`, `XXX_ERR` (for a description of these see ErrorDanger_ns). Pass `ERROR_ERR` to clear them all (except for `XXX_ERR` errors, which you probably shouldn't be clearing).

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!

See also:

ClearError_ns	Clears just a single error

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 InitNetica_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_bn and InitNetica_bn), but no data structures created by one session may be used by another.

If Netica closed successfully a non-negative integer will be returned, whereas if there was some problem a negative integer will be returned and an error message left in *`mesg`*.

Example:

See InitNetica_bn.

CompileNet_bn

`void CompileNet_bn (net_bn *net)`

Compiles network *`net`* for fast belief updating.

If the network is an auto-update network (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.

When the network is compiled, first an "elimination ordering" is determined, which is a list giving all the nodes of the network in some order, and using that 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 network is already compiled has no effect unless the network or its elimination ordering has been changed, in which case the network will be recompiled.

See also:

GetNodeBeliefs_bn	Convert findings to beliefs using the compiled network
SetNetElimOrder_bn	Set the elimination ordering to use when compiling the network
GetNetElimOrder_bn	Obtain the elimination ordering used to compile the network

DeleteLink_bn

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

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

`link_index` should be 0 for the first parent, and must be less than the number of links entering child.

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 DeleteLinks defined in the example below.

If *`child`* has a conditional probability 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.

See also:

AddLink_bn	Adds a link between two nodes
SwitchNodeParent_bn	Switches parents without changing conditional probabilities

Example:

```
// Removes all links from node parent to node child
//
void DeleteLinks (node_bn *parent, node_bn *child){
    const nodelist_bn *parents;
    int pn, num_parents;
again:
    parents = GetNodeParents_bn (child);
    num_parents = LengthNodeList_bn (parents);
    for (pn = 0;  pn < num_parents;  pn++)
        if (NthNode_bn (parents, pn) == parent){
            DeleteLink_bn (pn, child);
            goto again;
        }
}
```

DeleteNode_bn

`void DeleteNode_bn (node_bn *node)`

Removes *`node`* from its network, 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.

See also:

NewNode_bn	Creates a new node in a network

DeleteNodeRelation_bn

`void DeleteNodeRelation_bn (node_bn *node)`

Remove any information on *node*'s probabilistic or functional relation with its parents, including any experience information in the case of a probabilistic relation.

It does not remove the links from the parents of *node* to *node*.

See also:

DeleteLink_bn	Reduce the number of parents of the node
SetNodeFuncValue_bn	Give a node a functional relation with its parents
SetNodeProbs_bn	Give a node a probabilistic relation with its parents
NewNode_bn	Creates a new node without any relation information

DuplicateNodes_bn

```
nodelist_bn *DuplicateNodes_bn (const nodelist_bn *nodes,
                                net_bn *new_net)
```

Duplicates *nodes*, putting them in the network *new_net*. It is okay if *new_net* is the same as the network they are already in. All of *nodes* must be in the same network 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 FreeNodeList_bn). The order of the new list will correspond with the order of the old list. The old list is not modified or freed by DuplicateNodes_bn.

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 network.

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" code example below. If you wish to duplicate a whole network, see DuplicateNet below.

See also:

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

Example:

```
// 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 freed.
//
// In the process each node in nodes is freed, 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 = DuplicateNodes_bn (nodes, new_net);
    for (nn = 0;  nn < num_nodes;  nn++)
        DeleteNode_bn (NthNode_bn (nodes, nn));
    FreeNodeList_bn (nodes);         // because its full of invalid pointers
    return new_nodes;
}
```

Example:

```
// Handy functions to duplicate a single node
//
node_bn *DuplicateNode (node_bn *node, net_bn *new_net){
    node_bn *new_node;
    nodelist_bn *newnodes, *nodes = NewNodeList_bn (1, env);
    SetNthNode_bn (nodes, 0, node);
    newnodes = DuplicateNodes_bn (nodes, new_net);
    new_node = NthNode_bn (newnodes, 0);
    FreeNodeList_bn (nodes);
    FreeNodeList_bn (newnodes);
    return new_node;
}

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

Example:

```
// Duplicates a whole network
// Even duplicates the name, but you might want to give it a new name.
//
net_bn *DuplicateNet (net_bn *net, const char *newname){
    nodelist_bn *neworder;
    net_bn *new_net = NewNet_bn (newname, env);
    nodelist_bn *new_nodes = DuplicateNodes_bn (GetNetNodes_bn (net), new_net);

    neworder = MapNodeList (GetNetElimOrder_bn (net), new_net);
    SetNetElimOrder_bn (new_net, neworder);
    FreeNodeList_bn (neworder);

    FreeNodeList_bn (new_nodes);
    SetNetAutoUpdate_bn (new_net, GetNetAutoUpdate_bn (net));
    SetNetTitle_bn (new_net, GetNetTitle_bn (net));
    SetNetComment_bn (new_net, GetNetComment_bn (net));
    SetNetUserData_bn (new_net, 0, GetNetUserData_bn (net, 0));
    return new_net;
}
```

Example:

See DisconnectNodeGroup in file NeticaEx.c.

DupNodeList_bn

`nodelist_bn  *DupNodeList_bn  (const nodelist_bn *nodes)`

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

If a function returns a nonmodifiable list of nodes, this may be used to make a modifiable copy.

If the system runs out of memory, `NULL` will be returned and an error generated.

When you are finished with the list returned, pass it to FreeNodeList_bn.

WARNING: Do not pass it to the Standard C 'free' function.

Your software may use the returned list until disposing of it, or calling CloseNetica_bn, without fear that it will be modified by Netica (unless, of course, you deliberately pass it to a routine designed to change it).

See also:

FreeNodeList_bn	Free the new list created
NewNodeList_bn	Make a new empty list
DuplicateNodes_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`*.

If *`node`* already has findings that we wish to override with this new finding,

RetractNodeFindings_bn should be called first, otherwise an "inconsistent findings" error could result (see ChangeFinding in the examples below).

If the network is an auto-update network (see SetNetAutoUpdate_bn), then a belief updating will be done to reflect the new finding as soon as it is entered, otherwise GetNodeBeliefs_bn must be called before belief updating is done.

See also:

EnterFindingNot_bn	To indicate that node isn't in some state
EnterNodeValue_bn	To enter the 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

Example:

```
// This routine 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 ChangeFinding (node_bn *node, state_bn state){
    RetractNodeFindings_bn (node);
    EnterFinding_bn (node, state);
}
```

Example:

```
// This routine 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 = NodeNamed_bn (node_name, net);
    state_bn state = StateNamed_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*.

This can be called repeatedly to indicate all the states that we know *node* isn't in.

It also works in conjunction with EnterNodeLikelihood_bn to accumulate further observations.

If the network is an auto-update network (see SetNetAutoUpdate_bn), then a belief updating will be done to reflect the new finding as soon as it is entered, otherwise GetNodeBeliefs_bn must be called before belief updating is done.

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

---

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").

By calling it 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 in between calls.

The likelihood finding is equivalent to the following scenario:

> There are a number of possible observations you can make: ObsA, ObsB, ... ObsN.
>
> P(B|Si) denotes the probability of making observation B if the true state of *node* is Si.
>
> The vector LB is composed of <P(B|S1), P(B|S2), ..., P(B|Sm)> where m is the number of states.
>
> You actually make observation B, so you enter the vector LB as a likelihood finding for *node* (or LA if observation A was made, etc.).

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 (for example, a vector of all 1's may be used to indicate no added knowledge at all).

If you enter several accumulating likelihood findings for a node, they should correspond to observations that are independent given the value of the node (although if they are nearly independent, the approximation is generally pretty good).

If they are not independent, then they should be manually combined before entering, taking their dependencies into account, or alternately they can be added to the network as children of *node*, with arcs between them showing the dependencies, and having positive findings.

If the network is an auto-update network (see SetNetAutoUpdate_bn), then a belief updating will be done to reflect the new finding as soon as it is entered, otherwise GetNodeBeliefs_bn must be called before belief updating is done.

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
GetNodeFinding_bn	To retrieve positive 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 must be 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 is more detailed information and it will automatically be converted to a discrete state.

If the network is an auto-update network (see SetNetAutoUpdate_bn), then a belief updating will be done to reflect the new finding as soon as it is entered, otherwise GetNodeBeliefs_bn must be called before belief updating is done.

See also:

EnterFinding_bn To enter a finding for a discrete or discretized node

---

ErrorDanger_ns

`errdanger_ns ErrorDanger_ns (const report_ns *error)`

Given `error`, a report of an error, 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 (false alarm or "this is a test")
`REPORT_ERR`	Not an error, but a report of success
`NOTICE_ERR`	Notice of something unusual
`WARNING_ERR`	Event occurred at "warning" level - safe to proceed (e.g., user error, recovered okay)
`ERROR_ERR`	Event occurred at "error" level - operation not properly finished, but no internal inconsistencies (e.g., user error, didn't finish process)
`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.

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.

Do not try to modify or free the string returned.

If you are going to save the returned string for any amount of time, make a copy of it as soon as it's returned, because when this error report is removed with ClearError_ns or ClearErrors_ns, the memory for the returned string will be freed.

See also:

ErrorNumber_ns	Returns the error's identification number
ErrorDanger_ns	Returns how serious the error is

Example:

See GetError_ns

ErrorNumber_ns

```
long  ErrorNumber_ns  (const report_ns *error)
```

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

See also:

ErrorMessage_ns	Returns a complete error message
ErrorDanger_ns	Returns how serious the error is

FadeProbs_bn

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

Smoothes the conditional probabilities of each node in `nodes` to indicate a lesser certainty because the world may have changed a little since the last learning.

`degree` must be between 0 and 1, with 0 having no effect and 1 creating uniform distributions with no experience. Calling FadeProbs_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 FadeProbs_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. See the example below.

See also:

CaseRevisesProbs_bn	This also contains a 'degree', which can be used to weight the impact of learning a case

Example:

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

Example:

```
// 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 network being learned
if (time - lasttime >= mindelay){
    double degree = 1 - pow (rate, time - lasttime);
    FadeProbsNodes (GetNetNodes_bn (net), degree);
    lasttime = time;
}
```

FileNamed_ns

`stream_ns  *FileNamed_ns  (const char *filename, environ_ns *env)`

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.

`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.

See also:

WriteNet_bn — Saves a network to a file with name specified by the passed stream

ReadNet_bn — Reads a network from the file identified by the passed stream

Example: See WriteNet_bn

```
//
// Examples for UNIX
//
file = FileNamed_ns ("temp3", env);
file = FileNamed_ns ("/local/project1/configure.bn.txt", env);
file = FileNamed_ns ("../nets/Umbrella.dnet", env);
//
// Examples for MSDOS / Windows95
//
file = FileNamed_ns ("temp3", env);
file = FileNamed_ns ("C:local\project1\configur.txt", env);
file = FileNamed_ns ("..\nets\Umbrella.dnt", env);
//
// Examples for MacOS
//
file = FileNamed_ns ("temp3", env);
file = FileNamed_ns ("local:project1:configure.bn.txt", env);
file = FileNamed_ns ("::nets:Umbrella.dnet", env);
```

FreeNet_bn

`void FreeNet_bn (net_bn *net)`

Frees the memory used by network *`net`* and all its substructures (e.g. nodes).

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

See also:

DeleteNode_bn — Remove a node from a network, and free it

FreeNodeList_bn

`void FreeNodeList_bn (nodelist_bn *nodes)`

Frees 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 free nonmodifiable node lists returned by functions like GetNetNodes_bn and GetNodeParents_bn (they will be freed when the network is freed).

See also:

DeleteNode_bn — Remove a node from a network, and free it

GetError_ns

```
report_ns  *GetError_ns  (environ_ns *env,
                           errdanger_ns danger,
                           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 `danger`.

If `after` is an error report currently in `env`, then the next error after it (at least as serious as `danger`) will be returned. This can be used to step through the errors, as shown in the second example below.

`danger` must be one of `NOTHING_ERR`, `REPORT_ERR`, `NOTICE_ERR`, `WARNING_ERR`, `ERROR_ERR`, `XXX_ERR` (for a description of these see ErrorDanger_ns).

If there is not an error to be returned, `NULL` is returned.

Use ErrorNumber_ns to get it's error number, and ErrorMessage_ns to get an error message for it.

See also:

ErrorMessage_ns	Gets a message for the error report
ErrorNumber_ns	Gets the identification number for the error report
ErrorDanger_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:

```
// This routine prints all the serious errors without removing them
//
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));
    }
}
```

GetLinkName_bn

```
const  char  *GetLinkName_bn  (const node_bn *node,
                               int link_index)
```

Returns a (nonmodifiable) string which is the name for input number *link_index* of *node*, or the empty string (rather than NULL) if the link does not have a name. Numbering 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 or less characters (not including terminating 0).

Do not try to modify or free the string returned.

See also:

GetNodeParents_bn	Gets the actual parents of the links
SetLinkName_bn	Set a link name

GetNetAutoUpdate_bn

```
int  GetNetAutoUpdate_bn  (const net_bn *net)
```

Returns a number greater than 0 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.

If the net is auto-updating and findings are entered for a number of nodes at the same time (e.g. reading in a case), then updating will be done after all the nodes have received their findings.

See also:

SetNetAutoUpdate_bn	Change whether a net is auto-updating

GetNetComment_bn

```
const  char  *GetNetComment_bn  (const net_bn *net)
```

Returns a (nonmodifiable) 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.

Do not try to modify or free the string returned.

See also:

SetNetComment_bn	Change the net's comment string
GetNodeComment_bn	Get the comment for a particular node

GetNetElimOrder_bn

```
const  nodelist_bn  *GetNetElimOrder_bn  (const net_bn *net)
```

Returns a (nonmodifiable) list of the nodes of *net* in their "elimination order" (which is used for triangulation in the compilation of *net*), or NULL if an order hasn't been determined yet.

Only appropriate nodes will be included in the list (for example, "ASSUME" nodes won't).

Do not try to modify or free the list returned.

See also:

SetNetElimOrder_bn	Set the elimination order to use for future compilations

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 for the features enabled (in parenthesis).

This routine can be called before InitNetica_bn (but of course it must be called after NewNeticaEnviron_bn, because that is needed to form *env*).

Do not try to modify or free the string returned as **version*.

See also:

NewNeticaEnviron_bn	Form the required environ_ns

Example:

```
// This requires a global variable env,
// initialized by a call to NewNeticaEnviron_bn:
//   env = NewNeticaEnviron_bn (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 (nonmodifiable) 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 or less characters (not including terminating 0).

Do not try to modify or free the string returned.

See also:

SetNetName_bn	Change the net's name
GetNetTitle_bn	Gets the net's title

GetNetNodes_bn

`const nodelist_bn *GetNetNodes_bn (const net_bn *net)`

Returns a (nonmodifiable) list of all the nodes in network *`net`*.

If *`net`* has no directed cycles, the list will be in topological order (i.e. a parent will always appear before its children).

WARNING: If you intend to keep the list around for any length of time, use DupNodeList_bn on it, since a call to some other Netica function may render it invalid. Consecutive calls to this routine 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.

To obtain the number of nodes in the net, use the length of the returned list (it will not contain duplicates or `NULL` entries).

Do not try to modify or free the list returned.

See also:

DeleteNode_bn	Removes a node from the net
NewNode_bn	Creates a new node for the net

GetNetTitle_bn

`const char *GetNetTitle_bn (const net_bn *net)`

Returns a (nonmodifiable) 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.

Do not try to modify or free the string returned, and duplicate it if you need it for an extended period of time.

See also:

GetNetName_bn	Gets the net's name (limited chars and length, always exists)
SetNetTitle_bn	Change the net's title

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.

Pass 0 for *kind*. It is there only for future expandability.

See also:

SetNetUserData_bn	Attach a user pointer to a net
GetNodeUserData_bn	Recover the user pointer attached to a particular node

---

GetNode Beliefs_bn

`const prob_bn *GetNodeBeliefs_bn (node_bn *node)`

Returns a (nonmodifiable) belief vector for the states of *node*.

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 StateNamed_bn). It will be normalized, so that the sum of its entries is 1.

This provides the current beliefs that the variable represented by *node* is in each of its states, given the network model and all findings entered into all nodes of the network (positive findings, negative findings and likelihood findings). The network containing *node* must have been compiled before calling this (with CompileNet_bn), or an error will be generated. If the network has been modified (other than just changing its probabilities) it must be recompiled, but just entering findings does not require a recompile.

node should be a discrete (or discretized with threshold levels) nature node.

If belief updating hasn't been done since the last findings were entered, it will be done before this routine returns, which can be quite time consuming (you can call IsBeliefUpdated_bn before calling this to find out if belief updating will be done).

Do not try to modify or free the vector returned.

See also:

IsBeliefUpdated_bn	Find out if GetNodeBeliefs_bn is going to trigger belief updating
GetNodeNumberStates_bn	Determine length of vector returned
CompileNet_bn	To do the initial compiling before entering findings

Example:

```
// This routine is useful to get the belief that a certain node is in
// a certain state, based on the names of the node and state.
//
double GetNodeBelief (char *node_name, char *state_name, net_bn *net){
    node_bn *node = NodeNamed_bn (node_name, net);
    state_bn state = StateNamed_bn (state_name, node);
    return GetNodeBeliefs_bn (node) [state];
    }
```

GetNode Children_bn

`const nodelist_bn *GetNodeChildren_bn (const node_bn *node)`

Returns a (nonmodifiable) list of the children of *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*.

Do not try to modify or free the list returned.

See also:

GetNodeParents_bn	Get a list of the parents

GetNodeComment_bn

`const char *GetNodeComment_bn (const node_bn *node)`

Returns a (nonmodifiable) 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.

Do not try to modify or free the string returned.

See also:

SetNodeComment_bn	Set the node's comment string
GetNetComment_bn	Get the comment for a whole network

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 to 0 or 1 will make a node deterministic. Using SetNodeFuncValue_bn on the node also will.

This function starts with "Is" rather than "Get" to indicate that it is not always a simple data access; it may take a little while to return.

See also:

SetNodeProbs_bn	To change whether a node is deterministic
GetNodeDiscrete_bn	To determine if a node corresponds to a discrete variable
GetNodeKind_bn	To determine what kind of node it is

GetNodeDiscrete_bn

`bool_ns GetNodeDiscrete_bn (const node_bn *node)`

Returns TRUE if the variable corresponding to `node` is discrete (digital), and FALSE 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 network.

For example, continuous variables may be discretized by subdividing them into ranges, and discrete variables may provide values in continuous settings.

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 whether a node is discrete; this can only be set when the node is first created. A continuous node may be discretized anytime, by using SetNodeLevels_bn.

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 GetNodeDiscrete_bn returns
IsNodeDeterministic_bn	Returns whether a node is deterministicly related to its parents
GetNodeKind_bn	To determine what kind of node it is (nature, decision, etc.)

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 given by the parent states. The experience is also known as the "number cases", ess, or estimated sample size (see the Learning Networks chapter). It returns `UNDEF_DBL` if no experience has been assigned.

The order of the states in `parent_states` should match the order of the nodes in the list returned by GetNodeParents_bn. `parent_states` can be `NULL` if `node` has no parents.

To cycle through all of possibilities of `parent_states`, see the NeticaEx function NextStates.

See also:

SetNodeExperience_bn
GetNodeProbs_bn

GetNodeFinding_bn

```
state_bn  GetNodeFinding_bn  (const node_bn *node)
```

If a positive finding has been entered for `node`, it returns that finding. If no findings have been entered it returns `NO_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 they 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 with no zeroes then `NO_FINDING` will be returned (i.e. if a number of likelihood findings which exactly contradict and cancel each other are entered for *node*, `NO_FINDING` will be returned).

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 cannot be used on a continuous node, unless the node has first been discretized by giving it a set of levels (for example, using SetNodeLevels_bn).

See also:

GetNodeLikelihood_bn	To get likelihood or negative findings
GetNodeValue_bn	To get a finding for a continuous node
EnterFinding_bn	To enter a finding
RetractNodeFindings_bn	To clear away all findings entered so far for this node

---

GetNodeFuncValue_bn

```
double  GetNodeFuncValue_bn  (const node_bn *node,
                              const state_bn *parent_states)
```

For deterministic nodes. Given a vector of states for the parents of *node*, this returns the value of the node (which is functionally determined by the parent values). If the relationship between *node* and its parents has not yet been created, or if it is probabilistic rather than deterministic, this returns `UNDEF_DBL`.

The order of the states in *parent_states* should match the order of the nodes in the list returned by GetNodeParents_bn. *parent_states* can be `NULL` if *node* has no parents.

It ignores any findings entered in the net.

To cycle through all of possibilities of *parent_states*, see the NeticaEx function NextStates.

See also:

GetNodeProbs_bn	For nondeterministic discrete nodes
IsNodeDeterministic_bn	To discover if a node is deterministic
SetNodeFuncValue_bn	To originally create the relationship
OptimizeDecisions_bn	Also creates a deterministic relationship

GetNodeKind_bn

`nodekind_bn GetNodeKind_bn (const node_bn *node)`

Returns whether *`node`* is a nature, decision, utility or assume node.

The value returned will be one of:

`NATURE_NODE`	The type of node belief networks are normally composed of. 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.
`ASSUME_NODE`	A constant, or "clamped" variable.

`node`

See also:

IsNodeDeterministic_bn	To distinguish between "chance" and "deterministic" nodes
SetNodeKind_bn	To change what kind of node it is
GetNodeDiscrete_bn	To determine if a node corresponds to a discrete variable

GetNodeLevel_bn

```
level_bn  GetNodeLevel_bn  (const  node_bn *node,
                                    state_bn state)
```

Returns the level associated with a range (for continuous nodes), or point value (discrete nodes) of *`node`*. Returns `UNDEF_DBL` if *`node`* does not have a levels list. 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.

Since the usage of levels is a little different for each type of node, we discuss each separately:

***node* is continuous:** (GetNodeDiscrete_bn would return FALSE)

`state` must be between 0 and the number of states of *`node`*, inclusive.

The level returned is the minimum value of the range corresponding to *`state`*, and the maximum value of the range corresponding to *`state`* - 1.

***node* is discrete:** (GetNodeDiscrete_bn would return TRUE)

`state` must be between 0 and one less than the number of states of *`node`*, inclusive.

The level returned is the real quantity associated with *`state`*.

See also:

SetNodeLevels_bn	Sets the level values for all states at once

GetNodeLikelihood_bn

const prob_bn *GetNodeLikelihood_bn (const node_bn *node)

Returns a (nonmodifiable) 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, 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 completely consistent with the likelihood concept, 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 by giving it a set of levels (for example, using SetNodeLevels_bn).

Do not try to modify or free the vector returned.

See also:

GetNodeFinding_bn	Returns a scalar indicating what findings have been entered
GetNodeBeliefs_bn	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

Example:

```
// To temporarily remove all findings from a node and later restore them
//
size_t size = GetNodeNumberStates_bn (node) * sizeof (prob_bn);
char *save = malloc (size);
memcpy (save, GetNodeFinding_bn (node), size);
RetractNodeFindings_bn (node);
... <stuff without the evidence> ...
EnterNodeLikelihood_bn (node, save);                        // restores to original
free (save);
```

GetNodeName_bn

const char *GetNodeName_bn (const node_bn *node)

Returns a (nonmodifiable) C character string which is the name of *node*.

This may be different from *node*'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` or less characters (not including terminating 0).

Do not try to modify or free the string returned.

See also:

NodeNamed_bn	(inverse function) Gets the node given its name
SetNodeName_bn	Change the node's name
GetNodeTitle_bn	Gets the node's title

GetNodeNet_bn

`net_bn *GetNodeNet_bn (const node_bn *node)`

Returns the network that *`node`* is part of. Every node is part of some network.

See also:

DuplicateNodes_bn	Can move nodes from one network to another
GetNetNodes_bn	(inverse function) Get the list of nodes comprising a network
NewNet_bn	Originally created the network

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.

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

Example:

See the NeticaEx.c function NextStates.

GetNodeParents_bn

`const nodelist_bn *GetNodeParents_bn (const node_bn *node)`

Returns a (nonmodifiable) list of the parents of *`node`*. If it has no parents then the 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 routines.

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.

Do not try to modify or free the list returned.

See also:

GetNodeChildren_bn	Get a list of the children

GetNodeProbs_bn

`const prob_bn *GetNodeProbs_bn (const node_bn *node, const state_bn *parent_states)`

This returns the conditional probabilities of *node*, given that its parents are in the states indicated by the *parent_states* vector. 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 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 network, so its value will not change as findings are added or belief updating is done.

`NULL` will be returned if there is no relation associated with node, either by SetNodeProbs_bn or SetNodeFuncValue_bn.

The order of the states in *parent_states* should match the order of the nodes in the list returned by GetNodeParents_bn. *parent_states* can be `NULL` if *node* has no parents.

Do not try to modify or free the array returned.

To cycle through all of possibilities of *parent_states*, see the NeticaEx function NextStates.

See also:

GetNodeBeliefs_bn	Conditioned on findings, but not parents
AbsorbNodes_bn	Can be used to find probabilities conditioned on parents and findings
GetNodeFuncValue_bn	For deterministic nodes
GetNodeExperience_bn	The confidence of the probabilities obtained

Example:

```
// To just get the probability that node is in state, given parent_states
//
double prob = GetNodeProbs_bn (node, parent_states) [state];
```

GetNodeStateName_bn

`const char *GetNodeStateName_bn (const node_bn *node, state_bn state)`

Returns a (nonmodifiable) string which is the name for *state* of *node*, or the empty string (rather than `NULL`) if it does not have a name.

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` or less characters (not including terminating 0).

Do not try to modify or free the string returned.

See also:

StateNamed_bn	(inverse function) Returns the state with a given state name

---

GetNodeTitle_bn

`const char *GetNodeTitle_bn (const node_bn *node)`

Returns a (nonmodifiable) 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".

There is no restriction on the length of the title, or on what characters it might contain.

Do not try to modify or free the string returned.

See also:

GetNodeName_bn	Gets the node's name (limited chars and length, always exists)
SetNodeTitle_bn	Change the node's title

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 none has been.

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.

Pass 0 for *kind*. It is there only for future expandability.

See also:

SetNodeUserData_bn	Attach a user pointer to a node
GetNetUserData_bn	Recover the user pointer attached to a whole network

---

GetNodeValue_bn

`double GetNodeValue_bn (const node_bn *node)`

Returns the real-valued finding which has been entered for *node*, or `UNDEF_DBL` if *node* has no real-valued finding (note: do not use the `UNDEF_DBL` macro until after InitNetica_bn has been called).

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 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.

See also:

GetNodeFinding_bn	To get the discrete finding of a node
EnterNodeValue_bn	To enter a real-valued finding into a node
RetractNodeFindings_bn	To clear away all findings entered into a node

InitNetica_bn

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

This initializes the Netica system. Call it before any other Netica functions except NewNeticaEnviron_bn, or some of the configuration routines, such as ArgumentChecking_ns.

It should be passed the address of a pointer to an environ_ns structure which has been created by NewNeticaEnviron_bn. *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 it if InitNetica_bn is successful, or an error message if it isn't.

It will return a value greater than or equal 0 on success, and a negative value on failure. If it fails, then no other Netica API routines should be called with *env* except CloseNetica_bn.

Notice that *env* is a pointer to a pointer to an `environ_ns`. See the example below.

If a NULL license string was passed to NewNeticaEnviron_bn when *env* was created, InitNetica_bn may also directly print out a short message.

See also:

NewNeticaEnviron_bn	Creates the required environ_ns struct
CloseNetica_bn	Reverses the effects of InitNetica_bn

Example:

```
void main (void){
    char mesg[MESG_LEN_ns];
    environ_ns *env;
    int res;

    env = NewNeticaEnviron_bn (NULL);    // substitute your license string for NULL if
                                                       desired
    res = InitNetica_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);
    }
```

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 network and current findings entered into the network, otherwise it returns FALSE.

See also:

GetNodeBeliefs_bn Updates the beliefs (if they aren't already)

Length Node List_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` pointers, if you set some of them to `NULL`, or you created the list using NewNodeList_bn with nonzero length and never completely filled it. Netica functions that return node lists will never return a node list having `NULL` pointers unless that possibility is specifically mentioned in the documentation for that function.

Also remember that if it's a list that you made up, some of the nodes may be duplicated in the list, so you won't get the number of unique nodes.

Link Named_bn

`int LinkNamed_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. This is the same index as would be used to find the parent of the link in the node list returned by GetNodeParents_bn (node).

This is only applicable to links of *`node`*, 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.

See also:

GetNodeLinkName_bn (inverse function) Returns the name of a link given its index

---

New Netica Environ_bn

`environ_ns *NewNeticaEnviron_bn (const char *license)`

Call this to create a new `environ_ns` structure that it is suitable for passing to InitNetica_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.

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 return value of this routine, and then pass it to any routine that requires it.

Netica will make a copy of *`license`*; it won't modify or free the passed string.

Example:

See InitNetica_bn.

NewNet_bn

```
net_bn *NewNet_bn (const char *name, environ_ns *env)
```

Creates and returns a new network, initially having no nodes (use NewNode_bn to add nodes). If just nature nodes are added to this network, it will be a belief network, while if any decision nodes are added it will be a decision network.

The name of the new network will be *name*. It must be a legal IDname (see IDname in the index), which means it must have `NAME_MAX_ns` or less 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 network, you should pass it to FreeNet_bn.

See also:

FreeNet_bn	(reverse operation) Frees the memory used by a net
NewNode_bn	Add nodes to the new network

NewNode_bn

```
node_bn *NewNode_bn (const char *name, int num_states,
                                  net_bn *net)
```

Creates and returns a new node for network *`net`*.

If the node is for a variable which is discrete in the real world, pass the number of states it can take on for *`num_states`*.

If the node is for a variable which is continuous in the real world, always 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 a legal IDname (see IDname in the index), which means it must have `NAME_MAX_ns` or less 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.

The node will start off as a nature node (kind = `NATURE_NODE`), but it may be changed by calling SetNodeKind_bn.

Example:

See SetNodeLevels_bn for creating a discretized node for a continuous variable.

See also:

DuplicateNodes_bn	Duplicate nodes, or transfer them from another network
SetNodeKind_bn	Set what kind of node it is
SetNodeLevels_bn	Set the number of states for continuous variables
GetNodeDiscrete_bn	Determine if it was created as a continuous variable node
GetNodeNumberStates_bn	Recover *`num_states`*

DeleteNode_bn	(reverse operation) Removes the node from its network and frees the memory it uses

NewNodeList_bn

nodelist_bn *NewNodeList_bn (int length, environ_ns *env)

Creates and returns a new node list with *length* entries, each filled with NULL.

You can fill the entries using SetNthNode_bn, and you can change the length using AddNodeToList_bn or RemoveNthNode_bn.

If the system runs out of memory, NULL will be returned and an error generated.

Use FreeNodeList_bn to free the list when you are done with it.

WARNING: Do not pass it to the Standard C 'free' function.

Your software may use the returned list until disposing of it, or calling CloseNetica_bn, without fear that it will be modified by Netica (unless, of course, you deliberately pass it to a routine designed to change it).

See also:

DupNodeList_bn	Make a copy of an existing list
FreeNodeList_bn	Free the new list made
AddNodeToList_bn	Increases a list's length by adding a node

NodeNamed_bn

node_bn *NodeNamed_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.

name can be any string; it need not be a legal IDname (of course if it isn't, NULL will be returned).

If you have a list of nodes (nodelist_bn), and you wish to retrieve the node from it having a given name, then providing all its nodes come from the same network, you can just use NodeNamed_bn on that network (it would actually be faster than searching the list, because a symbol table is maintained for the network). Then, if necessary, you can use PositionInNodeList (see NthNode_bn) to make sure the found node is in the list.

See also:

GetNodeName_bn	(inverse function) Returns the name of the node

Example:

```
// Like NodeNamed_bn, but generates an error if the name doesn't exist.
//
node_bn *NodeNamed (char *node_name, net_bn *net){
    node_bn *node = NodeNamed_bn (node_name, net);
    if (node == NULL)
        ReportError_ns (env, 0, ERROR_ERR,
                    "NodeNamed: There is no node named '%s' in net '%s'",
```

```
                        node_name, GetNetName_bn (net));
    return node;
}
```

NthNode_bn

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

Returns the node at position *index* of 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.

For the inverse function (finding the index given the node) see the routine PositionInNodeList in the example below.

See also:

RemoveNthNode_bn	Remove the node from the list while obtaining it
LengthNodeList_bn	Find maximum node index

Example:

```
// To return the index of node in the list nodes (first is 0).
// This can also be used to test if node is a member of nodes,
// since this returns -1 iff it isn't.
//
int PositionInNodeList (const node_bn *node, const nodelist_bn *nodes){
    int i, num = LengthNodeList_bn (nodes);
    for (i = 0;  i < num;  i++)
        if (NthNode_bn (nodes, i) == node)
            return i;
    return -1;
}
```

OptimizeDecisions_bn

`void OptimizeDecisions_bn (const nodelist_bn *nodes)`

Finds the optimal decision function for each decision node in *nodes*, and attaches it to the node. Each decision function provides the optimal decision for the node (to maximize expected utility) for each configuration of parent states.

It is okay if *nodes* contains some nodes that aren't decision nodes; OptimizeDecisions_bn will act as though they weren't in the list. All of the nodes should come from the same network.

You can use GetNodeFuncValue_bn to "read out" the decisions found.

See also:

AbsorbNodes_bn	Another way to find optimal decisions
GetNodeFuncValue_bn	To get the decision functions found

Example:

```
// To optimize all the decision nodes in net.
//
OptimizeDecisions_bn (GetNetNodes_bn (net));
```

RandomCase_bn

```
int  RandomCase_bn  (const nodelist_bn *nodes,  int method,
                                double num)
```

Generates a random case for *nodes* (i.e., positive findings for each of them), by sampling from a probability distribution matching that of the network containing *nodes* , and consistent with any findings already entered in *nodes*.

Pass 0 for *method*; it is only for future expandability.

num indicates how much time to allocate for the task (in relative units), and a negative quantity will be returned if there was not enough time to finish, or it failed for some other reason.

ReadCase_bn

```
void  ReadCase_bn  (caseposn_bn *case_posn,  stream_ns *file,
                             const nodelist_bn *nodes,
                             long *ID_num,  double *freq)
```

Reads from *file* (which is usually a file created using WriteCase_bn) to recover the findings stored in it, and enter them into *nodes*.

WARNING: It does not retract findings that are already in *net*, and will even generate errors if findings in the file are inconsistent with findings already in *net*. So you probably want to call RetractAllFindings_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 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 nonNULL         \  - file has no cases          returns with *case_posn = NO_MORE_CASES
*case_posn = FIRST_CASE   /  - otherwise                  reads first case & sets *case_posn

case_posn nonNULL         \  - all cases read             returns with *case_posn = NO_MORE_CASES
*case_posn = NEXT_CASE    /  - otherwise                  reads next case & sets *case_posn

case_posn nonNULL           \                             generates error
*case_posn = NO_MORE_CASES /

case_posn nonNULL         \  - indicated case is in file    reads indicated case
*case_posn = case         /  - indicated case isnt in file  generates error
```

If *ID_num* is nonNULL, 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 nonNULL, then on return **freq* will be set to the frequency (i.e., multiplicity) of the case, or -1 if it doesn't have one.

It only reads evidence and values into the nodes of *nodes*. Other nodes in the network will not have any new findings entered, even if findings for them appear in the file. It is okay if *nodes* is empty. *nodes* must all be from the same network.

Doesn't modify or take *nodes*.

See also:

WriteCase_bn	Save it so that ReadCase_bn can read it back
RetractAllFindings_bn	You may want to call this first

ReadNet_bn

`net_bn *ReadNet_bn (stream_ns *file, bool_ns readvisual)`

Reads a network from *file*, and returns the network read, or `NULL` if reading was impossible. Even if this routine returns a non`NULL` value, you should check if it generated any errors, since it may report on a problem but return the best network it can.

If *readvisual* is FALSE, then any information about the visual display of the net for use by the graphical editor is ignored.

See also:

FileNamed_ns	Generates the required stream_ns
WriteNet_bn	Saves a network to file in a format understood by ReadNet_bn
ShowNet_bn	Display the network to the user and allow graphical editing

Example:

See WriteNet_bn

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 INT_MAX, 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.

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

Example:

```
// To remove the first occurrence of some node from the list
//
void RemoveNodeFromList (node_bn *node, nodelist_bn *nodes){
    int i, num = LengthNodeList_bn (nodes);
```

```
    for (i = 0;  i < num;  i++)
        if (NthNode_bn (nodes, i) == node){
            RemoveNthNode_bn (nodes, i);
            return;
        }
}
```

Example:

```
// 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, INT_MAX);
    SetNthNode_bn (nodes, index, lastnode);
}
```

Report Error_ns

```
report_ns  *ReportError_ns  (environ_ns *env,  int number,
                             errdanger_ns danger,
                             const char *mesg)
```

Generates a new error report with error number being `number`, danger level `danger`, and error message `mesg`. The new report is returned and registered with environment `env`.

`danger` must be one of `NOTHING_ERR`, `REPORT_ERR`, `NOTICE_ERR`, `WARNING_ERR`, `ERROR_ERR`, `XXX_ERR` (for a description of these see ErrorDanger_ns).

`number` must be between 0 and 999, inclusive.

The Netica API communicates to you every error it detects via a report_ns registered with the environment. This routine is provided so that you can make your own report_ns registered with the environment, allowing for a simpler and more consistent error handling in your software.

See also:

ClearError_ns	(reverse operation) Removes an error report from the system
GetError_ns	Obtain the error report from the environment
ErrorMessage_ns	Recover the error message from the report
ErrorNumber_ns	Recover the error number from the report
ErrorDanger_ns	Recover the danger level of the error from the report

Example:

```
// Like ReportError_ns, but with printf style arguments for the message.
// You must be very careful that your error message length is limited,
//   so that it doesn't run over the declared buffer size.
//
report_ns *ReportError (environ_ns *env, int number, errdanger_ns danger,
                                                const char *mesg, ...){
    va_list ap;
    char buf[400];
    va_start (ap, mesg);
    vsprintf (buf, mesg, ap);
    va_end (ap);
    return  ReportError_ns (env, number, danger, buf);
    }
```

RetractAllFindings_bn

```
void  RetractAllFindings_bn  (net_bn *net)
```

Retracts all findings (i.e. the current case) from all the nodes in network *net*.

If the network is an auto-update network (see SetNetAutoUpdate_bn), then a belief updating will be done to reflect the removal of findings as soon as this is called, otherwise GetNodeBeliefs_bn must be called before belief updating is done.

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 regular findings, negative findings, likelihood findings and the discovered value of a continuous node.

If *node* does not have any findings, calling this will have no effect.

If the network is an auto-update network (see SetNetAutoUpdate_bn), then a belief updating will be done to reflect the removal of findings as soon as this is called, otherwise GetNodeBeliefs_bn must be called before belief updating is done.

See also:

RetractAllFindings_bn	To remove the findings from all nodes in the network
EnterFinding_bn	To enter a finding for a node

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*.

If it is not possible to do the reversal, an error will be generated, and network will not be changed.

If both *parent* and *child* are nature nodes, and have conditional probability tables, these will be modified so that the reversal has no effect on the joint probability represented by the network. Note that this may mean adding links from the parents of *child* to *parent*, and from the parents of *parent* to child. If this is not desired then use DeleteLink_bn, followed by an AddLink_bn in the reverse direction.

SetLinkName_bn

```
void  SetLinkName_bn  (node_bn *node,  int link_index,
                                const char *link_name)
```

Names the link entering *node* from its *link_index* th parent to be *link_name*.

link_index corresponds to the ordering of the parents obtained by GetNodeParents_bn (with the first parent having *link_index* = 0). The reason that a 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*.

link_name must be a legal IDname (see IDname in the index), which means it must have `NAME_MAX_ns` or less 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 *link_name*.

link_name must be different from the name of any other links entering *node*, and must be different from the names of any parents of *node* which are connected to *node* by an unnamed link.

The purpose of a link name is to provide a local naming for a link, which is maintained even when a node is disconnected from a parent, and maybe later reconnected to another parent.

When links are first created, they are unnamed, and remain so until this routine 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 *link_name*; it won't modify or free the passed string.

See also:

GetLinkName_bn	Get an existing link name
GetNodeParents_bn	Gets the actual parents of the links (e.g. to find their names or determine their numbering)
SwitchNodeParent_bn	Connect the "input" up with a parent node

SetNetAutoUpdate_bn

```
void  SetNetAutoUpdate_bn  (net_bn *net,  int autoupdate)
```

Pass an *autoupdate* of 1 to have the new beliefs calculated immediately after entering any findings, or 0 to inhibit this (in which case you would call GetNodeBeliefs_bn to do the updating).

The reason for inhibiting automatic updating is that it is time consuming, and you may want to enter many findings before doing it. However, an advantage to updating after entering each finding, is that as each new finding is entered, it will be checked for consistency with the findings already entered.

If you are turning auto-updating on, and it currently needs doing, then updating will be done before this function returns.

See also:

GetNodeBeliefs_bn	Forces a belief update if required
GetNetAutoUpdate_bn	Returns the value that SetNetAutoUpdate_bn sets

SetNetComment_bn

`void SetNetComment_bn (net_bn *net, const char *comment)`

This stores a duplicate of the 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, etc.

Information pertaining 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.

See also:

GetNetComment_bn	Retrieve the comment associated with *`net`*
SetNodeComment_bn	Set a comment for a particular node

Example:

```
// Put string addon at the end of the existing comment for net
//
const char *origcomment = GetNetComment_bn (net);
size_t origlength = strlen (origcomment);
char *comment = malloc (origlength + strlen (addon) + 1);
strcpy (comment, origcomment);
strcpy (comment + origlength, addon);
SetNetComment_bn (comment, net);
free (comment);                    // that's right, don't free origcomment
```

SetNetElimOrder_bn

```
void  SetNetElimOrder_bn  (net_bn *net,
                              const nodelist_bn *elim_order)
```

Records the list of nodes *`elim_order`* with *`net`* to be used as its "elimination order" the next time *`net`* is compiled.

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.

Calling this routine has no effect on the current triangulation; it only takes action during the next compilation.

`elim_order` must include all the nodes of *`net`* without any duplication (except it should not include any nodes whose kind is "ASSUME").

Netica will make a copy of *`elim_order`*; it won't modify or free the passed list.

See also:

GetNetElimOrder_bn	Get the elimination order currently being used
CompileNet_bn	Do or redo the compilation to use the new elimination order
AddNodeToList_bn	Can use with NewNodeList_bn to build the list

---

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 or less 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.

See also:

SetNetTitle_bn
GetNetName_bn

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 network).

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.

See also:

SetNetName_bn	
GetNetTitle_bn	
SetNodeTitle_bn	Set the title for a particular node

SetNetUserData_bn

```
void  SetNetUserData_bn  (net_bn *net,  int kind,  void *data)
```

Attaches the data pointed to by *data* to the network *net*. Only the program using the Netica API 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 there only for future expandability.

See also:

GetNetUserData_bn	Get the user pointer of a net
SetNodeUserData_bn	Set the user pointer of a particular node

SetNodeComment_bn

```
void  SetNodeComment_bn  (node_bn *node,
                           const char *comment)
```

This stores a duplicate of the 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, why its important, notes for future changes, etc.

Information pertaining to the network as a whole should not be placed here, but rather in the network'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.

See also:

GetNodeComment_bn	Retrieve the comment associated with *node*
SetNetComment_bn	Set a comment for the whole network

Example:

To add to an existing comment, see SetNetComment_bn.

SetNodeExperience_bn

```
void  SetNodeExperience_bn  (node_bn *node,
                              const state_bn *parent_states,
                              double experience)
```

This is for deterministic nodes. It tells Netica that when each parent has the state indicated in the vector *parent_states*, the value of *node* is *func_value*.

Sets the conditional probabilities of *node*, given that its parents have the states indicated by *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*, and consist of the conditional probabilities:

This returns the conditional probabilities of *node*, given that its parents are in the states indicated by the *parent_states* vector. The length of *parent_states* must be the number of parents

This is to associate a degree of experience with each belief vector of *node*'s conditional probability table (see the chapter on Learning Networks). It sets the amount of experience with the situation in which each parent has the state indicated by the vector *parent_states*, to *experience*.

The order of the states in `parent_states` should match the order of the nodes in the list returned by GetNodeParents_bn. `parent_states` can be `NULL` if *node* has no parents.

To cycle through all of possibilities of `parent_states`, see the NeticaEx function NextStates.

See also:

CaseRevisesProbs_bn
CaseFileRevisesProbs_bn
FadeProbs_bn
SetNodeProbs_bn
GetNodeExperience_bn

SetNodeFuncValue_bn

```
void  SetNodeFuncValue_bn  (node_bn *node,
                             const state_bn *parent_states,
                             double func_value)
```

This is for deterministic nodes. It tells 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. `parent_states` can be `NULL` if `node` has no parents.

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 calling to see if the probabilities were successfully set.

To cycle through all of possibilities of `parent_states`, see the NeticaEx function NextStates.

See also:

SetNodeProbs_bn	To use if `node` isn't deterministic
GetNodeFuncValue_bn	Retrieve the function values

Example:

```
// This routine is similar to SetNodeProbs; see the comment for it.

#include <stdarg.h>
#define ARR_SIZE 20

void SetNodeFuncValue (node_bn *node, double value, ...){
    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){
        ReportError_ns (env, 0, XXX_ERR, "SetNodeFuncValue: Array size defined too
                                                        small");
        return;
    }
    va_start (ap, value);
    for (pn = 0;  pn < numparents;  pn++)
        parent_states[pn] = StateNamed_bn (va_arg (ap, char*), NthNode_bn (parents, pn));
    va_end (ap);
    SetNodeFuncValue_bn (node, parent_states, value);
}
```

```
#undef ARR_SIZE
```

Example:

```
// This doesn't use SetNodeFuncValue_bn, but it is useful for setting parent_states.
// This 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;
    }
```

SetNodeKind_bn

`void SetNodeKind_bn (node_bn *node, nodekind_bn kind)`

Sets whether `node` is a nature, decision, utility or assume node.

`kind` must be one of:

NATURE_NODE	The type of node belief networks are normally composed of. 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.
ASSUME_NODE	A constant, or "clamped" variable.

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 or discrete node being transformed to a utility node.

See also:

GetNodeKind_bn	To retrieve what kind of node it is

SetNodeLevels_bn

`void SetNodeLevels_bn (node_bn *node, int num_states, const level_bn *levels)`

Sets the levels list of `node` to `levels`. If `node` is continuous, this can be used to discretize it to `num_states` states.

The levels list is a list of real numbers used to translate from a discrete node to a continuous value, or from a continuous node to a discrete value. 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.

Conversely, if it is discrete, we may want a mapping from its state (represented as an integer), to a measurable value. For example, in a bang-bang control system, state 0 ("Off") may map to 0 Volts, and state 1 ("On") may map to 6.2 Volts.

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:** (GetNodeDiscrete_bn would return `FALSE`)

(these nodes can be created by NewNode_bn with `num_states` = 0)

The levels list is one longer than the number of states, and monotonicly ascends or descends.

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 InitNetica_bn has been called).

Once `node` has been given the levels list, Netica can translate a continuous value *val* for node to a discrete state *st*, by choosing *st* so that:

levels [*st*] <= *val* < levels [*st* + 1] or

levels [*st*] > *val* >= levels [*st* + 1]

A discrete state *st* can also be translated to the range: [levels[*st*], levels[*st* + 1]).

***node* is discrete:** (GetNodeDiscrete_bn would return `TRUE`)

There is one entry in the levels list for each state.

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.

See also:

GetNodeLevel_bn	Recover the levels that have been set
NewNode_bn	Must be called with `num_states` = 0 to make continuous node

Example:

```
// Here we make a continuous node and then discretize it into
// 3 states.
//
level_bn levels[4];                       // 1 more than the number of states
node_bn *node = NewNode_bn ("n1", 0, net);  // must pass 0 for num_states to create a node
                                                       for a continuous variable
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
SetNodeStateName_bn (node, 0, "low");     // naming the states is optional
```

```
SetNodeStateName_bn (node, 1, "medium");
SetNodeStateName_bn (node, 2, "high");   // but if we name one, we should name them all
```

Example:

```
// Here we make a 3-state discrete node and then give it
// levels to provide real values to its children.
//
level_bn levels[3];                      // 1 element for each state
node_bn *node = NewNode_bn ("n2", 3, net);  // discrete, with 3 states
levels[0] = -3.5;                        // state 0 gives -3.5
levels[1] = 0;
levels[2] = 5.5;
SetNodeLevels_bn (node, 3, levels);      // set the levels
SetNodeStateName_bn (node, 0, "low");    // naming the states is optional
SetNodeStateName_bn (node, 1, "medium");
SetNodeStateName_bn (node, 2, "high");   // but if we name one, we should name them all
```

SetNodeName_bn

`void SetNodeName_bn (node_bn *node, const char *name)`

Changes the name of *`node`* to *`name`*.

`name` must be a legal IDname (see IDname in the index), which means it must have `NAME_MAX_ns` or less characters, all of which are letters, digits or underscores, and it must be different from the name of every other node already in the same network.

Netica will make a copy of *`name`*; it won't modify or free the passed string.

See also:

SetNetTitle_bn
GetNetName_bn

SetNodeProbs_bn

```
void SetNodeProbs_bn (node_bn *node,
                      const state_bn *parent_states,
                      const prob_bn *probs)
```

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`*, 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. *`parent_states`* can be `NULL` if *`node`* has no parents.

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 calling to see if the probabilities were successfully set.

To cycle through all of possibilities of `parent_states`, see the NeticaEx function NextStates.

See also:

SetNodeFuncValue_bn	For deterministic nodes
GetNodeProbs_bn	Retrieve the probabilities
CaseRevisesProbs_bn	Revise the probabilities using the case currently entered
CaseFileRevisesProbs_bn	Revise the probabilities using a file of cases
FadeProbs_bn	Adjust the probabilities for a changing world
SetNodeExperience_bn	Associate a degree of experience with the probabilities

Example:

```
/*__________________________________________________________________________ SetNodeProbs
This routine 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.
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 routine.
___________*/

#include <stdarg.h>
#define ARR_SIZE 20

void SetNodeProbs (node_bn *node, ...){
    state_bn parent_states[ARR_SIZE];
    prob_bn probs[ARR_SIZE];
    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){
        ReportError_ns (env, 0, XXX_ERR, "SetNodeProbs: Array size defined too small");
        return;
    }
    va_start (ap, node);
    for (pn = 0;  pn < numparents;  pn++)
        parent_states[pn] = StateNamed_bn (va_arg (ap, char*), NthNode_bn (parents, pn));
    for (state = 0;  state < numstates;  state++)
        probs[state] = va_arg (ap, double);
    va_end (ap);
    SetNodeProbs_bn (node, parent_states, probs);
}

#undef ARR_SIZE
```

SetNodeStateName_bn

`void SetNodeStateName_bn (node_bn *node, state_bn state, const char *state_name)`

Names *`state`* of *`node`* with *`state_name`*.

`state` must be between 0 and one less than the number that would be returned by GetNodeNumberStates_bn, inclusive.

`state_name` must be a legal IDname (see IDname in the index), which means it must have `NAME_MAX_ns` or less 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 remove a state's name, pass `NULL` (not the empty string) for *`state_name`*. If you remove one, you should remove them all.

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.

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.

See also:

GetNodeStateName_bn	Get an existing state name if there is one
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
SetNodeStateName_bn (node, 0, "True");
SetNodeStateName_bn (node, 1, "False");
```

Example:

```
// This wraps several calls to SetNodeStateName_bn into one.
// For instance to give the node 'Weather' its 3 state names you could
// call:
//     SetNodeStateNames (Weather, "sunny", "cloudy", "rainy");
//
#include <stdarg.h>

void SetNodeStateNames (node_bn *node, ...){
    state_bn state, numstates = GetNodeNumberStates_bn (node);
    va_list ap;
    va_start (ap, node);
    for (state = 0;  state < numstates;  state++)
        SetNodeStateName_bn (node, state, va_arg (ap, char*));
    va_end (ap);
}
```

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.

See also:

SetNodeName_bn	
GetNodeTitle_bn	
SetNetTitle_bn	Set the title for the whole net

SetNodeUserData_bn

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

Attaches the data pointed to by *data* to *node*. Only the program using the Netica API 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 there only for future expandability.

See also:

GetNodeUserData_bn	Recover the user pointer attached to a node
SetNetUserData_bn	Attach a user pointer to the whole network

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 its 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.

See also:

AddNodeToList_bn	Insert a node in a list, increasing its length
LengthNodeList_bn	Find maximum node index

StateNamed_bn

```
state_bn  StateNamed_bn  (const char *name,
                               const node_bn *node)
```

Returns the state number of the state whose name is `name`, or `UNDEF_STATE` if there isn't one.

This is only applicable to states of `node`, another node may have a state with the same name, but a different state number.

Netica won't modify or free the passed `name` string.

See also:

GetNodeStateName_bn	(inverse function) Returns the state name of a given state

SwitchNodeParent_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 the relationship between `node` and its parents.

The new parent must be compatible with the old (e.g. same number of states).

`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.

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 number parents - 1, and the ordering can be obtained using GetNodeParents_bn. The reason that the parent number is passed instead of the actual parent is to handle situations with more than 1 link between the same 2 nodes.

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

WriteCase_bn

```
caseposn_bn  WriteCase_bn  (const nodelist_bn *nodes,
                               stream_ns *file,  long ID_num,
                               double freq)
```

Saves in `file` the set of findings currently entered in `net`, so that later they can be read back with ReadCase_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.

Pass -1 for *ID_num* and/or *freq* if you do not want them included.

It returns the file position of the new case (which can later be passed to ReadCase_bn).

It only saves findings and values from the nodes of *nodes*, and if the file already exists, it wont save findings or values from any of *nodes* that were not included in the node list used to first construct the file.

Doesn't modify or free *nodes*.

See also:

ReadCase_bn	Read back the case that WriteCase_bn saves
WriteNet_bn	Save the whole network

WriteNet_bn

void WriteNet_bn (const net_bn *net, stream_ns *file)

Saves *net* to a file specified by *file* (which can also indicate which directory to put the network in - see FileNamed_ns).

If *file* already exists it is overwritten. The network 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.

See also:

FileNamed_ns	Generates the required stream_ns
ReadNet_bn	Reads back the network saved
WriteCase_bn	Just save the findings entered so far as a case

Example:

```
// Sets net2 to a copy of net1, but without its visual information
//
file = FileNamed_ns ("temp.dnet", env);
WriteNet_bn (net1, file);
net2 = ReadNet_bn (file, FALSE);
if ((error = GetError_ns (env, WARNING_ERR, NULL)) != NULL)
    fprintf (stderr, "%s\n", ErrorMessage_ns (error));
```

9 Netica.h Header File

```
/*
 * Header file for Netica API, version 1.01.
 *
 * Copyright (C) 1992-1995 by Norsys Software Corp.  All rights reserved.
 */

#ifndef __Netica_h
#define __Netica_h

#ifdef __cplusplus
extern "C" {
#endif

#include <limits.h>    /* from the ANSI Standard C library */
#include <stdio.h>


#ifdef FALSE
typedef enum {FALSE_ns=0, TRUE_ns} bool_ns;
#else
typedef enum {FALSE=0, TRUE} bool_ns;
#endif

#define  INFINITY_ns  inf_dbl_ns
#define  UNDEF_DBL    undef_dbl_ns

#define  MESG_LEN_ns  256
#define  NAME_MAX_ns   30
enum    {QUERY_ns  =   -1};

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

typedef enum {NO_CHECK = 23, QUICK_CHECK, REGULAR_CHECK, COMPLETE_CHECK} checking_ns;

typedef enum {NATURE_NODE, ASSUME_NODE, DECISION_NODE, UTILITY_NODE} nodekind_bn;

enum {IMPOSS_STATE = -4, UNDEF_STATE};

enum {NO_FINDING = -8, NEGATIVE_FINDING, LIKELIHOOD_FINDING};   /* for GetNodeFinding_bn */

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

typedef long    state_bn;
typedef float   prob_bn;
typedef double  level_bn;
typedef long    caseposn_bn;
```

```
typedef struct net__bn       net_bn;
typedef struct node__bn      node_bn;
typedef struct nodelist__bn  nodelist_bn;
typedef struct environ__ns   environ_ns;
typedef struct report__ns    report_ns;
typedef struct stream__ns    stream_ns;

extern const double inf_dbl_ns;    /* use INFINITY_ns in your software */
extern const double undef_dbl_ns;  /* use UNDEF_DBL   in your software */
extern double BaseExperience_bn;


environ_ns *NewNeticaEnviron_bn (const char *license, int version);
int InitNetica_bn (environ_ns **envv, char *mesg);
int CloseNetica_bn (environ_ns *env, char *mesg);
int ArgumentChecking_ns (int setting, environ_ns *env);
int GetNeticaVersion_bn (environ_ns *env, const char **version);

report_ns *GetError_ns (environ_ns *env, errdanger_ns danger, report_ns *after);
long ErrorNumber_ns (const report_ns *error);
const char *ErrorMessage_ns (const report_ns *error);
errdanger_ns ErrorDanger_ns (const report_ns *error);
void ClearError_ns (report_ns *error);
void ClearErrors_ns (environ_ns *env, errdanger_ns danger);
report_ns *ReportError_ns (environ_ns *env, int number, errdanger_ns danger, const char *mesg);

stream_ns *FileNamed_ns (const char *filename, environ_ns *env);
void WriteNet_bn (const net_bn *net, stream_ns *file);
net_bn *ReadNet_bn (stream_ns *file, bool_ns readvisual);
caseposn_bn WriteCase_bn (const nodelist_bn *nodes, stream_ns *file, long ID_num, double freq);
void ReadCase_bn (caseposn_bn *case_posn, stream_ns *file, const nodelist_bn *nodes, long *ID_num,
                                double *freq);

void EnterFinding_bn (node_bn *node, state_bn state);
void EnterFindingNot_bn (node_bn *node, state_bn state);
void EnterNodeValue_bn (node_bn *node, double value);
void EnterNodeLikelihood_bn (node_bn *node, const prob_bn *likelihood);
state_bn GetNodeFinding_bn (const node_bn *node);
const prob_bn *GetNodeLikelihood_bn (const node_bn *node);
double GetNodeValue_bn (const node_bn *node);
void RetractNodeFindings_bn (node_bn *node);
void RetractAllFindings_bn (net_bn *net);

void CompileNet_bn (net_bn *net);
bool_ns IsBeliefUpdated_bn (const node_bn *node);
const prob_bn *GetNodeBeliefs_bn (node_bn *node);
double CaseProbability_bn (net_bn *net);
int RandomCase_bn (const nodelist_bn *nodes, int method, double num);

nodelist_bn *NewNodeList_bn (int length, environ_ns *env);
void AddNodeToList_bn (node_bn *node, nodelist_bn *nodes, int index);
node_bn *RemoveNthNode_bn (nodelist_bn *nodes, int index);
int LengthNodeList_bn (const nodelist_bn *nodes);
node_bn *NthNode_bn (const nodelist_bn *nodes, int index);
void SetNthNode_bn (nodelist_bn *nodes, int index, node_bn *node);
nodelist_bn *DupNodeList_bn (const nodelist_bn *nodes);
void FreeNodeList_bn (nodelist_bn *nodes);

void CaseRevisesProbs_bn (const nodelist_bn *nodes, int updating, double degree);
void CaseFileRevisesProbs_bn (stream_ns *file, const nodelist_bn *nodes, int updating, double degree);
void FadeProbs_bn (node_bn *node, double degree);

void ReverseLink_bn (node_bn *parent, node_bn *child);
void AbsorbNodes_bn (nodelist_bn *nodes);
void OptimizeDecisions_bn (net_bn *net);
```

```
net_bn *NewNet_bn (const char *name, environ_ns *env);
void FreeNet_bn (net_bn *net);
node_bn *NewNode_bn (const char *name, int num_states, net_bn *net);
nodelist_bn *DuplicateNodes_bn (const nodelist_bn *nodes, net_bn *new_net);
void DeleteNode_bn (node_bn *node);
int AddLink_bn (node_bn *parent, node_bn *child);
void DeleteLink_bn (int link_index, node_bn *child);
void SwitchNodeParent_bn (int link_index, node_bn *node, node_bn *new_parent);

void SetNetName_bn (net_bn *net, const char *name);
void SetNetAutoUpdate_bn (net_bn *net, int autoupdate);
void SetNetElimOrder_bn (net_bn *net, const nodelist_bn *elim_order);
void SetNetTitle_bn (net_bn *net, const char *title);
void SetNetComment_bn (net_bn *net, const char *comment);
void SetNetUserData_bn (net_bn *net, void *data, int kind);

void SetNodeName_bn (node_bn *node, const char *name);
void SetNodeKind_bn (node_bn *node, nodekind_bn kind);
void SetNodeStateName_bn (node_bn *node, state_bn state, const char *state_name);
void SetNodeLevels_bn (node_bn *node, int num_states, const level_bn *levels);
void SetLinkName_bn (node_bn *node, int link_index, const char *link_name);
void SetNodeProbs_bn (node_bn *node, const state_bn *parent_states, const prob_bn *probs);
void SetNodeFuncValue_bn (node_bn *node, const state_bn *parent_states, double func_value);
void SetNodeExperience_bn (node_bn *node, const state_bn *parent_states, double experience);
void DeleteNodeRelation_bn (node_bn *node);
void SetNodeTitle_bn (node_bn *node, const char *title);
void SetNodeComment_bn (node_bn *node, const char *comment);
void SetNodeUserData_bn (node_bn *node, void *data, int kind);

const char *GetNetName_bn (const net_bn *net);
const nodelist_bn *GetNetNodes_bn (const net_bn *net);
node_bn *NodeNamed_bn (const char *name, const net_bn *net);
int GetNetAutoUpdate_bn (const net_bn *net);
const nodelist_bn *GetNetElimOrder_bn (const net_bn *net);
const char *GetNetTitle_bn (const net_bn *net);
const char *GetNetComment_bn (const net_bn *net);
void *GetNetUserData_bn (const net_bn *net, int kind);

net_bn *GetNodeNet_bn (const node_bn *node);
const char *GetNodeName_bn (const node_bn *node);
nodekind_bn GetNodeKind_bn (const node_bn *node);
bool_ns IsNodeDeterministic_bn (const node_bn *node);
bool_ns GetNodeDiscrete_bn (const node_bn *node);
int GetNodeNumberStates_bn (const node_bn *node);
const char *GetNodeStateName_bn (const node_bn *node, state_bn state);
state_bn StateNamed_bn (const char *name, const node_bn *node);
level_bn GetNodeLevel_bn (const node_bn *node, state_bn state);
const nodelist_bn *GetNodeParents_bn (const node_bn *node);
const nodelist_bn *GetNodeChildren_bn (const node_bn *node);
const char *GetLinkName_bn (const node_bn *node, int link_index);
int LinkNamed_bn (const char *name, const node_bn *node);
const prob_bn *GetNodeProbs_bn (const node_bn *node, const state_bn *parent_states);
double GetNodeFuncValue_bn (const node_bn *node, const state_bn *parent_states);
double GetNodeExperience_bn (const node_bn *node, const state_bn *parent_states);
const char *GetNodeTitle_bn (const node_bn *node);
const char *GetNodeComment_bn (const node_bn *node);
void *GetNodeUserData_bn (const node_bn *node, int kind);

#ifdef __cplusplus
}
#endif
#endif  /* __Netica_h */
```

10 Bibliography

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

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

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, Chichester.

Neapolitan, Richard E. (1990) *Probabilistic Reasoning in Expert Systems: Theory and Algorithms* , John Wiley & Sons, New York.

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

Qi, Runping (1994) *Decision graphs: Algorithms and applications to influence diagram evaluation and high-level path planning under uncertainty*, PhD Thesis, Dept. of Computer Science, Univ. of British Columbia, Canada. Also Tech. Report 94-27, Dept. of Computer Science, Univ. of British Columbia.

Qi, Runping and David Poole (1995) "New method for influence diagram evaluation" in *Computational Intelligence*, **11**(3).

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, Amsterdam.

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.

Zhang, Lianwen (Nevin) (1993) *A computational theory of decision networks*, PhD thesis, Dept. of Computer Science, Univ. of British Columbia, BC, Canada. Also released as Tech. Report 94-8, Dept. of Computer Science, Univ. of British Columbia, BC, Canada.

Zhang, Lianwen (Nevin), Runping Qi and David Poole (1994) "A computational theory of decision networks" in *International Journal of Approximate Reasoning*, **11**(2), 83-158.

11 Netica API Function Groups

System

NewNeticaEnviron_bn	Creates a new environment to pass to InitNetica_bn
InitNetica_bn	Initializes the Netica system
CloseNetica_bn	Signals an end to using Netica system, and frees all possible resources (e.g. memory)
ArgumentChecking_ns	Adjusts the amount that Netica functions check their arguments
GetNeticaVersion_bn	Gets the software version of Netica currently running

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
ErrorDanger_ns	Returns the danger 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
ReportError_ns	Make your own error report using Netica
ArgumentChecking_ns	Adjusts the amount that Netica functions check their arguments

File Operations

FileNamed_ns	Returns a stream for the file with the given name
WriteNet_bn	Saves a network to a file
ReadNet_bn	Reads a network from a file
WriteCase_bn	Saves a network's current set of findings to a file
ReadCase_bn	Reads findings from a file, and enters into a network
CaseFileRevisesProbs_bn	Reads a file of cases to revise probabilities

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
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
GetNodeValue_bn	Returns the real number finding entered for a continuous node
RetractNodeFindings_bn	Retracts all findings for a single node
RetractAllFindings_bn	Retracts all findings (i.e. the current case) from a network

Compiling and Belief Updating

CompileNet_bn	Compiles a network for fast belief updating
IsBeliefUpdated_bn	Returns TRUE iff node's beliefs have been updated to reflect current findings
GetNodeBeliefs_bn	Returns a node's current beliefs as a vector with 1 probability for each state, doing belief updating if necessary
CaseProbability_bn	Returns the joint probability of the findings entered so far
SetNetElimOrder_bn	Sets the node order used to guide compilation
GetNetElimOrder_bn	Retrieves the node order used to guide compilation

Learning a Network

CaseRevisesProbs_bn	Uses the current case to revise each node's probabilities
CaseFileRevisesProbs_bn	Reads a file of cases to revise probabilities
FadeProbs_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	Sets the probabilities without learning from cases (or starts them off)
SetNodeExperience_bn	Manually sets the amount of experience (or starts it off)

Node Lists

NewNodeList_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)
LengthNodeList_bn	Returns the number of nodes in a list
DupNodeList_bn	Duplicates a list of nodes
FreeNodeList_bn	Frees the memory used by a list of nodes
NodeNamed_bn	Returns the node with the given name, from a given network
GetNetNodes_bn	Returns a list of all the nodes in a network

GetNodeParents_bn	Returns a list of the parents of a node
GetNodeChildren_bn	Returns a list of the children of a node
GetNetElimOrder_bn	Returns a list of the elimination order used for compilation
SetNetElimOrder_bn	Provides the elimination order to be used for the next compilation

Cases

(see also "Findings")	To enter a case into a network, and to read it out
WriteCase_bn	Saves a network's current set of findings to a file
ReadCase_bn	Reads findings from a file, and enters into a network
CaseRevisesProbs_bn	The current case is used to revise each node's probabilities
CaseFileRevisesProbs_bn	Reads a file of cases to revise probabilities
CaseProbability_bn	Returns the joint probability of the findings entered so far
RandomCase_bn	Generates a random case in a net, according to the net's distribution
RetractAllFindings_bn	Retracts all findings (i.e. the current case) from a network

High-Level Network Modification

ReverseLink_bn	Reverses a single link while maintaining joint probability
AbsorbNodes_bn	Absorbs out (sum or max) some network nodes
OptimizeDecisions_bn	Finds the optimal decision for each decision node in a net

Low-Level Network Modification

NewNet_bn	Creates a new empty network
FreeNet_bn	Frees all memory used by a network and all its substructures
NewNode_bn	Creates a new node for a given net
DuplicateNodes_bn	Duplicates nodes, and can put the duplicates in a new network
DeleteNode_bn	Removes a node from its network, and frees the memory it required
AddLink_bn	Adds a link from one node to another
DeleteLink_bn	Removes a link from one node to another
SwitchNodeParent_bn	Switches a parent of the node without changing the node
SetNetName_bn	Changes the name of the network
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 network
SetNetComment_bn	Attaches a comment string to the net
SetNetUserData_bn	Sets a net's "user" field to reference the specified data
SetNodeName_bn	Changes the name of a 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
SetNodeLevels_bn	Sets a threshold number for continuous / discrete conversion
SetLinkName_bn	Sets the name of a link to the given string
SetNodeFuncValue_bn	Sets the value of the node when the parents have the given values

SetNodeProbs_bn	Sets the conditional probability of the node given its parents values
SetNodeExperience_bn	Attaches an experience level to a conditional probability vector
DeleteNodeRelation_bn	Removes a node's functon, probabilities, and experience info
SetNodeTitle_bn	Sets the string used to title a node
SetNodeComment_bn	Attaches a comment string to the node
SetNodeUserData_bn	Sets a node's "user" field to reference the specified data

Retrieving Network Information

GetNetName_bn	Returns the name of the network
GetNetNodes_bn	Returns a list of all the nodes in a network
NodeNamed_bn	Returns the node having the given name from the network
GetNetAutoUpdate_bn	Returns whether the network does belief updating immediately
GetNetElimOrder_bn	Returns a list of the elimination order used for triangulation
GetNetTitle_bn	Returns the string which is the network's title
GetNetComment_bn	Returns the comment associated with the network
GetNetUserData_bn	Returns a net's "user" field (i.e. the value it was set to)

GetNodeNet_bn	Returns the network containing the given node
GetNodeName_bn	Returns the name of the given node
GetNodeKind_bn	Returns whether the node is a nature, decision, utility, etc. node
GetNodeDiscrete_bn	Returns whether the the node is for a discrete variable
IsNodeDeterministic_bn	Discovers whether the node's value is a deterministic function of its parents
GetNodeNumberStates_bn	Returns the number of states node can take on
GetNodeStateName_bn	Returns the name of the given state
StateNamed_bn	Returns the state number of the state with the given name
GetNodeLevel_bn	Returns a threshold number for continuous / discrete conversion
GetLinkName_bn	Returns the name associated with a link
LinkNamed_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
GetNodeProbs_bn	Returns the conditional probabilities of the node given its parents
GetNodeExperience_bn	Returns how much learning is associated with the node
GetNodeFuncValue_bn	Returns the deterministic value of a continuous node
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
GetNodeValue_bn	Returns the real number finding entered for a continuous node
GetNodeBeliefs_bn	Returns a node's current beliefs as a vector with 1 probability for each state
GetNodeTitle_bn	Returns the string titling the node
GetNodeComment_bn	Returns a comment string for the node
GetNodeUserData_bn	Returns the current value of the node's "user" field

User Data Fields

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)

12 Index