# HUGIN API

## REFERENCE MANUAL

Version 7.3

**HUGIN**EXPERT

# HUGIN API Reference Manual

This manual was prepared using the LaTeX Document Preparation System and the PDFTeX typesetting software.

Set in 11 point Bitstream Charter, Bitstream Courier, and AMS Euler.

Version 7.3, March 2010.

# Preface

The “HUGIN API 7.3 Reference Manual” provides a reference for the C language Application Program Interface to the HUGIN system. However, brief descriptions of the Java and C++ versions are also provided (see Chapter 1).

The present manual assumes familiarity with the methodology of Bayesian belief networks and (limited memory) influence diagrams (LIMIDs) as well as knowledge of the C programming language and programming concepts.

As introductions to Bayesian belief networks and influence diagrams, the books by Jensen and Nielsen [12] and Kjærulff and Madsen [15] are recommended. Deeper treatments of the subjects are given in the books by Cowell *et al* [7] and Pearl [26].

## Overview of the manual

Chapter 1 explains how to use the HUGIN API within your own applications. It also gives some general information on the functions and data types defined by the HUGIN API and explains the mechanisms for error handling. Finally, instructions on how to take advantage of multi-processor systems to speed up inference are given.

Chapter 2 describes the functions for creating and modifying belief networks and LIMIDs, collectively known as domains. It also explains how to save and load domains to/from knowledge base files.

Chapter 3 provides the tools for constructing object-oriented belief network and LIMID models. Moreover, a function for converting an object-oriented model to an equivalent domain is given (which is needed because inference cannot be performed directly in an object-oriented model).

Tables are used to represent conditional probability distributions, policies, utility functions, sets of experience counts, and sets of fading factors associated with the nodes of the network, as well as joint probability distributions and so-called “mixture marginals” (representing marginal distributions of continuous nodes). Chapter 4 explains how to access and modify the contents of tables.

Chapter 5 describes how the contents of a conditional probability, a policy, or a utility table can be generated from a mathematical description of the relationship between a node and its parents.

Chapter 6 explains how to transform a domain into a secondary structure (a junction forest), suitable for inference. This transformation is known as compilation. It also explains how to improve performance of inference by controlling the triangulation step and by performing approximation and compression.

Chapter 7 explains how to access the collection of junction trees of a compiled domain and how to traverse a junction tree.

Chapter 8 shows how to handle the beliefs and the evidence that form the core of the reasoning process in the HUGIN inference engine. This chapter explains how to enter and retract evidence, how to determine independence properties induced by evidence and network structure, how to retrieve beliefs and expected utilities, how to compute values of function nodes, how to examine evidence, and how to save evidence as a case file for later use.

Chapter 9 documents the functions used to control the inference engine itself. The chapter also explains how to perform *conflict analysis*, *simulation*, *value of information analysis*, *sensitivity analysis*, and how to find the *most probable configurations* of a set of nodes.

Chapter 10 explains how conditional probability distributions can be adapted as new evidence is observed, and Chapter 11 describes how both the network structure and the conditional probability distributions can be extracted ("learned") from data (a set of cases).

Chapter 12 describes the NET language, a language used to specify the nodes and the structure of a network as well as the numerical data required to form a complete specification.

Chapter 13 describes how to enter and modify information that is purely descriptive. This information is not used by other parts of the HUGIN API. It is used by the HUGIN GUI application to generate a graphical display of a network.

Appendix A gives an example of a network using CG variables. Appendix B provides a history of news and changes for all releases of the HUGIN API since version 2.

Finally, an index is provided. The index contains the names of all functions, types, and constants of enumeration types, defined in this manual.

## A note on the API function descriptions

The description of functions in this manual are given in terms of ISO/ANSI C function prototypes, giving the names and types of the functions and their arguments.

The notation "*h_domain_compile*$^{(89)}$" is used to refer to an API function (in this case, the *h_domain_compile* function). The parenthesized, superscripted number (89) refers to the page where the function is described.

## A note on the examples

Throughout the manual, brief examples are provided to illustrate the use of particular functions. These examples will not be complete applications. Rather, they will be small pieces of code that show how a function (or a group of functions) might be used.

While each example is intended to illustrate the use of a particular function, other functions of the HUGIN API will be used to make the examples more realistic. As this manual is not intended as a tutorial but as a reference, many examples will use functions described later in the manual. Therefore, if you read the manual sequentially, you cannot expect to be able to understand all examples the first time through.

For the sake of brevity, most examples do not include error checking. It should be pointed out that using this practice in real applications is strongly discouraged.

## Acknowledgements

Lars P. Fischer wrote the HUGIN API 1.1 manual, and Per Abrahamsen wrote the HUGIN API 1.2 (Extensions) manual. The present document is partly based on these manuals.

I would also like to thank Marianne Bangsø, Søren L. Dittmer, Uffe Kjærulff, Michael Lang, Anders L. Madsen, Lars Nielsen, Lars Bo Nielsen, and Kristian G. Olesen for providing constructive comments and other contributions that have improved this document as well as the API itself. In particular, Anders L. Madsen wrote a large part of Chapter 10.

Any errors and omissions remaining in this manual are, however, my responsibility.

The development of the functionality concerning the function node type (introduced in HUGIN 7.3) has been sponsored by Danish mortgage credit institution Nykredit Realkredit (`www.nykredit.dk`).

—Frank Jensen
Hugin Expert A/S
March, 2010

# Contents

# Chapter 1

# General Information

This chapter explains how to use the HUGIN API within your own applications. It also gives some general information on the functions and data types defined by the HUGIN API and explains the mechanisms for error handling. Finally, instructions on how to take advantage of multi-processor systems to speed up inference is given.

## 1.1 Introduction

The HUGIN API contains a high performance inference engine that can be used as the core of knowledge based systems built using Bayesian belief networks or "limited memory" influence diagrams (LIMIDs) [20]. A knowledge engineer can build knowledge bases that model the application domain, using probabilistic descriptions of causal relationships in the domain. Given this description, the HUGIN inference engine can perform fast and accurate reasoning.

The HUGIN API is provided in the form of a library that can be linked into applications written using the C, C++, or Java programming languages. The C version provides a traditional function-oriented interface, while the C++ and Java versions provide an object-oriented interface. The present manual describes the C interface. The C++ and Java interfaces are described in online documentation supplied with the respective libraries.

On Windows platforms only, C# and Visual Basic language interfaces are also available.

The HUGIN API is used just like any other library. It does not require any special programming techniques or program structures. The HUGIN API does not control your application. Rather, your application controls the HUGIN API by telling it which operations to perform. The HUGIN inference engine sits passive until you engage it.

Applications built using the HUGIN API can make use of any other library packages such as database servers, GUI toolkits, etc. The HUGIN API itself only depends on (in addition to the Standard C library) the presence of the Zlib library (`www.zlib.net`), which is preinstalled in the Solaris, Linux, and Mac OS X operating environments.

## 1.2 Using the HUGIN API on UNIX platforms

The first step in using the C version of the HUGIN API is to include the definitions for the HUGIN functions and data types in the program. This is done by inserting the following line at the top of the program source code:

```
# include "hugin.h"
```

The `hugin.h` header file contains all the definitions for the API.

When compiling the program, you must inform the C compiler where the header file is stored. Assuming the HUGIN system has been installed in the directory `/usr/local/hugin`, the following command is used:

```
cc -I/usr/local/hugin/include -c myapp.c
```

This will compile the source code file `myapp.c` and store the result in the object code file `myapp.o`, without linking. The `-I` option adds the directory `/usr/local/hugin/include` to the search path for include files.

If you have installed the HUGIN system somewhere else, the path above must be modified as appropriate. If the environment variable `HUGINHOME` has been defined to point to the location of the HUGIN installation, the following command can be used:

```
cc -I$HUGINHOME/include -c myapp.c
```

Using the environment variable, `HUGINHOME`, has the advantage that if the HUGIN system is moved, only the environment variable must be changed.

When the source code, possibly stored in several files, has been compiled, the object files must be linked to create an executable file. At this point, it is necessary to specify that the object files should be linked with the HUGIN library:

```
cc myapp.o other.o -L$HUGINHOME/lib -lhugin -lm -lz
```

The `-L$HUGINHOME/lib` option specifies the directory to search for the HUGIN libraries, while the `-lhugin` option specifies the library to link with. The `-lz` option directs the compiler/linker to link with the Zlib library (`www.zlib.net`). This option is needed if either of the *h_domain_save_as_kb*(45) or *h_kb_load_domain*(46) functions is used.

If the source code for your application is a single file, you can simplify the above to:

```
cc -I$HUGINHOME/include myapp.c
   -L$HUGINHOME/lib -lhugin -lm -lz -o myapp
```

compiling the source code file `myapp.c` and storing the final application in the executable file `myapp`. (Note that the above command should be typed as a single line.)

Following the above instructions will result in an executable using the single-precision version of the HUGIN API library. If, instead, you want to use the double-precision version of the HUGIN API library, you must define `H_DOUBLE` when you invoke the compiler, and specify `-lhugin2` for the linking step:

```
cc -DH_DOUBLE -I$HUGINHOME/include myapp.c
   -L$HUGINHOME/lib -lhugin2 -lm -lz -o myapp
```

(Again, all this should be typed on one line.)

The above might look daring, but it would typically be done in a `Makefile` so that you will only have to do it once for each project.

The instructions above also assume that you are using the 32-bit version of the HUGIN API. If, instead, you are using the 64-bit version, you will need to specify `$HUGINHOME/lib64` as the directory to search for libraries.

The `hugin.h` header file has been designed to work with both ISO C compliant compilers and C++ compilers. For C++ compilers, the `hugin.h` header file depends on the symbol `__cplusplus` being defined (this symbol should be automatically defined by the compiler).

Some API functions take pointers to `stdio` **FILE** objects as arguments. This implies that inclusion of `hugin.h` also implies inclusion of `<stdio.h>`. Moreover, in order to provide suitable type definitions, the standard C header `<stddef.h>` is also included.

## Object-oriented versions of the HUGIN API: Java and C++

The standard HUGIN API as defined by the `hugin.h` header file and described in the present manual has a function-oriented interface style. Object-oriented versions, more appropriate for use in object-oriented language environments, have been developed for the Java and C++ languages. These versions have almost identical interfaces, and it should be very easy for developers to switch between them (if this should ever be necessary).

The Java and C++ versions use classes for modeling domains, nodes, etc. Each of the classes has a set of methods enabling you to manipulate objects of the class. These methods will throw exceptions when errors occur. The exception classes are all subclasses of the main HUGIN exception class (*ExceptionHugin*). In Java, this is an extension of the standard Java *Exception* class.

The classes, methods, and exceptions are all specified in the online documentation distributed together with these interfaces.

**C++** To use the C++ HUGIN API definitions in your code, you must include the `hugin` header file (note that there is no suffix):

```
# include "hugin"
```

All entities defined by the C++ API are defined within the `HAPI` namespace. To access these entities, either use the `HAPI::` prefix or place the following declaration before the first use of C++ API entities (but after the `hugin` header file has been included):

```
using namespace HAPI;
```

Like the C API, the C++ API is available in two versions: a single-precision version and a double-precision version. To use the single-precision version, use a command like the following for compiling and linking:

```
g++ -I$HUGINHOME/include myapp.c
    -L$HUGINHOME/lib -lhugincpp -lm -lz -o myapp
```

(This should be typed on one line.) To use the double-precision version, define the `H_DOUBLE` preprocessor symbol and specify `-lhugincpp2` for the linking step:

```
g++ -DH_DOUBLE -I$HUGINHOME/include myapp.c
    -L$HUGINHOME/lib -lhugincpp2 -lm -lz -o myapp
```

(Again, this should be typed on one line.)

Also, the C++ API is available as both a 32-bit and a 64-bit version. To use the 64-bit version, specify `$HUGINHOME/lib64` as the directory to search for libraries.

**Java** The Java version of the HUGIN API library is provided as two files (the files comprising the 64-bit version have an additional `-64` suffix):

- `hapi73.jar` (`hapi73-64.jar`) contains the Java interface to the underlying C library. This file must be mentioned by the `CLASSPATH` environment variable.

- `libhapi73.so` (`libhapi73-64.so`) contains the native version of the HUGIN API for the platform used. When running the Java VM, this file must be located in a directory mentioned by the `LD_LIBRARY_PATH` environment variable.

The Java version of the HUGIN API is a double-precision library.

Java SE, version 1.4.2 or newer, is required: `http://java.sun.com/javase/downloads/`.

## 1.3 Using the HUGIN API on Windows platforms

C, C++, Java, C# (.NET), and Visual Basic language interfaces for the HUGIN API are available on the Windows platforms.

Four[1] sets of library files are provided with developer versions of HUGIN for Windows: One for Microsoft Visual Studio 6.0, one for Microsoft Visual Studio .NET 2003, one for Microsoft Visual Studio 2005, and one for Microsoft Visual Studio 2008.[2] Each set of library files contains libraries for all combinations of the following:

- C and C++ programming languages;
- `Debug` and `Release` configurations;
- single-precision and double-precision.

Each of these libraries has two parts: An import library and a DLL. For example, the import library for the 32-bit, double-precision, C++ version of the HUGIN API compiled for Microsoft Visual Studio 2008, `Debug` configuration, is named `hugincpp2-7.3-vc9d.lib`, and the corresponding DLL file is named `hugincpp2-7.3-vc9d.dll`. For 64-bit versions, there is an additional suffix: `hugincpp2-7.3-vc9d-x64` plus the usual `.lib` and `.dll` extensions.

In general, the library files have unique names, indicating language (C or C++), API version number, compiler, configuration, and 32-bit/64-bit. This naming scheme makes it possible for all DLLs to be in the search path simultaneously.

### C version of the HUGIN API

The C version of the HUGIN API is located in the `HDE7.3C` subdirectory of the main HUGIN installation directory.

To use the C version of the HUGIN API in your code, you must include the `hugin.h` header file:

```
# include "hugin.h"
```

#### Microsoft Visual Studio 6.0

Let ⟨Path⟩ denote the main directory of the Hugin installation (for example: `C:\Program Files\Hugin Expert\Hugin Developer 7.3`), and let ⟨Configuration⟩ denote the active configuration (either `Debug` or `Release`).

---

[1] For 64-bit packages, only two sets of library files are provided (for Microsoft Visual Studio 2005 and Microsoft Visual Studio 2008).

[2] If you need libraries for other development environments, please contact `info@hugin.com`.

(1) Click “Settings” on the “Project” menu. Click the “C/C++” tab, and select “Preprocessor” in the “Category” box.

  (a) Add ⟨Path⟩\`HDE7.3C`\`Include` to the “Additional include directories” box.

(2) Click “Settings” on the “Project” menu. Click the “Link” tab, and select “Input” in the “Category” box.

  (a) Add ⟨Path⟩\`HDE7.3C`\`Lib`\`VC6`\⟨Configuration⟩ to the “Additional library path” box.

  (b) Add the import library to the “Object/library modules” list:

    - If ⟨Configuration⟩ is `Debug`, add `hugin-7.3-vc6d.lib`.
    - If ⟨Configuration⟩ is `Release`, add `hugin-7.3-vc6.lib`.

(3) Click “Settings” on the “Project” menu. Click the “C/C++” tab, and select “Code Generation” in the “Category” box.

  - If ⟨Configuration⟩ is `Debug`, make sure “Debug Multithreaded DLL” is selected in the “Use run-time library” box.
  - If ⟨Configuration⟩ is `Release`, make sure “Multithreaded DLL” is selected in the “Use run-time library” box.

The above steps set up Microsoft Visual Studio 6.0 to use the single-precision version of the HUGIN API. If you want to use the double-precision version, modify the instructions as follows:

(1) Click “Settings” on the “Project” menu. Click the “C/C++” tab, and select “Preprocessor” in the “Category” box.

  (b) Add `H_DOUBLE` to the “Preprocessor definitions” box.

(2) Click “Settings” on the “Project” menu. Click the “Link” tab, and select “Input” in the “Category” box.

  (b) Add the import library to the “Object/library modules” list:

    - If ⟨Configuration⟩ is `Debug`, add `hugin2-7.3-vc6d.lib`.
    - If ⟨Configuration⟩ is `Release`, add `hugin2-7.3-vc6.lib`.

When running the compiled program, the DLL corresponding to the import library used in the compilation must be located in a directory mentioned in the search path.

### Microsoft Visual Studio .NET 2003

Let ⟨Path⟩ denote the main directory of the Hugin installation (for example: `C:\Program Files\Hugin Expert\Hugin Developer 7.3`), and let ⟨Configuration⟩ denote the active configuration (either `Debug` or `Release`).

(1a) Click "Properties" on the "Project" menu. Click the "C/C++" folder, and select the "General" property page.

- Add ⟨Path⟩`\HDE7.3C\Include` to the "Additional Include Directories" property.

(2a) Click "Properties" on the "Project" menu. Click the "Linker" folder, and select the "General" property page.

- Add ⟨Path⟩`\HDE7.3C\Lib\VC7\`⟨Configuration⟩ to the "Additional Library Directories" property.

(2b) Click "Properties" on the "Project" menu. Click the "Linker" folder, and select the "Input" property page.

- Add the import library to the "Additional Dependencies" property:
  - If ⟨Configuration⟩ is `Debug`, add `hugin-7.3-vc7d.lib`.
  - If ⟨Configuration⟩ is `Release`, add `hugin-7.3-vc7.lib`.

(3) Click "Properties" on the "Project" menu. Click the "C/C++" folder, and select the "Code Generation" property page.

- If ⟨Configuration⟩ is `Debug`, make sure "Multi-threaded Debug DLL (/MDd)" is selected in the "Runtime Library" property.
- If ⟨Configuration⟩ is `Release`, make sure "Multi-threaded DLL (/MD)" is selected in the "Runtime Library" property.

The above steps set up Microsoft Visual Studio .NET 2003 to use the single-precision version of the HUGIN API. If you want to use the double-precision version, modify the instructions as follows:

(1b) Click "Properties" on the "Project" menu. Click the "C/C++" folder, and select the "Preprocessor" property page.

- Add `H_DOUBLE` to the "Preprocessor Definitions" property.

(2b) Click "Properties" on the "Project" menu. Click the "Linker" folder, and select the "Input" property page.

- Add the import library to the "Additional Dependencies" property:

  - If ⟨Configuration⟩ is `Debug`, add `hugin2-7.3-vc7d.lib`.
  - If ⟨Configuration⟩ is `Release`, add `hugin2-7.3-vc7.lib`.

When running the compiled program, the DLL corresponding to the import library used in the compilation must be located in a directory mentioned in the search path.

#### Microsoft Visual Studio 2005 and Microsoft Visual Studio 2008

The instructions for using the HUGIN C API in Visual Studio 2005 and Visual Studio 2008 are the same as those given above for Visual Studio .NET 2003, except that the compilers are named `vc8` and `vc9` instead of `vc7`.

64-bit versions of the HUGIN C API are available for Visual Studio 2005 and Visual Studio 2008. The library files for these versions have an additional suffix (`-x64`). This suffix must be included when the import library is specified.

### C++ object-oriented version of the HUGIN API

The C++ version of the HUGIN API is located in the `HDE7.3CPP` subdirectory of the main HUGIN installation directory. The documentation for all classes and their members is located in the `Doc` subdirectory below the `HDE7.3CPP` directory.

To use the C++ HUGIN API definitions in your code, you must include the `hugin` header file (note that there is no suffix):

```
# include "hugin"
```

All entities defined by the C++ API are defined within the `HAPI` namespace. To access these entities, either use the `HAPI::` prefix or place the following declaration before the first use of C++ API entities (but after the `hugin` header file has been included):

```
using namespace HAPI;
```

#### Microsoft Visual Studio 6.0

Let ⟨Path⟩ denote the main directory of the Hugin installation (for example: `C:\Program Files\Hugin Expert\Hugin Developer 7.3`), and let ⟨Configuration⟩ denote the active configuration (either `Debug` or `Release`).

(1) Click “Settings” on the “Project” menu. Click the “C/C++” tab, and select “Preprocessor” in the “Category” box.

    (a) Add ⟨Path⟩\\`HDE7.3CPP`\\`Include` to the "Additional include directories" box.

(2) Click "Settings" on the "Project" menu. Click the "Link" tab, and select "Input" in the "Category" box.

    (a) Add ⟨Path⟩\\`HDE7.3CPP`\\`Lib`\\`VC6`\\⟨Configuration⟩ to the "Additional library path" box.
    (b) Add the import library to the "Object/library modules" list:
        - If ⟨Configuration⟩ is `Debug`, add `hugincpp-7.3-vc6d.lib`.
        - If ⟨Configuration⟩ is `Release`, add `hugincpp-7.3-vc6.lib`.

(3) Click "Settings" on the "Project" menu. Click the "C/C++" tab, and select "Code Generation" in the "Category" box.

    - If ⟨Configuration⟩ is `Debug`, make sure "Debug Multithreaded DLL" is selected in the "Use run-time library" box.
    - If ⟨Configuration⟩ is `Release`, make sure "Multithreaded DLL" is selected in the "Use run-time library" box.

(4) Click "Settings" on the "Project" menu. Click the "C/C++" tab, and select "C++ Language" in the "Category" box.

    - Make sure "Enable exception handling" is selected.
    - Make sure "Enable Run-Time Type Information (RTTI)" is selected.

The above steps set up Microsoft Visual Studio 6.0 to use the single-precision version of the HUGIN API. If you want to use the double-precision version, modify the instructions as follows:

(1) Click "Settings" on the "Project" menu. Click the "C/C++" tab, and select "Preprocessor" in the "Category" box.

    (b) Add `H_DOUBLE` to the "Preprocessor definitions" box.

(2) Click "Settings" on the "Project" menu. Click the "Link" tab, and select "Input" in the "Category" box.

    (b) Add the import library to the "Object/library modules" list:
        - If ⟨Configuration⟩ is `Debug`, add `hugincpp2-7.3-vc6d.lib`.
        - If ⟨Configuration⟩ is `Release`, add `hugincpp2-7.3-vc6.lib`.

When running the compiled program, the DLL corresponding to the import library used in the compilation must be located in a directory mentioned in the search path.

**Microsoft Visual Studio .NET 2003**

Let ⟨Path⟩ denote the main directory of the Hugin installation (for example: `C:\Program Files\Hugin Expert\Hugin Developer 7.3`), and let ⟨Configuration⟩ denote the active configuration (either `Debug` or `Release`).

(1a) Click "Properties" on the "Project" menu. Click the "C/C++" folder, and select the "General" property page.

- Add ⟨Path⟩`\HDE7.3CPP\Include` to the "Additional Include Directories" property.

(2a) Click "Properties" on the "Project" menu. Click the "Linker" folder, and select the "General" property page.

- Add ⟨Path⟩`\HDE7.3CPP\Lib\VC7\`⟨Configuration⟩ to the "Additional Library Directories" property.

(2b) Click "Properties" on the "Project" menu. Click the "Linker" folder, and select the "Input" property page.

- Add the import library to the "Additional Dependencies" property:
  - If ⟨Configuration⟩ is `Debug`, add `hugincpp-7.3-vc7d.lib`.
  - If ⟨Configuration⟩ is `Release`, add `hugincpp-7.3-vc7.lib`.

(3) Click "Properties" on the "Project" menu. Click the "C/C++" folder, and select the "Code Generation" property page.

- If ⟨Configuration⟩ is `Debug`, make sure "Multi-threaded Debug DLL (/MDd)" is selected in the "Runtime Library" property.
- If ⟨Configuration⟩ is `Release`, make sure "Multi-threaded DLL (/MD)" is selected in the "Runtime Library" property.

(4a) Click "Properties" on the "Project" menu. Click the "C/C++" folder, and select the "Code Generation" property page

- Make sure "Enable C++ Exceptions" is selected.

(4b) Click "Properties" on the "Project" menu. Click the "C/C++" folder, and select the "Language" property page

- Make sure "Enable Run-Time Type Info" is selected.

The above steps set up Microsoft Visual Studio .NET 2003 to use the single-precision version of the HUGIN API. If you want to use the double-precision version, modify the instructions as follows:

(1b) Click “Properties” on the “Project” menu. Click the “C/C++” folder, and select the “Preprocessor” property page.

  - Add `H_DOUBLE` to the “Preprocessor Definitions” property.

(2b) Click “Properties” on the “Project” menu. Click the “Linker” folder, and select the “Input” property page.

  - Add the import library to the “Additional Dependencies” property:
    - If ⟨Configuration⟩ is `Debug`, add `hugincpp2-7.3-vc7d.lib`.
    - If ⟨Configuration⟩ is `Release`, add `hugincpp2-7.3-vc7.lib`.

When running the compiled program, the DLL corresponding to the import library used in the compilation must be located in a directory mentioned in the search path.

#### Microsoft Visual Studio 2005 and Microsoft Visual Studio 2008

The instructions for using the HUGIN C++ API in Visual Studio 2005 and Visual Studio 2008 are the same as those given above for Visual Studio .NET 2003, except that the compilers are named `vc8` and `vc9` instead of `vc7`.

64-bit versions of the HUGIN C++ API are available for Visual Studio 2005 and Visual Studio 2008. The library files for these versions have an additional suffix (`-x64`). This suffix must be included when the import library is specified.

### .NET version of the HUGIN API

The .NET version of the HUGIN API is located in the `HDE7.3CS` subdirectory of the main HUGIN installation directory (called ⟨Path⟩ below).

The documentation for all classes and their members is located in the ⟨Path⟩\ `HDE7.3CS\Doc` directory. The documentation is written for C#, but the API can also be used with other .NET-based languages.

All entities defined by the HUGIN .NET API are defined within the `HAPI` namespace. To access these entities, either use the `HAPI.` prefix, or place the following declaration before any namespace and class declarations:

```
using HAPI;
```

The HUGIN .NET API is provided in the form of a DLL targeting version 2.0 of the .NET framework. There are four versions of the HUGIN .NET API, corresponding to all combinations of:

- single-precision and double-precision;

- 32-bit and 64-bit Windows platforms.

The 32-bit DLLs are named `hugincs-7.3-2.0.dll` (single-precision) and `hugincs2-7.3-2.0.dll` (double-precision), where `7.3` denotes the version number of the HUGIN API, and `2.0` denotes the version of the .NET framework targeted by the DLL. The 64-bit DLLs have an additional suffix (`-x64`).

Note that certain types vary between the DLLs: The **h_number_t** type is either a single-precision or a double-precision floating-point type, and **size_t** and **h_index_t** are either 32-bit or 64-bit integer types. In order to make switching between the different HUGIN .NET API versions easier, it is recommended to declare aliases for **h_number_t**, **size_t** and **h_index_t** in the appropriate source files and use these aliases as types. This can be done by placing the following piece of code at the beginning of the source files:

```
#if X64
using size_t = System.UInt64;
using h_index_t = System.Int64;
#else
using size_t = System.UInt32;
using h_index_t = System.Int32;
#endif
#if H_DOUBLE
using h_number_t = System.Double;
#else
using h_number_t = System.Single;
#endif
```

The symbol `H_DOUBLE` must be defined (only) when using a double-precision version of the HUGIN .NET API, and the symbol `X64` must be defined (only) when using a 64-bit version of the HUGIN .NET API. (See more examples in the documentation accompanying the HUGIN .NET API.)

The Microsoft .NET Framework version 2.0 is required by the HUGIN .NET API: `http://msdn.microsoft.com/netframework/`

### Microsoft Visual Studio 2005 and Microsoft Visual Studio 2008

The following steps set up a Microsoft Visual Studio 2005 / Microsoft Visual Studio 2008 C# Project to use the .NET version of the HUGIN API:

(1) Click "Add Reference" on the "Project" menu. Click the "Browse" tab, browse to the location of the HUGIN .NET API DLL files and select a file corresponding to the desired version.

(2) Click “Properties” on the “Project” menu. Click the “Build” fan, and configure “Platform target” to either x86 or x64 (according to the HUGIN .NET API version used).

(3) If using the type aliasing scheme described above, define the appropriate symbols in the field “Conditional compilation symbols” under the “General” category on the “Build” fan.

When running the compiled program, the DLL referenced in the project must be located in a directory mentioned in the search path.

### .NET Compact Framework version of the HUGIN API

The HUGIN API PDA support consists of a single-precision version of the HUGIN .NET API for the .NET Compact Framework. The API targets devices with a configuration of:

- Windows Mobile 6 Professional
- .NET Compact Framework 2.0 (or higher)

Visual Studio 2008 and Windows Mobile 6 Professional SDK Refresh must be installed on the development computer. The SDK can be downloaded at `http://www.microsoft.com/downloads/details.aspx?FamilyID=06111A3A-A651-4745-88EF-3D48091A390B`

The HUGIN API consists of two files: A .NET Compact Framework assembly (C#) and a DLL containing a native ARM implementation of the HUGIN decision engine.

The API files are located in the ⟨Path⟩\\`HDE7.2CS`\\`CompactFramework`\\`Lib` directory:

- `hugincs-7.2-2.0.dll` — .NET Compact Framework assembly (add this file as a reference in the VS2008 compact framework solution);
- `pda-7.2-vc9.dll` — HUGIN decision engine.

Both files are needed for the HUGIN API to function. The .NET assembly is a managed wrapper for the native ARM implementation of the HUGIN decision engine. Both files must be copied to the embedded device along with the application executable in order for the API to function.

Developer and researcher APIs require that license details be stored on the device. License details are configured on the device by copying the executable found in ⟨Path⟩\\`HDE7.2CS`\\`CompactFramework`\\`HuginRegister` to the device and running it.

### Java version of the HUGIN API

The Java version of the HUGIN API is located in the `HDE7.3J` subdirectory of the main HUGIN installation directory (called ⟨Path⟩ below).

The documentation for all the classes and their members is located in the ⟨Path⟩\\`HDE7.3J`\\`Doc` directory. An entry to this documentation is installed in the Start-up menu.

When running a Hugin-based Java application, the Java VM must have access to the following files (the files comprising the 64-bit version have an additional `-64` suffix):

- `hapi73.jar` (`hapi73-64.jar`): This file is located in the ⟨Path⟩\\`HDE7.3J`\\`Lib` directory. Add this JAR file to the classpath when running the Java VM: For the Sun Java SE VM, set the `CLASSPATH` environment variable to include ⟨Path⟩\\`HDE7.3J`\\`Lib`\\`hapi73.jar`, or specify it using the `-cp` (or the `-classpath`) option of the `java.exe` command.

- `hapi73.dll` (`hapi73-64.dll`): This file is located in the ⟨Path⟩\\`HDE7.3J`\\`Bin` directory. When running the Java VM, this file must be in the search path (or specified using the `-Djava.library.path` option).

The Java version of the HUGIN API is a double-precision library.

Java SE, version 1.4.2 or newer, is required: `http://java.sun.com/javase/downloads/`.

### Visual Basic version of the HUGIN API

The Visual Basic version of the HUGIN API is located in the `HDE7.3X` subdirectory of the main HUGIN installation directory (called ⟨Path⟩ below).

The documentation for all the classes and their members is located in the ⟨Path⟩\\`HDE7.3X`\\`Doc` directory. An entry to this documentation is installed in the Start-up menu.

To use the Visual Basic HUGIN API in your code, perform the following step:

- Click “References” on the “Project” menu, and select the “Hugin API ActiveX Server” in the list of available modules.

When running the program, the following files must be accessible:

- `hapi73.dll`: This is located in the ⟨Path⟩\\`HDE7.3J`\\`Lib` directory. It is automatically registered when Hugin is installed.

- `nphapi73.dll`: This is installed in the `System32` (or `System`, depending on your OS version) directory.

The Visual Basic HUGIN API is a single-precision version of the HUGIN API, and it is only available as a 32-bit version.

## 1.4 Naming conventions

### Naming conventions for the C version

The HUGIN C API reserves identifiers beginning with *h_*. Your application should not use any such names as they might interfere with the HUGIN API. (The HUGIN API also uses names beginning with *_h_* internally; you shouldn't use any such names either.)

The HUGIN API uses various types for representing domains, nodes, tables, cliques, junction trees, error codes, triangulation methods, etc.

All types defined by the HUGIN API have the suffix **_t**.

The set of types, defined by the HUGIN API, can be partitioned into two groups: scalar types and opaque references.

### Naming conventions for the Java and C++ versions

The Java and C++ HUGIN API classes have been constructed based on the different HUGIN API opaque pointer types (see Section 1.5). For example, the **h_domain_t** type in C corresponds to the Domain class in Java/C++. The convention is that you uppercase all letters following an underscore character (_), remove the *h_* prefix and **_t** (or *_T* after uppercasing) suffix, and remove all remaining underscore characters. So, for example, the following classes are defined in the Java and C++ APIs: Clique, Expression, JunctionTree, Model, Node, and Table.

There are some differences between C and object-oriented languages such as Java and C++ that made it natural to add some extra classes. These include different Node subclasses (DiscreteChanceNode, DiscreteDecisionNode, BooleanDCNode, LabelledDDNode, etc.) and a lot of Expression subclasses (AddExpression, ConstantExpression, BinomialDistribution, BetaDistribution, etc.). Each group forms their own class hierarchy below the corresponding superclass. Some of the most specialized Node classes use abbreviations in their names (to avoid too long class names): e.g., BooleanDCNode is a subclass of DiscreteChanceNode which again is a subclass of Node. Here, BooleanDCNode is abbreviated from BooleanDiscreteChanceNode.

The methods defined on the Java/C++ HUGIN API classes all correspond to similar C API functions. For example, the *setName* method of the Node class corresponds to *h_node_set_name*(39). The rule is: the *h_* prefix is removed, letters immediately following all (other) underscore characters are uppercased, and, finally, the underscore characters themselves are removed.

There are some exceptions where functions correspond to class constructors: e.g., the *h_domain_new_node*(29) function in the C version corresponds to a number of different Node subclass constructors in the Java/C++ versions.

## 1.5 Types

### Opaque pointer types

All (structured) objects within the HUGIN API are represented as *opaque pointers*. An opaque pointer is a well-defined, typed, pointer that points to some data that is not further defined. Using opaque pointers makes it possible to manipulate references to data, without knowing the structure of the data itself.

This means that the HUGIN API provides pointers to these types but does not define the structure of the data pointed at. The real data are stored in structures, but the definitions of these structures are hidden. The reason for this is that manipulation of these structures requires knowledge of the workings of the inference engine, and that hiding the structure makes applications independent of the actual details, preventing that future changes to the internals of the HUGIN API require changes in user programs.

Values of opaque pointer types should only be used in the following ways:

- As sources and destinations of assignments.
- As arguments to and return values from functions (both HUGIN API and user-defined functions).
- In comparisons with NULL or 0 or another opaque pointer value of the same type.

You should never try to dereference these pointers. Objects, referenced by an opaque pointer, should only be manipulated using the API functions. This ensures that the internal data structures are always kept consistent.

### Scalar types

Probabilistic reasoning is about numbers, so the HUGIN API will of course need to handle numbers. The beliefs and utilities used in the inference engine are of type **h_number_t**, which is defined as a single-precision floating-point value in the standard version of the HUGIN library. The HUGIN API also defines another floating-point type, **h_double_t**, which is defined as a double-precision floating-point type in the standard version of the HUGIN API. This type is used to represent quantities that are particularly sensitive to range (e.g., the joint probability of evidence — see *h_domain_get_normalization_constant*(124)) and precision (e.g., the summation operations performed as part of a marginalization operation is done with double precision).

The reason for introducing the **h_number_t** and **h_double_t** types is to make it easier to use higher precision versions of the HUGIN API with just a simple recompilation of the application program with some extra flags defined.

The HUGIN API uses a number of enumeration types. Some examples: The type **h_triangulation_method_t** defines the possible triangulation methods used during compilation; the type **h_error_t** defines the various error codes returned when errors occur during execution of API functions. Both of these types will have new values added as extra features are added to the HUGIN API in the future.

Many functions return integer values. However, these integer values have different meanings for different functions.

Functions with no natural return value simply return a status result that indicates if the function failed or succeeded. If the value is zero, the function succeeded; if the value is nonzero, the function failed and the value will be the error code of the error. Such functions can be easily recognized by having the return type **h_status_t**.

Some functions have the return type **h_boolean_t**. Such functions have truth values (i.e., 'true' and 'false') as their natural return values. These functions will return a positive integer for 'true', zero for 'false', and a negative integer if an error occurs. The nature of the error can be revealed by using the *h_error_code*(18) function and friends.

The HUGIN API also defines a number of other types for general use: The type **h_string_t** is used for character strings (this type is used for node names, file names, labels, etc.). The type **h_count_t** is an integral type to denote "counts" (e.g., the number of states of a node), and **h_index_t** is an integral type to denote indexes into ordered lists (e.g., an identification of a particular state of a node); all (non-error) values of these types are non-negative, and a negative value from a function returning a value of one of these types indicates an error.

## 1.6 Errors

Several types of errors can occur when using a function from the HUGIN API. These errors can be the result of errors in the application program, of running out of memory, of corrupted data files, etc.

As a general principle, the HUGIN API will try to recover from any error as well as possible. The API will then inform the application program of the problem and take no further action. It is then up to the application program to choose an appropriate action.

This way of error handling is chosen to give the application programmer the highest possible degree of freedom in dealing with errors. The HUGIN API will never make a choice of error handling, leaving it up to the application programmer to create as elaborate an error recovery scheme as needed.

When a HUGIN API function fails, the data structures will always be left in a consistent state. Moreover, unless otherwise stated explicitly for a particular function, this state can be assumed identical to the state before the failed API call.

To communicate errors to the user of the HUGIN API, the API defines the enumeration type **h_error_t**. This type contains constants to identify the various types of errors. All constants for values of the **h_error_t** type have the prefix *h_error_*.

All functions in the HUGIN API (except those described in this section) set an error indicator. This error indicator can be inspected using the *h_error_code* function.

▸ **h_error_t** ***h_error_code*** **(void)**

Return the error indicator for the most recent call to an API function (other than *h_error_code*, *h_error_name*, and *h_error_description*). If this call was successful, *h_error_code* will return *h_error_none* (which is equal to zero). If this call failed, *h_error_code* will return a nonzero value indicating the nature of the error. If no relevant API call has been made, *h_error_code* will return *h_error_none* (but see also Section 1.8 for information on the error indicator in multithreaded applications).

All API functions return a value. Instead of explicitly calling *h_error_code* to check for errors, this return value can usually be used to check the status (success or failure) of an API call.

All functions with no natural return value (i.e., the return type is **void**) have been modified to return a value. These functions have return type **h_status_t** (which is an alias for an integral type). A zero result from such a function indicates success while a nonzero result indicates failure. Other functions use an otherwise impossible value to indicate errors. For example, consider the *h_node_get_belief*(108) function which returns the belief for a state of a (chance) variable. This is a nonnegative number (and less than or equal to one since it is a probability). This function returns a negative number if an error occurred. Such a convention is not possible for the *h_node_get_expected_utility*(111) function since any real number is a valid utility; in this case, the *h_error_code* function must be used.

Also, most functions that return a pointer value use NULL to indicate errors. The only exception is the group of functions that handle arbitrary "user data" (see Section 2.9.1) since NULL can be a valid datum.

It is important that the application always checks for errors. Even the most innocent-looking function might generate an error.

Note that, if an API function returns a value indicating that an error occurred, the inference engine may be in a state where normal progress of the application is impossible. This is the case if, say, a domain could not be loaded. For the sanity of the application it is therefore good programming practice to always examine return values and check for errors, just like when using ordinary Standard C library calls.

### 1.6.1 Handling errors

The simplest way to deal with errors in an application is to print an error message and abort execution of the program. To generate an appropriate error message, the following functions can be used.

Each error has a short unique name, which can be used for short error messages.

▸ **h_string_t** ***h_error_name*** (**h_error_t** *code*)

Return the name of the error with code *code*. If *code* is not a valid error code, `"no_such_error"` is returned.

▸ **h_string_t** ***h_error_description*** (**h_error_t** *code*)

Return a long description of the error with code *code*. This description is suitable for display in, e.g., a message window. The string contains no 'newline' characters, so you have to format it yourself.

**Example 1.1** The following code fragment attempts to load a domain from the HUGIN Knowledge Base file named *file_name*. The file is assumed to be protected by *password*.

```
h_domain_t d;
...
if ((d = h_kb_load_domain (file_name, password)) == NULL)
{
    fprintf (stderr, "h_kb_load_domain failed: %s\n",
             h_error_description (h_error_code ()));
    exit (EXIT_FAILURE);
}
```

If the domain could not be loaded, an error message is printed and the program terminates. Lots of things could cause the load operation to fail: the file is non-existing or unreadable, the HUGIN KB file was generated by an incompatible version of the API, the HUGIN KB file was corrupted, insufficient memory, etc. ∎

More sophisticated error handling is also possible by reacting to a specific error code.

**Example 1.2** The propagation functions (see Section 9.2) may detect errors that will often not be considered fatal. Thus, more sophisticated error handling than simple program termination is required.

```
h_domain_t d;
...
if (h_domain_propagate
       (d, h_equilibrium_sum, h_mode_normal) != 0)
    switch (h_error_code ())
    {
    case h_error_inconsistency_or_underflow:
        /*  impossible evidence has been detected,
            retract some evidence and try again   */
        ...
        break;
    ...
    default:
        ...
    }
```

∎

### 1.6.2 General errors

Here is a list of some error codes that most functions might generate.

***h_error_usage*** This error code is returned when a "trivial" violation of the interface for an API function has been detected. Examples of this error: NULL pointers are usually not allowed as arguments (if they are, it will be stated so explicitly); asking for the belief in a non-existing state of a node; etc.

***h_error_no_memory*** The API function failed because there was insufficient (virtual) memory available to perform the operation.

***h_error_io*** Functions that involve I/O (i.e., reading from and writing to files on disk). The errors could be: problems with permissions, files do not exist, disk is full, etc.

## 1.7 Taking advantage of multiple processors

In order to achieve faster inference through parallel execution on multiprocessor systems, many of the most time-consuming table operations have been made threaded. Note, however, that in the current implementation table operations for compressed domains (see Section 6.6) are not threaded.

The creation of threads (or tasks) is controlled by two parameters: the desired *level of concurrency* and the *grain size*. The first of these parameters specifies the maximum number of threads to create when performing a specific table operation, and the second parameter specifies a lower limit on the size of the tasks to be performed by the threads. The size of a task is approximately equal to the number of floaing-point operations needed to perform the task (e.g., the number of elements to sum when performing a marginalization task).

▶ **h_status_t** ***h_domain_set_concurrency_level***
(**h_domain_t** *domain*, **size_t** *level*)

This function sets the level of concurrency associated with *domain* to *level* (this must be a positive number). Setting the concurrency level parameter to 1 will cause all table operations (involving tables originating from *domain*) to be performed sequentially. The initial value of this parameter is 1.

Note that the concurrency level and the grain size parameters are specific to each domain.[3] Hence, the parameters must be explicitly set for all domains for which parallel execution is desired.

▶ **h_count_t** ***h_domain_get_concurrency_level*** (**h_domain_t** *domain*)

This function returns the current concurrency level associated with *domain*.

▶ **h_status_t** ***h_domain_set_grain_size*** (**h_domain_t** *domain*, **size_t** *size*)

This function sets the grain size parameter associated with *domain* to *size* (this value must be positive). The initial value of this parameter is 10 000.

▶ **h_count_t** ***h_domain_get_grain_size*** (**h_domain_t** *domain*)

This function returns the current value of the grain size parameter associated with *domain*.

A table operation involving discrete nodes can naturally be divided into a number ($n$) of suboperations corresponding to the state values of one or more of the discrete nodes. These suboperations are distributed among a number ($m$) of threads such that each thread performs either $\lfloor n/m \rfloor$ or $\lceil n/m \rceil$ suboperations. The number $m$ of threads is chosen to be the highest number satisfying $m \leq l$ and $m \leq n/\lceil g/s \rceil$, where $l$ is the concurrency level, $s$ is the suboperation size, and $g$ is the grain size. If no number $m \geq 2$ satisfies these conditions, the table operation is performed sequentially.

### 1.7.1 Multiprocessing in the Solaris Operating Environment

In order to take advantage of multi-processor systems running the Solaris Operating Environment, you must link your application with a threads library:

---

[3] Chapter 2 explains the *domain* concept as used in the HUGIN API.

```
cc myapp.o otherfile.o
   -L$HUGINHOME/lib -lhugin -lm -lz -lpthread
```

(This should be typed on one line.)

If you omit the `-lpthread` linker option, you get a single-threaded executable.

Note that the Solaris Operating Environment provides two threads libraries: `libpthread` for POSIX threads and `libthread` for Solaris threads. The (Solaris version of the) HUGIN API can be used with both of these libraries.[4]

Due to the nature of the Solaris scheduling system and the fact that the threads created by the HUGIN API are compute-bound, it is necessary (that is, in order to take advantage of multiple processors) to declare how many threads should be running at the same time.

This is done using the POSIX threads function *pthread_setconcurrency* (or the Solaris threads function *thr_setconcurrency*).

**Example 1.3** Assuming we have a system (running the Solaris Operating Environment) with four processors (and we want to use them all for running our application), we tell the HUGIN API to create up to four threads at a time, and we tell the Solaris scheduler to run four threads simultaneously.

```
# include "hugin.h"
# include <pthread.h>
...
h_domain_t d;
...
h_domain_set_concurrency_level (d, 4);
pthread_setconcurrency (4);
...
/* do compilations, propagations, and other stuff
   that involves inference */
```

We could use *thr_setconcurrency* instead of *pthread_setconcurrency* (in that case we would include `<thread.h>` instead of `<pthread.h>`). ∎

### 1.7.2 Multiprocessing on Windows platforms

The HUGIN API can only be used in "multithreaded mode" on Windows platforms, so nothing special needs to be done for multiprocessing. (See Section 1.3 for instructions on how to use the HUGIN API in a Windows application.)

---

[4] Experiments performed on a Sun Enterprise 250 running Solaris 7 (5/99) indicates that the best performance is achieved using the POSIX threads library.

## 1.8 Using the HUGIN API in a multithreaded application

The HUGIN API can be used safely in a multithreaded application. The major obstacle to thread-safety is shared data — for example, global variables. The only global variable in the HUGIN API is the error code variable. When the HUGIN API is used in a multithreaded application, an error code variable is maintained for each thread. This variable is allocated the first time it is accessed. It is recommended that the first HUGIN API function (if any) being called in a specific thread be the *h_error_code*(18) function. If this function returns zero, it is safe to proceed (i.e., the error code variable has been successfully allocated). If *h_error_code* returns nonzero, the thread must not call any other HUGIN API function, since the HUGIN API functions critically depend on being able to read and write the error code variable. (Failure to allocate the error code variable is very unlikely, though.)

**Example 1.4** This code shows the creation of a thread, where the function executed by the thread calls *h_error_code*(18) as the first HUGIN API function. If this call returns zero, it is safe to proceed.

This example uses POSIX threads.

```
# include "hugin.h"
# include <pthread.h>
pthread_t thread;
void *data;  /* pointer to data used by the thread */

void *thread_function (void *data)
{
    if (h_error_code () != 0)
        return NULL;  /* it is not safe to proceed */

    /* now the Hugin API is ready for use */
    ...
}
...
pthread_create (&thread, NULL, thread_function, data);
```

Note that the check for *h_error_code*(18) returning zero should also be performed for the main (only) thread in a multithreaded (singlethreaded) application, when using a thread-safe version of the HUGIN API (all APIs provided by Hugin Expert A/S is thread-safe as of version 6.1). ∎

In order to create a multithreaded application, it is necessary to link with a thread library. See the previous section for instructions on how to do this. (You most likely also need to define additional compiler flags in order to get thread-safe versions of functions provided by the operating system — see the system documentation for further details.)

The most common usage of the HUGIN API in a multithreaded application will most likely be to have one or more dedicated threads to process their own domains (e.g., insert and propagate evidence, and retrieve new beliefs). In this scenario, there is no need (and is also unnecessarily inefficient) to protect each node or domain by a mutex (mutual exclusion) variable, since only one thread has access to the domain. However, if there is a need for two threads to access a common domain, a mutex must be explicitly used.

**Example 1.5** The following code fragment shows how a mutex variable is used to protect a domain from being accessed by more than one thread simultaneously. (This example uses POSIX threads.)

```
# include "hugin.h"
# include <pthread.h>
h_domain_t d;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
...
/* In Thread A: */
if (pthread_mutex_lock (&mutex) != 0)
    /* handle error */ ...;
else
{
    /* now domain ‘d’ can be used;
       for example, evidence can be entered and
       propagated, and beliefs can be retrieved;
       or, the network can be modified; etc. */
    ...
    pthread_mutex_unlock (&mutex);
}
...
/* In Thread B: */
if (pthread_mutex_lock (&mutex) != 0)
    /* handle error */ ...;
else
{
    /* use domain ‘d’ */
    ...
    pthread_mutex_unlock (&mutex);
}
```

Since domain *d* is being used by more than one thread, it is important that while one thread is modifying the data structures belonging to *d*, other threads do not attempt to read or write the same data structures. This is achieved by requiring all threads to lock the *mutex* variable while they access the data structures of *d*. The thread library ensures that only one thread at a time can lock the *mutex* variable. ∎

Many HUGIN API functions that operate on nodes also modify the state of the domain or class to which the nodes belong. For example, entering evidence to a node clearly modifies the state of the node, but it also modifies book-keeping information relating to evidence within the domain to which the node belongs.

On the other hand, many HUGIN API functions only read attributes of a class, domain, or node. Such functions can be used simultaneously from different threads on the same or related objects, as long as it has been ensured that no thread is trying to modify the objects concurrently with the read operations. Examples of functions that only read attributes are: *h_node_get_category*(30), *h_domain_get_attribute*(43), *h_node_get_belief*(108), etc.

In general, all functions with *_get_* or *_is_* as part of their names do not modify data, unless their descriptions explicitly state that they do. Examples of the latter category are:

- *h_node_get_name*(39) and *h_class_get_name*(49) will assign names to the node or class, if no name has previously been assigned. (If the node or class is known to be named, then these functions will not modify data.)

- *h_node_get_table*(36), *h_node_get_experience_table*(142), and *h_node_get_fading_table*(143) will create a table if one doesn't already exist.

- *h_domain_get_marginal*(109) and *h_node_get_distribution*(110) must, in the general case, perform a propagation (which needs to modify the junction tree).

- All HUGIN API functions returning a list of nodes may have to allocate and store the list.

# Chapter 2

# Nodes and Domains

The functions described in this chapter allow an application to construct and modify "flat" belief network and LIMID models, known as *domains*. Chapter 3 provides functions for constructing object-oriented models for belief networks and LIMIDs. An object-oriented model must be converted to a domain before it can be used for inference.

A large part of the functions (those that operate on nodes) described in this chapter can also be used for nodes in object-oriented models. If special conditions apply when a function is used for such nodes, they are stated in the description of the function (usually prefixed with "OOBN:").[1]

## 2.1 Types

Nodes and domains are the fundamental objects used in the construction of belief network and LIMID models in HUGIN. The HUGIN API introduces the opaque pointer types **h_node_t** and **h_domain_t** to represent these objects.

### 2.1.1 Node category

In ordinary belief networks, all nodes represent random variables. But, in LIMIDs, nodes also represent decisions and utility functions. And in object-oriented models, nodes also represent class instances — see Section 3.8.

In all of these network models, so-called *function* nodes can be created: A function node represents a real-valued function that depends on (some or all of) the parents of the node. Function nodes are not (directly) involved in the inference process — evidence cannot be specified for function nodes,

---

[1] Although slightly misleading, we use the abbreviation OOBN to refer to object-oriented belief networks as well as object-oriented LIMIDs.

but the function associated with the node can be evaluated using the results of inference or simulation as input.

In order to distinguish between the different types of nodes, the HUGIN API associates with each node a *category*, represented as a value of the enumeration type **h_node_category_t**. The constants of this enumeration type are:

- *h_category_chance* (for nodes representing random variables),
- *h_category_decision* (for nodes representing decisions),
- *h_category_utility* (for nodes representing utility functions),
- *h_category_function* (for function nodes), and
- *h_category_instance* (for nodes representing class instances in object-oriented models).

In addition, the special constant *h_category_error* is used for handling errors.

### 2.1.2 Node kind

Another grouping of nodes exists, called the *kind*[2] of a node. This grouping is a characterization of the state space of the node. The HUGIN API introduces the enumeration type **h_node_kind_t** to represent it.

Chance and decision nodes are either *discrete* or *continuous*.[3] The enumeration constants *h_kind_discrete* and *h_kind_continuous* represent those kinds. Discrete nodes have a finite number of states. Continuous nodes are real-valued and have a special kind of distribution, known as a *Conditional Gaussian* (CG) distribution, meaning that the distribution is Gaussian (also known as 'normal') given values of the parents. For this reason, continuous nodes are also referred to as CG nodes. (See Appendix A for further information on CG variables.)

For completeness, a third kind *other* (represented by the enumeration constant *h_kind_other*) has been introduced in order to provide a "kind" for the remaining three node categories.

In addition, the special constant *h_kind_error* is used for handling errors.

## 2.2 Domains: Creation and deletion

A *domain* is used to hold all information associated with a (non-OOBN) network model.

---

[2] The terms *category* and *kind* have been deliberately chosen so as not to conflict with the traditional vocabulary used in programming languages. Thus, the term 'type' was ruled out.

[3] Currently, the HUGIN API does not support LIMIDs with continuous nodes. Thus, all (chance and decision) nodes of a LIMID must be discrete.

▸ **h_domain_t** ***h_new_domain*** **(void)**

Create a new empty domain. If creation fails, NULL is returned.

When a domain is no longer needed, the internal memory used by the domain can be reclaimed and made available for other purposes.

▸ **h_status_t** ***h_domain_delete*** (**h_domain_t** *domain*)

Release all (internal) memory resources used by *domain*.

All existing references to objects owned by *domain* (for example, nodes) are invalidated by this operation, since those objects are also deleted.

A domain can also be created by cloning an existing domain.

▸ **h_domain_t** ***h_domain_clone*** (**h_domain_t** *domain*)

Create a clone of *domain*. The clone will be identical to *domain*, except that the clone will not be compiled (even if *domain* is compiled).[4]

## 2.3 Nodes: Creation and deletion

The following function is used for creating nodes in a domain. Only chance, decision, utility, and function nodes are permitted in a domain.

▸ **h_node_t** ***h_domain_new_node***
(**h_domain_t** *domain*, **h_node_category_t** *category*,
**h_node_kind_t** *kind*)

Create a new node of the specified *category* and *kind* within *domain*. The new node has default (or no) values assigned to its attributes: for example, it has no name, it has just one state (if the node is a discrete node), and it has no table. The attributes of the new node must be explicitly set using the relevant API functions.

If the new node is not a function node, and *domain* is compiled, the corresponding compiled structure is removed since it no longer reflects the domain (see Section 6.5).

If an error occurs, the function returns NULL.

▸ **h_domain_t** ***h_node_get_domain*** (**h_node_t** *node*)

Retrieve the domain to which *node* belongs. If *node* is NULL, or it belongs to a class, NULL is returned.

---

[4] Chapter 6 provides information on compiling domains — a prerequisite for performing inference.

▸ **h_node_category_t *h_node_get_category*** (**h_node_t** *node*)

Return the category of *node*. If *node* is NULL, *h_category_error* is returned.

▸ **h_node_kind_t *h_node_get_kind*** (**h_node_t** *node*)

Return the kind of *node*. If *node* is NULL, *h_kind_error* is returned.

The following function is intended for modifying a network. If a complete domain is to be disposed off, use *h_domain_delete*(29) instead.

▸ **h_status_t *h_node_delete*** (**h_node_t** *node*)

Delete *node* (and all links involving *node*) from the domain or class to which *node* belongs. If *node* has children, the tables and models of those children are adjusted (see *h_node_remove_parent*(32) for a description of the adjustment procedure).

If *node* is not a function node, and it belongs to a domain, then that domain is "uncompiled" (see Section 6.5).

OOBN: Special actions are taken if *node* is an interface node, an instance node, or an output clone. See Section 3.7 and Section 3.8 for further details.

A new node can also be created by cloning an existing node.

▸ **h_node_t *h_node_clone*** (**h_node_t** *node*)

Create a clone of *node*. The clone belongs to the same domain or class as *node*, and it has attributes that are identical to (or clones of) the corresponding attributes of *node*: category, kind, subtype, number of states, state labels, state values, parents, tables (conditional probability, policy, utility, experience, and fading), model, case data, evidence, structure learning constraints, label, and user-defined attributes. However, the user data pointer is not copied (it is set to NULL for the clone).

The clone has no name (because there cannot be two identically named nodes in the same domain or class). Also, the clone has no children (because that would imply changes to the children).

OOBN: If *node* is an interface node or an output clone, then the clone has none of these properties.

OOBN: If *node* is a class instance, then the clone is an instance of the same class as *node* and has the same inputs as *node*.

OOBN: If *node* belongs to a domain derived from a class (a so-called "runtime" domain), then the "source list" of *node* is *not* copied to the clone.

If (and only if) the cloning process succeeds, *node* is not a function node, and it belongs to a domain, then that domain is "uncompiled" (see Section 6.5).

## 2.4 The links of the network

The links of a belief network or a LIMID are directed edges between the nodes of the network. [Undirected edges are also possible, but the API interface to support them has not yet been defined. However, see Chapter 12 for a description of the NET language interface.]

If there exists a directed edge from a node $u$ to a node $v$, we say that $u$ is a *parent* of $v$ and that $v$ is a *child* of $u$. The HUGIN API provides functions for adding and removing parents to/from a node, replacing a parent with another compatible parent, reversing a directed edge between two nodes, as well as functions for retrieving the current set of parents and children of a node.

The semantics of links depend on the categories of the nodes involved. For chance nodes, the incoming links represent probabilistic dependence: The distribution of a chance node is conditionally independent of all non-descendants of the node given its parents. For decision nodes, the incoming links represent availability of information: The parents of a decision node are assumed to be known when the decision is made. For utility nodes, the parents represent the set of nodes that the utility function depends on. For function nodes, the parents represent the set of nodes that the function associated with the node is allowed to depend on.

Chance, decision, and utility nodes can only have chance and decision nodes as parents. Nodes representing class instances must have no parents and no children.

The network cannot be an arbitrary directed graph. It must be *acyclic*.

It is not possible to link nodes from different domains or classes.

The quantitative part of the relationship between a chance, a decision, or a utility node and its parents is represented as a *table* (see Section 2.6). This table can either be specified directly (as a complete set of numbers), or it can be generated from a *model* (see Chapter 5). For function nodes, only models can be used. When links are added, removed, or reversed, the tables and models involved are automatically updated.

▸ **h_status_t** ***h_node_add_parent*** (**h_node_t** *child*, **h_node_t** *parent*)

Add *parent* as a new parent of *child* (i.e., add a directed link from *parent* to *child*). If *child* is a chance, a decision, or a utility node, then *parent* must be a chance or a decision node. And if *child* is discrete, then *parent* must also be discrete.

OOBN: *child* and *parent* must not be class instances, and *child* must not be an input node or an output clone.

If adding the link would create a directed cycle in the network, the operation is not performed.

The tables[5] of *child* are updated as follows: The value associated with a given state configuration of the new table is the same as the value associated with the corresponding state configuration (that is, exclusive the value of *parent*) of the old table.

The model (if any) of *child* is not affected by this operation.

Finally, if *child* is not a function node, and *child* and *parent* belong to a domain, then that domain is "uncompiled" (see Section 6.5).

▸ **h_status_t *h_node_remove_parent*** (**h_node_t** *node*, **h_node_t** *parent*)

Delete the directed link from *parent* to *node*. The tables of *node* are updated as follows: If *parent* is discrete, the contents of the updated table are the parts of the old table corresponding to state 0 of *parent* (see Section 2.5). If *parent* is continuous, the $\beta(i)$-parameters (see *h_node_set_beta*(38)) corresponding to the *parent* → *node* link are deleted from the table.

The model (if any) of *node* is updated as follows: If *parent* is a "model node" (see Section 5.4), the model is deleted. Otherwise, all expressions that refer to *parent* are deleted from the model.

Finally, if *node* is not a function node and belongs to a domain, then that domain is "uncompiled" (see Section 6.5).

▸ **h_status_t *h_node_switch_parent***
(**h_node_t** *node*, **h_node_t** *old_parent*, **h_node_t** *new_parent*)

Substitute *new_parent* for *old_parent* as a parent of *node*, while preserving the validity of the tables and model of *node* (all references to *old_parent* are replaced by references to *new_parent*). The *old_parent* and *new_parent* nodes must be "compatible"—see below for the definition of compatibility.

If switching parents would create a directed cycle in the network, the operation is not performed.

As usual, if *node* is not a function node, and it belongs to a domain, then that domain is "uncompiled" (see Section 6.5).

In order for two nodes to be *compatible*, the following conditions must hold: The nodes must have

- the same category and kind;
- the same subtype and the same number of states (if the nodes are discrete);
- the same list of state labels (if the nodes are labeled);
- the same list of state values (if the nodes are numbered or of interval subtype).

---

[5] In addition to a conditional probability table, two other tables can be associated with a chance node: An *experience* and a *fading* table (see Section 10.1) can be created for the purpose of parameter learning. All tables are updated in the same manner.

The motivation for this definition is that compatible nodes should be interchangeable with respect to the table generator (see Chapter 5). That is, replacing one node in a model with another compatible node should not affect the table produced by the table generator. That is also the reason for requiring the lists of state labels to be identical only for labeled nodes, although all discrete nodes can have state labels.

▸ **h_status_t *h_node_reverse_edge*** (**h_node_t** *node*$_1$, **h_node_t** *node*$_2$)

Reverse the directed edge between *node*$_1$ and *node*$_2$ (which must be chance nodes of the same kind). This is done such that the joint probability distribution defined by the modified network is equivalent to the joint probability distribution defined by the original network. In order to accomplish this, *node*$_1$ inherits the parents of *node*$_2$ (except *node*$_1$, of course), and vice-versa for *node*$_2$.

OOBN: The nodes must not be input nodes or output clones.

The operation is not performed, if reversal of the edge would create a directed cycle in the network.

The experience and fading tables (see Section 10.1) as well as models (if any) of *node*$_1$ and *node*$_2$ are deleted.

Finally, if the nodes belong to a domain, then that domain is “uncompiled” (see Section 6.5).

▸ **h_node_t** $*$***h_node_get_parents*** (**h_node_t** *node*)

Return the parents of *node* (as a NULL-terminated list). If an error occurs, NULL is returned.

The list of parents is stored within the node data structure. The application must not modify or deallocate this list.

When a new discrete parent is added to *node*, the parent is added at the front of the list of parents. A new continuous parent is added at the end of the list.

**Example 2.1** The following code prints the names of the parents of a node:

```
h_node_t n, *parents, *p;
...
if ((parents = h_node_get_parents (n)) == 0)
    /* handle error */;
else
{
    printf ("Parents of %s:\n", h_node_get_name (n));
    for (p = parents; *p != 0; p++)
        printf ("%s\n", h_node_get_name (*p));
}
```

∎

If the list returned by *h_node_get_parents* (or *h_node_get_children*) is used to control the iteration of a loop (such as the for-loop in the example above), then API functions that modify the list must not be used in the body of the loop. For example, calling *h_node_add_parent*(31) modifies the list of parents of the child and the list of children of the parent: The contents of the lists are obviously modified, but the memory locations of the lists might also change. Other similar functions to watch out for are *h_node_remove_parent*(32), *h_node_switch_parent*(32), *h_node_reverse_edge*(33), and *h_node_delete*(30).

The problem can be avoided if a copy of the list is used to control the loop.

▸ **h_node_t** ∗***h_node_get_children*** (**h_node_t** *node*)

Return the children of *node* (as a NULL-terminated list). If an error occurs, NULL is returned.

The list of children is stored within the node data structure. The application must not modify or deallocate this list.

### 2.4.1 The requisite parents of decision nodes

Not all available observations matter when a decision must be made. Intuitively, a parent of a decision node is said to be *requisite* if the value of the parent may affect the optimal choice of the decision.

The performance of inference in a LIMID can be improved, if the network is simplified by removing the nonrequisite parents of all decision nodes.

Lauritzen and Nilsson [20] present an algorithm for removing the nonrequisite parents of the decision nodes in a LIMID. The result of this algorithm is a LIMID, where no parent of a decision node is nonrequisite. This network is known as the *minimal reduction* of the LIMID.[6]

▸ **h_node_t** ∗***h_node_get_requisite_parents*** (**h_node_t** *node*)

Return a list of the requisite parents of *node* (which must be a decision node belonging to a domain). If an error occurs, NULL is returned.

If the requisite parents are not already known (see below), the function computes the minimal reduction of the underlying LIMID network. The parents of *node* in this minimal reduction are returned.

Notice that the function does not remove the nonrequisite parents — it only tells which parents are (non)requisite. In order to remove the nonrequisite parents, *h_node_remove_parent*(32) must be used.

In order to improve performance, the results of the minimal reduction algorithm are cached (that is, the requisite parents of all decisions are cached).

---

[6] Our definition of requisiteness is slightly different than the one given by Lauritzen and Nilsson: We define a parent of a decision node to be *requisite* if and only if it is also a parent of the decision node in the minimal reduction of the LIMID.

If a link is added, or if a link is removed (unless that link represents a nonrequisite parent of a decision node), the cached results are deleted. Notice that deletion of nodes usually removes links (but creation of nodes does not add links).

The list of requisite parents is stored within the node data structure. The application must not modify or deallocate this list. Also, this list is deleted by the HUGIN API when the contents become invalid (as explained above). In this case, an updated list must be requested using *h_node_get_requisite_parents*.

## 2.5 The number of states of a node

As mentioned above, discrete nodes in the HUGIN API has a finite number of states. The enumeration of the states follows traditional C conventions: If a node has $n$ states, the first state has index $0$, the second state has index $1$, $\dots$, and the last state has index $n - 1$.

The following function is used to specify the number of states of a discrete node.

▸ **h_status_t** ***h_node_set_number_of_states*** (**h_node_t** *node*, **size_t** *count*)

Set the number of states of *node* to *count* (*node* must be a discrete node, and *count* must be a positive integer). Unless *count* is equal to the current number of states of *node*, any evidence specified for *node* (see Section 8.1.1) is removed, and if *node* belongs to a domain then that domain is "uncompiled" (see Section 6.5).

OOBN: *node* must not be an output clone.

If *node* is a boolean node, then *count* must be 2 (that is, a boolean node must always have two states).

Changing the number of states of a node has implications for all tables in which the node appears. The affected tables are the table associated with the node itself and the tables associated with the children of the node. (See Section 2.6 and Section 10.1 for more information about those tables.)

Let $\langle N_1, \dots, N_k, \dots, N_l \rangle$ be the node list of some table for which the number of states of node $N_k$ is being changed from $n_k$ to $m_k$, and let $\langle i_1, \dots, i_{k-1}, i_k, i_{k+1}, \dots, i_l \rangle$ be a configuration of that table (see Section 4.1 for an explanation of node lists and configurations). If $m_k < n_k$, the updated table is obtained by deleting the data associated with the configurations for which $i_k \geq m_k$. If $m_k > n_k$, the updated table is obtained by copying the data associated with the configuration $\langle i_1, \dots, i_{k-1}, 0, i_{k+1}, \dots, i_l \rangle$ to the new configurations (i.e., the configurations for which $i_k \geq n_k$).

Changing the number of states of a node might also have implications for the models of the children of the node. In the current implementation, if the node appears as a "model node" in the model of a child, then all expressions in the model are deleted.[7]

If *count* is smaller than the current number of states of *node*, then the labels and the values associated with the deleted states are deleted.

OOBN: If *node* is an output node of its class, then the changes described above are applied to all its output clones (recursively, if output clones are themselves output nodes).

▸ **h_count_t** ***h_node_get_number_of_states*** (**h_node_t** *node*)

Return the number of states of *node* (which must be a discrete node). If an error occurs, a negative number is returned.

## 2.6 The table of a node

A *table* is associated with each (chance, decision, and utility) node[8] in a belief network or a LIMID model. This table has different semantics depending on the category of the node:

- For a chance node, the table is a conditional probability table (CPT).
- For a decision node, the table represents a policy that prescribes what the decision maker should do given observations of the parents. This table is similar to a CPT (i.e., the table contains conditional probability distributions, but the distributions are usually deterministic).
- For a utility node, the table represents a utility function.

OOBN: Nodes representing class instances do not have tables. Also, output clones do not have tables.

For utility and discrete (chance and decision) nodes, the contents of the tables can be specified directly (by writing to the data arrays of the tables) or indirectly (using the table generator — see Chapter 5). For continuous nodes, the parameters of the CG distributions must be specified using appropriate functions.

▸ **h_table_t** ***h_node_get_table*** (**h_node_t** *node*)

Retrieve the table associated with *node* (which must be a chance, a decision, or a utility node). If an error occurs, NULL is returned.

OOBN: *node* must not be a class instance or an output clone.

---

[7] The models should really be resized in the same way as the tables.

[8] Instead of tables, function nodes use models to express the (functional) relationship between the child and its parents. See Chapter 5.

The returned table is the real thing (i.e., not a copy), and the contents of the table can be modified using functions that provide access to the internal data structures of tables (see Chapter 4).

The node list of the table contains the discrete parents of *node*, followed by *node* (if *node* is not a utility node), followed by the continuous parents of *node*. The order of the parents in this node list is the same as the order in the node list returned by *h_node_get_parents*$^{(33)}$, unless the order has been changed — see *h_table_reorder_nodes*$^{(64)}$. (The node list determines the configurations of the table — see Section 4.1.)

The table is not created until it is needed (for example, requested by this function or required in order to compile a domain). It is recommended to specify the parents of a node before specifying its table as this simplifies the creation of links (because then the table won't have to be resized for each link being added).

However, if the table has been created, and the set of parents of *node* is changed, or the number of states of *node* or one of its parents is changed, then the table is resized accordingly. Because of that, any pointers to the internal data structures of the table (for example, a pointer to the data array of the table) that the application may be holding become invalid and must be retrieved again from the table.

If *node* is a discrete node, the entries of the table corresponding to a given parent state configuration should be nonnegative and sum to 1. If not, the inference engine (that is, the propagation operation) will adjust (normalize) the table entries in order to satisfy this condition. There are no such restrictions on utility tables.

It is also possible to specify the contents of a node table indirectly through the table generation facility (see Chapter 5). If this facility is used for some node table, then the contents of that table is generated from a mathematical description of the relationship between the node and its parents. It is possible to modify the contents generated from such a description, but note that the inference engine will regenerate the table when certain parameters are changed (see Section 5.8 for precise details).

If the contents of a table is changed, the updated table will be used by the inference engine, provided it has been notified of the change. HUGIN API functions that change node tables automatically provide this notification. However, for changes made by storing directly into a table (i.e., using the array pointer returned by *h_table_get_data*$^{(62)}$ for storing values), an explicit notification must be provided. The following function does this.

▸ **h_status_t** ***h_node_touch_table*** (**h_node_t** *node*)

Notify the inference engine that the table of *node* has been modified by storing directly into its data array. This notification must be provided before subsequent calls to other HUGIN API functions.

*Omission of such a notification may cause the inference engine to malfunction!*

**Example 2.2** This piece of code shows how to specify the probability table for the variable $A$ in the “Chest Clinic” belief network [21]. This variable is binary and has no parents: $P(A = \mathit{yes}) = 0.01$ and $P(A = \mathit{no}) = 0.99$.

```
h_node_t A;
h_table_t table = h_node_get_table (A);
h_number_t *data = h_table_get_data (table);

data[0] = 0.01;
data[1] = 0.99;

h_node_touch_table (A);
```

∎

The conditional distribution for a continuous random variable $Y$ with discrete parents $I$ and continuous parents $Z$ is a (one-dimensional) Gaussian distribution conditional on the values of the parents:

$$p(Y|I = i, Z = z) = \mathcal{N}(\alpha(i) + \beta(i)^{T}z, \gamma(i))$$

[This is known as a *CG distribution*.] Note that the mean depends linearly on the continuous parent variables and that the variance does not depend on the continuous parent variables. However, both the linear function and the variance are allowed to depend on the discrete parent variables. (These restrictions ensure that exact inference is possible.)

The following six functions are used to set and get the individual elements of the conditional distribution for a continuous node. In the prototypes of these functions, *node* is a continuous chance node, *parent* is a continuous parent of *node*, *i* is the index of a discrete parent state configuration[9] (see Section 4.1 for an explanation of configuration indexes), and *alpha*, *beta*, and *gamma* refer to the $\alpha(i)$, $\beta(i)$, and $\gamma(i)$ components of a CG distribution as specified above.

▶ **h_status_t** ***h_node_set_alpha***
(**h_node_t** *node*, **h_double_t** *alpha*, **size_t** *i*)

▶ **h_status_t** ***h_node_set_beta***
(**h_node_t** *node*, **h_double_t** *beta*, **h_node_t** *parent*, **size_t** *i*)

▶ **h_status_t** ***h_node_set_gamma***
(**h_node_t** *node*, **h_double_t** *gamma*, **size_t** *i*)

Here, *gamma* must be nonnegative.

---

[9] If *node* has no discrete parents, then there is only one configuration — the empty configuration. In this case, *i* must be 0.

▶ **h_double_t** ***h_node_get_alpha*** (**h_node_t** *node*, **size_t** *i*)

▶ **h_double_t** ***h_node_get_beta*** (**h_node_t** *node*, **h_node_t** *parent*, **size_t** *i*)

▶ **h_double_t** ***h_node_get_gamma*** (**h_node_t** *node*, **size_t** *i*)

For the last three functions: If an error is detected, a negative value is returned, but this is not of any use for error detection (except for *h_node_get_gamma*), since any real value is valid for both the $\alpha(i)$ and $\beta(i)$ parameters. Thus, errors must be checked for using the *h_error_code*(18) function.

## 2.7 The name of a node

The HUGIN system uses a number of text files:

- NET files for storing specifications of networks,
- data files for storing learning data,
- case files for storing a single case,
- node list files (e.g., for storing elimination orders for triangulations),
- log files for generating logs of actions performed by some HUGIN API calls (e.g., compilation operations).

In order to refer to a node in text written to any of these files, the node must be named. If the node doesn't have a name, a name is automatically assigned as part of the operation that generates the text. But nodes can also be explicitly named using the following function.

▶ **h_status_t** ***h_node_set_name*** (**h_node_t** *node*, **h_string_t** *name*)

Create a copy of *name* and assign it to *node*. No other node in the network (a domain or a class) to which *node* belongs must have the same name. The name must have the same form as a C identifier, except that if *node* belongs to a domain, then the name can be a sequence of identifiers, separated by single dots (periods) — see Section 12.7. The reason for this exception is to permit naming (using "meaningful" names) of nodes of a runtime domain.

▶ **h_string_t** ***h_node_get_name*** (**h_node_t** *node*)

Retrieve the name of *node*. If *node* has not previously been assigned a name, a valid name is automatically assigned. If an error occurs, NULL is returned.

Note that the name returned by *h_node_get_name* is not a copy. Thus, the application must not modify or free it.

▸ **h_node_t *h_domain_get_node_by_name***
(**h_domain_t** *domain*, **h_string_t** *name*)

Return the node with name *name* in *domain*, or NULL if no node with that name exists in *domain*, or if an error occurs (i.e., *domain* or *name* is NULL).

## 2.8 Iterating through the nodes of a domain

An application may need to perform some action for all nodes of a domain. To handle such situations, the HUGIN API provides a set of functions for iterating through the nodes of a domain, using an order determined by the age of the nodes: the first node in the order is the youngest node (i.e., the most recently created node that hasn't been deleted), . . . , and the last node is the oldest node.

If the application needs the nodes in some other order, it must obtain all the nodes, using the functions described below, and sort the nodes according to the desired order.

▸ **h_node_t *h_domain_get_first_node*** (**h_domain_t** *domain*)

Return the first node of *domain*, using the order described above, or NULL if *domain* contains no nodes, or *domain* is NULL (this is considered an error).

▸ **h_node_t *h_node_get_next*** (**h_node_t** *node*)

Return the node that follows *node* in the order, or NULL if *node* is the last node in the order, or if *node* is NULL (which is considered an error).

**Example 2.3** This function counts the nodes of a given domain.

```
int count_nodes (h_domain_t d)
{
    h_node_t n;
    int count = 0;

    for (n = h_domain_get_first_node (d); n != 0;
         n = h_node_get_next (n))
       count++;
    return count;
}
```

■

## 2.9 User data

Applications sometimes need to associate data with the nodes of a domain (or the domain itself). Examples of such data: the window used to display the beliefs of a node, the last time the display was updated, the external source used to obtain findings for a node, etc.

The HUGIN API provides two ways to associate user data with domains and nodes:

- as arbitrary data, managed by the user, or
- as attributes (key/value pairs — where the key is an identifier, and the value is a character string), managed by the HUGIN API.

### 2.9.1 Arbitrary user data

The HUGIN API provides a slot within the node structure for exclusive use by the application. This slot can be used to hold a pointer to arbitrary data, completely controlled by the user.

▸ **h_status_t** ***h_node_set_user_data*** (**h_node_t** *node*, **void** ∗*p*)

Store the pointer *p* in the user data slot of *node*.

▸ **void** ∗***h_node_get_user_data*** (**h_node_t** *node*)

Return the value stored in the user data slot of *node*. If no value has been stored, the stored value is NULL, or *node* is NULL (this is an error), NULL is returned.

No other functions in the HUGIN API touch the user data slots of the nodes.

**Example 2.4** In an application displaying the beliefs of nodes in windows, each node will have a window associated with it. The simplest way to keep track of these belief windows is to store them in the user data fields of the nodes.

Creating and storing a belief window can be done in the following way:

```
belief_window w;
h_node_t n;
...
w = create_belief_window (n);
h_node_set_user_data (n, (void*) w);
```

where *create_belief_window* is a function defined by the application. Note the cast to type **void** ∗ in the call to *h_node_set_user_data* (for this to work properly, *belief_window* should be a pointer type).

Now, the belief window can be used in the following way:

```
belief_window w;
h_node_t n;
...
w = (belief_window) h_node_get_user_data (n);
update_belief_window (w, n);
```

where *update_belief_window* is a function defined by the application. Again, note the cast of the pointer type. ∎

Using the user data facility is analogous to adding an extra slot to the node data structure. It must be noted that only one such slot is available. If more are needed, store a list of slots or create a compound data type (e.g., a C structure). Note also that the extra slot is *not* saved in HUGIN KB or in NET files. If this is needed, the application must create the necessary files. Alternatively, the attribute facility described below can be used.

It is also possible to associate user data with a domain as a whole. This is done using the following functions.

▸ **h_status_t** ***h_domain_set_user_data*** (**h_domain_t** *domain*, **void** ∗*p*)

Store the pointer *p* in the user data slot of *domain*.

▸ **void** ∗***h_domain_get_user_data*** (**h_domain_t** *domain*)

Return the value stored in the user data slot of *domain*. If no value has been stored, the stored value is NULL, or *domain* is NULL (this is an error), NULL is returned.

### 2.9.2 User-defined attributes

In the previous section, we described a way to associate arbitrary data with a node or a domain object. That data can be anything (for example, a list or a tree). However, if more than one data object is required, the user must build and maintain a data structure for the objects himself. Moreover, the data are not saved in the HUGIN KB (Section 2.10) and the NET files (Chapter 12).

Sometimes we need the ability to associate several user-specified data objects with domains and nodes *and* to have these data objects saved in the HUGIN KB and the NET files. The HUGIN API provides this feature but at the cost of requiring the data to be *C-strings* (i.e., sequences of non-null characters terminated by a null character). Each data object (string) is stored under a name — a *key*, which must be a C-identifier (i.e., a sequence of letters and digits, starting with a letter, and with an underscore counting as a letter).

The HUGIN GUI tool stores strings using the UTF-8 encoding. If you intend to have your networks loaded within that tool, you should make sure to store your attributes using the UTF-8 encoding.

▸ **h_status_t** ***h_node_set_attribute***
(**h_node_t** *node*, **h_string_t** *key*, **h_string_t** *value*)

Insert (or update, if *key* is already defined) the *key*/*value*-pair in the attribute list for *node*. If *value* is NULL, the attribute is removed.

OOBN: If *node* is an output clone, then the attribute is not saved if the class is saved as a NET file (because the NET file format doesn't support that).

▸ **h_string_t** ***h_node_get_attribute*** (**h_node_t** *node*, **h_string_t** *key*)

Lookup the value associated with *key* in the attribute list for *node*. If *key* is not present, or if an error occurs, NULL is returned.

The string returned by *h_node_get_attribute* is stored in the attribute list for *node*. The application must not deallocate this string.

The following two functions perform similar operations on domains.

▸ **h_status_t** ***h_domain_set_attribute***
(**h_domain_t** *domain*, **h_string_t** *key*, **h_string_t** *value*)

▸ **h_string_t** ***h_domain_get_attribute***
(**h_domain_t** *domain*, **h_string_t** *key*)

If you want to create your own attributes, pick some attribute names that are not likely to clash with somebody elses choices (or with names that the HUGIN API or the HUGIN GUI tool might use in the future). For example, use a common prefix for your attribute names.

In order to access the values of attributes, one must know the names of the attributes in advance. The following set of functions provides a mechanism for iterating over the list of attributes associated with a given node or domain.

The notion of an attribute as a key/value pair is represented by the opaque pointer type **h_attribute_t**.

▸ **h_attribute_t** ***h_node_get_first_attribute*** (**h_node_t** *node*)

Retrieve the first attribute object for *node*. If the attribute list is empty (or *node* is NULL), NULL is returned.

▸ **h_attribute_t** ***h_domain_get_first_attribute*** (**h_domain_t** *domain*)

Retrieve the first attribute object for *domain*. If the attribute list is empty (or *domain* is NULL), NULL is returned.

The attributes returned by these functions are actual objects within the attribute lists of *node* or *domain*. Do not attempt to deallocate these objects.

▸ **h_attribute_t** ***h_attribute_get_next*** (**h_attribute_t** *attribute*)

Retrieve the attribute object that follows *attribute* in the attribute list containing *attribute*. If *attribute* is the last object in the list, or *attribute* is NULL (this is a usage error), NULL is returned.

Given an attribute object, the following functions can be used to retrieve the key and value parts of the attribute.

▸ **h_string_t** ***h_attribute_get_key*** (**h_attribute_t** *attribute*)

▶ **h_string_t** ***h_attribute_get_value*** (**h_attribute_t** *attribute*)

Retrieve the key or value associated with *attribute*. (These are the actual strings stored within the attribute — do not modify or deallocate them.)

Note that *h_node_set_attribute*(42) and *h_domain_set_attribute*(43) modify (or even delete, if the *value* argument is NULL) objects in the attribute lists for the affected node or domain. If the application is holding (a pointer to) an attribute, and then calls one of these functions to change the value of (or even delete) the attribute, then (pointers to) the old value (or the attribute itself, if the *value* argument was NULL) are no longer valid. These facts should be kept in mind when iterating over attribute lists.

## 2.10 HUGIN Knowledge Base files

When a domain has been created, it can be saved to a file in a portable binary format. Such a file is known as a *HUGIN Knowledge Base* (*HUGIN KB*, or simply *HKB*, for short) file. By convention, we name such files using the extension `.hkb`. (A domain can also be saved in a textual format, called the NET format — see Chapter 12.)

When a domain is loaded from an HKB file, the "state" of the domain is generally the same as when it was saved — with the following exceptions:

- The case data used by the structure learning and EM algorithms (see Chapter 11) are not stored in the HKB file.
- If the domain is compiled (but not compressed), then the contents of the HKB file include only information about the structure of the junction trees, not the numerical data stored in the tables of the junction trees. In other words, a compiled domain is saved as if it was only triangulated. When the domain is loaded from the HKB file, it must be explicitly compiled (using *h_domain_compile*(89)) before it can be used for inference[10] — this will use the already constructed junction trees.
- If the domain is compressed (which implies compiled), then the domain will also be compressed when loaded. This property implies that compressed domains can be created on computers with large amounts of (virtual) memory and then later be loaded on computers with limited amounts of (virtual) memory.

---

[10] This is a change in HUGIN API version 6.6. In previous versions of the HUGIN API, the compilation was attempted as part of the load operation — except in very old versions, where the table data were actually included in the HKB file, and therefore the table data could be loaded directly from the file.

When a compressed domain is loaded, a propagation is required before beliefs can be retrieved.[11]

- If the domain is a triangulated influence diagram (i.e., the HKB file was created by a HUGIN API older than version 7.0), then the domain is loaded in uncompiled form. The domain is then treated as a LIMID.

There is no published specification of the HKB format, and since the format is binary (and non-obvious), the only way to load an HKB file is to use the appropriate HUGIN API function. This property makes it possible to protect HKB files from unauthorized access: A *password* can be embedded in the HKB file, when the file is created; this password must then be supplied, when the HKB file is loaded. (The password is embedded in the HKB file in "encrypted" form, so that the true password cannot easily be discovered by inspection of the HKB file contents.)

In general, the format of an HKB file is specific to the version of the HUGIN API that was used to create it. Thus, when upgrading the HUGIN API (which is also used by the HUGIN GUI tool, so upgrading that tool usually implies a HUGIN API upgrade), it may be necessary to save a domain in the NET format (see Section 12.9) using the old software before upgrading to the new version of the software (because the new software may not be able to load the old HKB files).[12]

HUGIN KB files are (as of HUGIN API version 6.2) automatically compressed using the Zlib library (`www.zlib.net`). This implies that the developer (i.e., the user of the HUGIN API) must (on some platforms) explicitly link to the Zlib library, if the application makes use of HKB files — see Section 1.2.

▸ **h_status_t** ***h_domain_save_as_kb***
(**h_domain_t** *domain*, **h_string_t** *file_name*, **h_string_t** *password*)

Save *domain* as a HUGIN KB to a file named *file_name*. If *password* is not NULL, then the HKB file will be protected by the given password (i.e., the file can only be loaded if this password is supplied to the *h_kb_load_domain* function).

---

[11] This is also a change in HUGIN API version 6.6. See the previous footnote (replacing "compilation" with "propagation").

[12] The HKB formats for HUGIN API versions 3.*x* and 4.*x* were identical, but the HKB format changed for version 5.0 and again for versions 5.1, 5.2, and 5.3. Versions 5.4, 6.0, and 6.1 used the same format as version 5.3. Versions 6.2–6.5 also used this format for HKB files that were not password protected, but a newer revision of the format was used for password protected HKB files. The HKB format changed for versions 6.6, 6.7, 7.0, and 7.1. Version 7.2 used the same format as version 7.1. Version 7.3 also uses this format for networks without function nodes, but a newer revision of the format is used for networks with function nodes. HUGIN API 7.3 can load HKB files produced by version 5.0 or any later version up to (at least) version 7.3 — but future versions of the HUGIN API might not.

▸ **h_domain_t** ***h_kb_load_domain***
(**h_string_t** *file_name*, **h_string_t** *password*)

Load a domain from the HUGIN KB file named *file_name*. A reference to the loaded domain is returned. In case of errors, NULL is returned.

If the HKB file is password protected, then the *password* argument must match the password used to create the HKB file (if not, the load operation will fail with an "invalid password" error). If the HKB file is not protected, the *password* argument is ignored.

The name of the file from which a domain was loaded or to which it was saved is stored within the domain object. The file name used in the most recent load or save operation can be retrieved using *h_domain_get_file_name*(181).

# Chapter 3

# Object-Oriented Belief Networks and LIMIDs

This chapter provides the tools for constructing object-oriented belief network and LIMID models.

An object-oriented model is described by a *class*. Like a domain, a class has an associated set of nodes, connected by links. However, a class may also contain special nodes representing instances of other classes. A class instance represents a network. This network receives input through *input nodes* and provides output through *output nodes*. Input nodes of a class are "placeholders" to be filled-in when the class is instantiated. Output nodes can be used as parents of other nodes within the class containing the class instance.

Object-oriented models cannot be used directly for inference: An object-oriented model must be converted to an equivalent "flat" model (represented as a domain — see Chapter 2) before inference can take place.

## 3.1 Classes and class collections

Classes are represented as C objects of type **h_class_t**.

An object-oriented model is comprised of a set of classes, some of which are instantiated within one or more of the remaining classes. The type **h_class_collection_t** is introduced to represent this set of classes. The HUGIN API requires that there be no references to classes outside of a class collection (i.e., the class referred to by a class instance must belong to the same class collection as the class that contains the class instance).

A class collection is edited as a unit: Modifying parts of the interface of a class will cause all of its instances to be modified in the same way.

## 3.2 Creating classes and class collections

A class always belongs to a (unique) class collection. So, before a class can be created, a class collection must be created.

▶ **h_class_collection_t** ***h_new_class_collection*** (**void**)

Create a new empty class collection.

▶ **h_class_t** ***h_cc_new_class*** (**h_class_collection_t** *cc*)

Create a new class. The new class will belong to class collection *cc*.

▶ **h_class_t** ∗***h_cc_get_members*** (**h_class_collection_t** *cc*)

Retrieve the list of classes belonging to class collection *cc*. The list is a NULL-terminated list.

▶ **h_class_collection_t** ***h_class_get_class_collection*** (**h_class_t** *class*)

Retrieve the class collection to which class *class* belongs.

## 3.3 Deleting classes and class collections

▶ **h_status_t** ***h_cc_delete*** (**h_class_collection_t** *cc*)

Delete class collection *cc*. This also deletes all classes belonging to *cc*.

▶ **h_status_t** ***h_class_delete*** (**h_class_t** *class*)

Delete class *class* and remove it from the class collection to which it belongs. If *class* is instantiated, then this operation will fail. (The *h_class_get_instances*(51) function can be used to test whether *class* is instantiated.)

## 3.4 Naming classes

In order to generate textual descriptions of classes and class collections in the form of NET files, it is necessary to name classes.

▶ **h_status_t** ***h_class_set_name*** (**h_class_t** *class*, **h_string_t** *name*)

Set (or change) the name of *class* to *name*. The *name* must be a valid name (i.e., a valid C identifier) and must be distinct from the names of the other classes in the class collection to which *class* belongs.

- **h_string_t** ***h_class_get_name*** (**h_class_t** *class*)

Retrieve the name of *class*. If *class* is unnamed, a new unique name will automatically be generated and assigned to *class*.

- **h_class_t** ***h_cc_get_class_by_name***
  (**h_class_collection_t** *cc*, **h_string_t** *name*)

Retrieve the class with name *name* in class collection *cc*. If no such class exists in *cc*, NULL is returned.

## 3.5 Creating basic nodes

Creating basic (i.e., non-instance) nodes in classes is similar to the way nodes are created in domains (see Section 2.3).

- **h_node_t** ***h_class_new_node***
  (**h_class_t** *class*, **h_node_category_t** *category*,
  **h_node_kind_t** *kind*)

Create a new basic node of the indicated *category* and *kind* within *class*. The node will have default values assigned to its attributes: The desired attributes of the new node should be explicitly set using the relevant API functions.

- **h_class_t** ***h_node_get_home_class*** (**h_node_t** *node*)

Retrieve the class to which *node* belongs. If *node* is NULL, or *node* does not belong to a class (i.e., it belongs to a domain), NULL is returned.

Deletion of basic nodes is done using *h_node_delete*(30).

## 3.6 Naming nodes

Nodes belonging to classes can be named, just like nodes belonging to domains. The functions to handle names of class nodes are the same as those used for domain nodes (see Section 2.7) plus the following function.

- **h_node_t** ***h_class_get_node_by_name*** (**h_class_t** *class*, **h_string_t** *name*)

Retrieve the node with name *name* in *class*. If no node with that name exists in *class*, or if an error occurs (i.e., *class* or *name* is NULL), NULL is returned.

## 3.7 The interface of a class

A class has a set of input nodes and a set of output nodes. These nodes represent the interface of the class and are used to link instances of the class to other class instances and network fragments.

For the following functions, when a node appears as an argument, it must belong to a class. If not, a usage error is generated.

▸ **h_status_t** ***h_node_add_to_inputs*** (**h_node_t** *node*)

Add *node* to the set of input nodes associated with the class to which *node* belongs. The following restrictions must be satisfied: *node* must be a chance or a decision node, *node* must not be an output node, *node* must not be an output clone (see Section 3.8), and *node* must have no parents.

The last condition is also enforced by *h_node_add_parent*(31) — it will not add parents to input nodes.

▸ **h_node_t** ∗***h_class_get_inputs*** (**h_class_t** *class*)

Retrieve the list of input nodes associated with *class*.

▸ **h_status_t** ***h_node_remove_from_inputs*** (**h_node_t** *node*)

Remove *node* from the set of input nodes associated with the class to which *node* belongs. (It is checked that *node* is an input node.) Input bindings (see Section 3.9) involving *node* in the instances of the class to which *node* belongs are deleted.

▸ **h_status_t** ***h_node_add_to_outputs*** (**h_node_t** *node*)

Add *node* to the set of output nodes associated with the class to which *node* belongs. The following restrictions must be satisfied: *node* must be a chance or a decision node, and *node* must not be an input node. Output clones (see Section 3.8) corresponding to *node* are created for all instances of the class to which *node* belongs.

▸ **h_node_t** ∗***h_class_get_outputs*** (**h_class_t** *class*)

Retrieve the list of output nodes associated with *class*.

▸ **h_status_t** ***h_node_remove_from_outputs*** (**h_node_t** *node*)

Remove *node* from the set of output nodes associated with the class to which *node* belongs. (It is checked that *node* is an output node.) All output clones corresponding to *node* are deleted (if any of these output clones are output nodes themselves, their clones are deleted too, recursively).

This function illustrates that modifying one class may affect many other classes. This can happen when a class is modified such that its interface, or some attribute of a node in the interface, is changed. In that case, all instances of the class are affected. It is most efficient to specify the interface of a class before creating instances of it.

Deletion of an interface node (using *h_node_delete*(30)) implies invocation of either *h_node_remove_from_inputs* or *h_node_remove_from_outputs*, depending on whether the node to be deleted is an input or an output node, respectively.

## 3.8 Creating instances of classes

A class can be instantiated within other classes. Each such instance is represented by a so-called *instance node*. Instance nodes are of category *h_category_instance*.

▶ **h_node_t** ***h_class_new_instance*** (**h_class_t** $C_1$, **h_class_t** $C_2$)

Create an instance of class $C_2$. An instance node representing this class instance is added to class $C_1$. The return value is this instance node.

Output clones (see below) corresponding to the output nodes of class $C_2$ are also created and added to class $C_1$.

The classes $C_1$ and $C_2$ must belong to the same class collection. This ensures that dependencies between distinct class collections cannot be created.

Note that instance nodes define a “part-of” hierarchy for classes: classes containing instances of some class C are parents of C. This hierarchy must form an acyclic directed graph. The *h_class_new_instance* function checks this condition. If the condition is violated, or memory is exhausted, the function returns NULL.

The *h_node_delete*(30) function is used to delete instance nodes. Deleting an instance node will also cause all output clones associated with the class instance to be deleted (see below).

▶ **h_class_t** ***h_node_get_instance_class*** (**h_node_t** *instance*)

Retrieve the class of which the instance node *instance* is an instance. (That is, the class passed as the second argument to *h_class_new_instance* when *instance* was created.)

▶ **h_node_t** ****h_class_get_instances*** (**h_class_t** *class*)

Retrieve a NULL-terminated list of all instances of *class* (the list contains an instance node for each instance of *class*).

Note that the instance nodes do *not* belong to *class*.

### Output clones

Whenever a class instance is created, “instances” of all output nodes of the class are also created. These nodes are called *output clones*. Since several instances of some class C can exist in the same class, we need a way to distinguish different copies of some output node Y of class C corresponding to different instances of C — the output clones serve this purpose. For example, when specifying output Y of class instance I as a parent of some node, the output clone corresponding to the $(I, Y)$ combination must be passed to *h_node_add_parent*(31). Output clones are retrieved using the *h_node_get_output*(53) function.

Many API operations are not allowed for output clones. The following restrictions apply:

- Output clones can be used as parents, but cannot have parents themselves.
- Output clones do not have tables or models.
- For discrete output clones, attributes relating to states (i.e., subtype, number of states, state labels, and state values) can be retrieved, but not set. These attributes are identical to those of the “real” output node (known as the *master* node) and change automatically whenever the corresponding attributes of the master are modified. For example, when the number of states of an output node is changed, then all tables in which one or more of its clones appear will automatically be resized as described in Section 2.5.
- An output clone cannot be deleted directly. Instead, it is automatically deleted when its master is deleted or removed from the class interface (see *h_node_remove_from_outputs*(50)), or when the class instance to which it is associated is deleted.

Output clones are created without names, but they can be named just like other nodes.

An output clone belongs to the same class as the instance node with which it is associated. Hence, it appears in the node list of that class (and will be seen when iterating over the nodes of the class).

▸ **h_node_t** ***h_node_get_master*** (**h_node_t** *node*)

Retrieve the output node from which *node* was cloned (*node* must be an output clone).

Note that the master itself can be an output clone (since *h_node_add_to_outputs*(50) permits output clones to be output nodes).

▶ **h_node_t** ***h_node_get_instance*** (**h_node_t** *node*)

Retrieve the instance node associated with the output clone *node*.

▶ **h_node_t** ***h_node_get_output*** (**h_node_t** *instance*, **h_node_t** *output*)

Retrieve the output clone that was created from *output* when *instance* was created. (This implies that *output* is an output node of the class from which *instance* was created, and that *output* is the master of the output clone returned.)

▶ **h_status_t** ***h_node_substitute_class*** (**h_node_t** *instance*, **h_class_t** *new*)

Change the class instance *instance* to be an instance of class *new*. Let *old* be the original class of *instance*. Then the following conditions must hold:

- for each input node in *old*, there must exist an input node in *new* with the same name, category, and kind;
- for each output node in *old*, there must exist a compatible output node in *new* with the same name.

(Note that this implies that interface nodes must be named.) The notion of compatibility referred to in the last condition is the same as that used by *h_node_switch_parent*(32) and for input bindings (see Section 3.9 below).

The input bindings for *instance* are updated to refer to input nodes of class *new* instead of class *old* (using match-by-name).

Similarly, the output clones associated with *instance* are updated to refer to output nodes of class *new* instead of class *old* (again using match-by-name). This affects only the value returned by *h_node_get_master*(52) — in all other respects, the output clones are unaffected.

Extra output clones will be created, if class *new* has more output nodes than class *old*.

## 3.9 Putting the pieces together

In order to make use of class instances, we need to specify inputs to them and use their outputs. Using their outputs is simply a matter of specifying the outputs (or, rather, the corresponding output clones) as parents of the nodes that actually use these outputs. Inputs to class instances are specified using the following function.

▶ **h_status_t** ***h_node_set_input***
(**h_node_t** *instance*, **h_node_t** *input*, **h_node_t** *node*)

This establishes an input binding: *node* is the node to be used as actual input for the formal input node *input* of the class of which *instance* is an instance;

*instance* and *node* must belong to the same class, and *input* and *node* must be of the same category and kind.

The *h_node_set_input* function does not prevent the same node from being bound to two or more input nodes of the same class instance. However, it is an error if a node ends up being parent of some other node "twice" in the runtime domain (Section 3.10). This happens if some node A is bound to two distinct input nodes, $X_1$ and $X_2$, of some class instance I, and $X_1$ and $X_2$ have a common child in the class of which I is an instance. This will cause *h_class_create_domain*(54) to fail.

Note that for a given input binding to make sense, the formal and actual input nodes must be *compatible*. The notion of compatibility used for this purpose is the same as that used by the *h_node_switch_parent*(32) and *h_node_substitute_class*(53) functions. This means that the nodes must be of the same category and kind, and (if the nodes are discrete) have the same subtype, the same number of states, the same list of state labels, and the same list of state values (depending on the subtype). Only the category/kind restriction is checked by *h_node_set_input*. The other restrictions are checked by *h_class_create_domain*(54) (since subtype, state labels, and state values can be changed at any time, whereas category and kind cannot).

▸ **h_node_t** ***h_node_get_input*** (**h_node_t** *instance*, **h_node_t** *input*)

For the class instance represented by the instance node *instance*, retrieve the actual input node bound to the formal input node *input* (which must be an input node of the class of which *instance* is an instance). If an error is detected, or no node is bound to the specified input node, NULL is returned.

▸ **h_status_t** ***h_node_unset_input*** (**h_node_t** *instance*, **h_node_t** *input*)

Delete the input binding (if any) for *input* in class instance *instance* (*input* must be an input node of the class of which *instance* is an instance).

## 3.10 Creating a runtime domain

Before inference can be performed, a class must be expanded to its corresponding flat domain (known as the "runtime" domain).

▸ **h_domain_t** ***h_class_create_domain*** (**h_class_t** *class*)

Create the runtime domain corresponding to *class*. The runtime domain is not compiled — it must be explicitly compiled before it can be used for inference.

Creating a runtime domain is a recursive process: First, domains corresponding to the instance nodes within *class* are constructed (using *h_class_* *create_domain* recursively). These domains are then merged into a common domain, and copies of all non-instance nodes of *class* are added to the domain. Finally, the copies of the formal input nodes of the subdomains are identified with their corresponding actual input nodes, if any.

Note that the runtime domain contains only basic (i.e., non-instance) nodes.

The attributes of the runtime domain are copies of those of *class*.

Models and tables are copied to the runtime domain. In particular, if tables are up-to-date with respect to their models in *class*, then this will also be the case in the runtime domain. This can save a lot of time (especially if many copies of a class are made), since it can be very expensive to generate a table. Generating up-to-date tables is done using *h_class_generate_tables*(84).

In order to associate a node of the runtime domain with (the path of) the node of the object-oriented model from which it was created, a list of nodes (called the *source* list) is provided. This node list traces a path from the root of the object-oriented model to a leaf of the model. Assume the source list corresponding to a runtime node is $\langle N_1, \ldots, N_m \rangle$. All nodes except the last must be instance nodes: $N_1$ must be a node within *class*, and $N_i$ $(i > 1)$ must be a node within the class of which $N_{i-1}$ is an instance.

The nodes of the runtime domain are assigned names based on the source lists: If the name of node $N_i$ is $n_i$, then the name of the runtime node is the "dotted name" $n_1.n_2.\cdots.n_m$. Because the names of the source nodes are not allowed to contain "dots," this scheme will generate unique (and "meaningful") names for all runtime nodes. (As a side-effect of this operation, the source nodes are also assigned names — if they are not already named.)

▸ **h_node_t** ∗***h_node_get_source*** (**h_node_t** *node*)

Return the source list for *node*; *node* must belong to a runtime domain created by *h_class_create_domain* from an object-oriented model. Each node in the source list belongs to some class of this model.

Note that the contents of the source list will in general be invalidated when some class of the object-oriented model is modified.

**Example 3.1** Consider the object-oriented model shown in Figure 3.1. It has three basic nodes, A, B, and C, and two instance nodes, $I_1$ and $I_2$, which are instances of the same class. This class has two input nodes, X and Y, and one output node, Z. Input X of class instance $I_1$ is bound to A. Input Y of class instance $I_1$ and input X of class instance $I_2$ are both bound to B. Input Y of class instance $I_2$ is unbound.

The runtime domain corresponding to this object-oriented model is shown in Figure 3.2. Note that bound input nodes do not appear in the runtime domain: The children of a bound input node instead become children of the node to which the input node is bound. Unbound input nodes, on the other hand, do appear in the runtime domain.

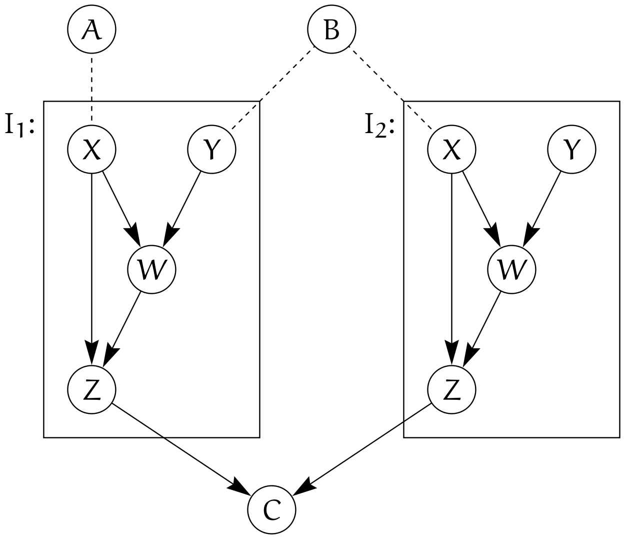

Figure 3.1: An object-oriented belief network model.

The node lists returned by *h_node_get_source*(55) for each node of the runtime domain are as follows: $A_0$: $\langle A \rangle$, $B_0$: $\langle B \rangle$, $C_0$: $\langle C \rangle$, $W_1$: $\langle I_1, W \rangle$, $Z_1$: $\langle I_1, Z \rangle$, $Y_2$: $\langle I_2, Y \rangle$, $W_2$: $\langle I_2, W \rangle$, $Z_2$: $\langle I_2, Z \rangle$. ∎

## 3.11 Node iterator

In order to iterate over the nodes of a class, the following function is needed.

▸ **h_node_t *h_class_get_first_node*** (**h_class_t** *class*)

Return a pointer to the first node of *class*, or NULL if *class* contains no nodes, or *class* is NULL (this is considered an error).

This function should be used in conjunction with the *h_node_get_next*(40) function.

## 3.12 User data

Section 2.9 describes functions for associating user-defined data with domains and nodes. Similar functions are also provided for classes.

The first two functions manage generic pointers to data structures that must be maintained by the user application.

▸ **h_status_t *h_class_set_user_data*** (**h_class_t** *class*, **void** ∗*data*)

Store the pointer *data* within *class*.

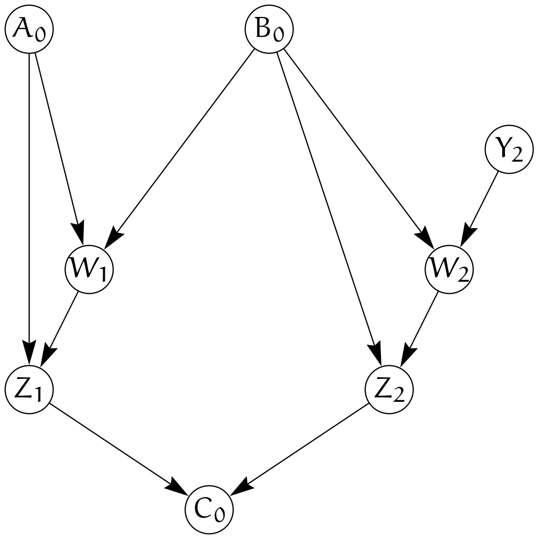

Figure 3.2: A runtime domain corresponding to the object-oriented model shown in Figure 3.1.

▸ **void** ∗***h_class_get_user_data*** (**h_class_t** *class*)

Retrieve the value stored within the user data slot of *class*. If no value has been stored, the stored value is NULL, or *class* is NULL (this is an error), NULL is returned.

The following functions manage key/value-type attributes.

▸ **h_status_t** ***h_class_set_attribute***
(**h_class_t** *class*, **h_string_t** *key*, **h_string_t** *value*)

Insert (or update, if *key* is already defined) the *key*/*value*-pair in the attribute list for *class* (*key* must be a valid C language identifier). If *value* is NULL, the attribute is removed.

▸ **h_string_t** ***h_class_get_attribute*** (**h_class_t** *class*, **h_string_t** *key*)

Lookup the value associated with *key* in the attribute list for *class*. If *key* is not present, or if an error occurs, NULL is returned.

This function is needed for iterating over the attributes of a class.

▸ **h_attribute_t** ***h_class_get_first_attribute*** (**h_class_t** *class*)

Retrieve the first attribute object for *class*. If the attribute list is empty (or *class* is NULL), NULL is returned.

The remaining functions needed for iteration over attributes are described in Section 2.9.

# Chapter 4

# Tables

Tables are used within HUGIN for representing the conditional probability, policy, and utility potentials of nodes, the probability and utility potentials on separators and cliques of junction trees, evidence potentials, etc.

The HUGIN API makes (some of) these tables accessible to the programmer via the opaque pointer type **h_table_t** and associated functions.

The HUGIN API currently does not provide means for the programmer to construct his own table objects, just the functions to manipulate the tables created by HUGIN.

## 4.1 What is a table?

A *potential* is a function from the state space of a set of variables into the set of real numbers. A *table* is a computer representation of a potential.

The HUGIN API introduces the opaque pointer type **h_table_t** to represent table objects.

Consider a potential defined over a set of nodes. In general, the state space of the potential has both a *discrete part* and a *continuous part*. Both parts are indexed by the set $\mathcal{I}$ of configurations of states of the discrete nodes. The discrete data are comprised of numbers $x(i)$ ($i \in \mathcal{I}$). If the potential is a probability potential, $x(i)$ is a probability (i.e., a number between $0$ and $1$, inclusive). If the potential is a utility potential, $x(i)$ can be any real number.

Probability potentials with continuous nodes represent so-called CG potentials (see [7, 16, 19]). They can either represent conditional or marginal probability distributions. CG potentials of the conditional type are accessed using special functions — see Section 2.6. For CG potentials of the marginal type, we have, for each $i \in \mathcal{I}$, a number $x(i)$ (a probability), a mean value vector $\mu(i)$, and a (symmetric) covariance matrix $\Sigma(i)$; $\mu(i)$ and $\Sigma(i)$ are the mean value vector and the covariance matrix for the conditional distribution of the continuous nodes given the configuration $i$ of the discrete nodes.

To be able to use a table object effectively, it is necessary to know some facts about the representation of the table.

The set of configurations of the discrete nodes (i.e., the discrete state space $\mathcal{I}$) is organized as a multi-dimensional array in *row-major* format. Each dimension of this array corresponds to a discrete node, and the ordered list of dimensions defines the format as follows.[1]

Suppose that the list of discrete nodes is $\langle N_1, \ldots, N_n \rangle$, and suppose that node $N_k$ has $s_k$ states. A configuration of the states of these nodes is a list $\langle i_1, \ldots, i_n \rangle$, with $0 \leq i_k < s_k$ $(1 \leq k \leq n)$.

The set of configurations is mapped into the index set $\{0, \ldots, S - 1\}$ where

$$S = \prod_{k=1}^{n} s_k$$

(This quantity is also known as the *size* of the table.)

A specific configuration $\langle i_1, \ldots, i_n \rangle$ is mapped to the index value

$$\sum_{k=1}^{n} a_k i_k$$

where

$$a_k = s_{k+1} \cdots s_n$$

(Note that this mapping is one-to-one.)

**Example 4.1** The mapping from state configurations to table indexes can be expressed using a simple loop. Let *node_count* be the number of discrete nodes in the given table, let *state_count[k]* be the number of states of the $k$th node , and let *configuration[k]* be the state of the $k$th node in the state configuration. Then the table index corresponding to the given state configuration can be computed as follows:

```
size_t k, index = 0;

for (k = 0; k < node_count; k++)
    index = index * state_count[k] + configuration[k];
```

An API function is also provided to perform this computation: *h_table_get_index_from_configuration*$^{(61)}$. ∎

Many HUGIN API functions use the index of a configuration whenever the states of a list of discrete nodes are needed. Examples of such functions are: *h_node_set_alpha*$^{(38)}$, *h_node_get_alpha*$^{(39)}$, *h_table_get_mean*$^{(63)}$, etc.

---

[1] This only applies to *uncompressed* tables. If a table is compressed, then there is no simple way to map a configuration to a table index. (Compression is described in Section 6.6.)

As a convenience, the HUGIN API provides functions to convert between table indexes and state configurations.

▸ **size_t** ***h_table_get_index_from_configuration***
(**h_table_t** *table*, **const size_t** $*$*configuration*)

Compute the table index corresponding to the state configuration specified in the *configuration* argument. This must be a list containing state indexes for the (discrete) nodes of *table* — the kth element of *configuration* must be a state index for the kth node in the node list of *table*. A state index for each discrete node of *table* must be specified (so the length of the *configuration* list must be at least the number of discrete nodes of *table*).

If an error occurs, "(**size_t**) $-1$" is returned.

▸ **h_status_t** ***h_table_get_configuration_from_index***
(**h_table_t** *table*, **size_t** $*$*configuration*, **size_t** *index*)

Compute the state configuration corresponding to the specified table *index*. The configuration is stored in the *configuration* list (this list must be allocated by the user application). The function is the inverse of the above function, so the kth element of *configuration* is the state index corresponding to the kth node in the node list of *table*. The length of the *configuration* list must be at least the number of discrete nodes of *table*.

**Example 4.2** Given three discrete nodes, $A$ with 2 states ($a_0$ and $a_1$), $B$ with 3 states ($b_0$, $b_1$, and $b_2$), and $C$ with 4 states ($c_0$, $c_1$, $c_2$, and $c_3$), here is a complete list of all configurations of $\langle A, B, C\rangle$ and their associated indexes:

$\langle a_0, b_0, c_0\rangle$, index 0;
$\langle a_0, b_0, c_1\rangle$, index 1;
$\langle a_0, b_0, c_2\rangle$, index 2;
$\langle a_0, b_0, c_3\rangle$, index 3;
$\langle a_0, b_1, c_0\rangle$, index 4;
$\langle a_0, b_1, c_1\rangle$, index 5;
$\langle a_0, b_1, c_2\rangle$, index 6;
$\langle a_0, b_1, c_3\rangle$, index 7;
$\langle a_0, b_2, c_0\rangle$, index 8;
$\langle a_0, b_2, c_1\rangle$, index 9;
$\langle a_0, b_2, c_2\rangle$, index 10;
$\langle a_0, b_2, c_3\rangle$, index 11;
$\langle a_1, b_0, c_0\rangle$, index 12;
$\langle a_1, b_0, c_1\rangle$, index 13;
$\langle a_1, b_0, c_2\rangle$, index 14;
$\langle a_1, b_0, c_3\rangle$, index 15;
$\langle a_1, b_1, c_0\rangle$, index 16;

$\langle a_1, b_1, c_1 \rangle$, index 17;
$\langle a_1, b_1, c_2 \rangle$, index 18;
$\langle a_1, b_1, c_3 \rangle$, index 19;
$\langle a_1, b_2, c_0 \rangle$, index 20;
$\langle a_1, b_2, c_1 \rangle$, index 21;
$\langle a_1, b_2, c_2 \rangle$, index 22;
$\langle a_1, b_2, c_3 \rangle$, index 23.

∎

## 4.2 The nodes and the contents of a table

▸ **h_node_t** ∗***h_table_get_nodes*** (**h_table_t** *table*)

Retrieve the NULL-terminated list of nodes associated with *table*. If an error occurs, NULL is returned.

The first part of this list is comprised of the discrete nodes of the potential represented by *table*. The order of these nodes determines the layout of the discrete state configurations as described in the previous section. The second part of the list is comprised of the continuous nodes of the potential represented by *table*.

The pointer returned by *h_table_get_nodes* is a pointer to the list stored in the table structure. Do not modify or deallocate it.

▸ **h_number_t** ∗***h_table_get_data*** (**h_table_t** *table*)

Retrieve a pointer to the array of *table* holding the actual discrete data (denoted by $x(i)$ in Section 4.1). This array is a one-dimensional (row-major) representation of the multi-dimensional array.

Since the pointer returned is a pointer to the actual array stored within the table structure, it is possible to modify the contents of the table through this pointer. But, of course, the pointer must not be deallocated.

Note that pointers to nodes and data arrays within tables may be invalidated by other API functions. For example, suppose the application holds a pointer to the data array of the conditional probability table (CPT) of some node $X$. Changing the set of parents of $X$ (or the number of states of $X$ or one of its parents) causes the CPT to be "resized." Most likely, this will cause the data array of the CPT to be moved to a different memory location, and the old pointer is no longer valid.

For tables with continuous nodes, *h_table_get_data* is used to access the $x(i)$ component. To access the $\mu(i)$ and $\Sigma(i)$ components, the following functions must be used (we assume that *table* is a table returned by *h_domain_get_marginal*(109) or *h_node_get_distribution*(110)):

- **h_double_t** ***h_table_get_mean*** (**h_table_t** *table*, **size_t** *i*, **h_node_t** *node*)

Return the mean value of the conditional distribution of the continuous node *node* given the discrete state configuration *i*.

- **h_double_t** ***h_table_get_covariance***
  (**h_table_t** *table*, **size_t** *i*, **h_node_t** $node_1$, **h_node_t** $node_2$)

Return the covariance of the conditional distribution of the continuous nodes $node_1$ and $node_2$ given the discrete state configuration *i*.

- **h_double_t** ***h_table_get_variance***
  (**h_table_t** *table*, **size_t** *i*, **h_node_t** *node*)

Return the variance of the conditional distribution of the continuous node *node* given the discrete state configuration *i*.

## 4.3 Deleting tables

The HUGIN API also provides a function to release the storage resources used by a table. The *h_table_delete* function can be used to deallocate tables returned by *h_domain_get_marginal*(109), *h_node_get_distribution*(110), *h_node_get_experience_table*(142), and *h_node_get_fading_table*(143). All other deletion requests are ignored (e.g., a table returned by *h_node_get_table*(36) cannot be deleted).

- **h_status_t** ***h_table_delete*** (**h_table_t** *table*)

Release the memory resources used by *table*.

## 4.4 The size of a table

The number of state configurations of discrete nodes represented in a table is referred to as the *size* of the table. If the table has $n$ discrete nodes, and the kth node has $s_k$ states, then the size of the table is

$$\prod_{k=1}^{n} s_k$$

This assumes that all state configurations are represented in the table. If some state configurations have been removed (by a process known as *compression* — see Section 6.6), the size will be smaller.

- **size_t** ***h_table_get_size*** (**h_table_t** *table*)

Return the size of *table*. If an error occurs, “(**size_t**) $-1$” is returned.

Associated with each state configuration of discrete nodes in the table is a real number of type **h_number_t**. In a single-precision version of the HUGIN API, **h_number_t** is a 4-byte quantity, but in a double-precision version, it is an 8-byte quantity.

If the table has continuous nodes, then there will be additional data stored in the table (such as mean and covariance values). If the table has $m$ continuous nodes, the number of additional data elements is

$$\frac{m(m+3)}{2} \prod_{k=1}^{n} s_k$$

We call this quantity the *CG size* of the table. (If the table is compressed, then this quantity is reduced proportional to the number of discrete configurations removed from the table.)

▶ **size_t** ***h_table_get_cg_size*** (**h_table_t** *table*)

Return the CG size of *table*. If an error occurs, "(**size_t**) $-1$" is returned. Each CG data element occupies 8 bytes.

## 4.5 Rearranging the contents of a table

Sometimes it is convenient to enforce a specific layout of the contents of a table. This can be done by permuting the node list of the table.

▶ **h_status_t** ***h_table_reorder_nodes*** (**h_table_t** *table*, **h_node_t** **order*)

Reorder the node list of *table* to be *order* (the contents of the data arrays are reorganized according to the new node order); *order* must be a NULL-terminated list containing a permutation of the node list of *table*. If *table* is the node table of some (chance or decision) node, then that node must have the same position in *order* as in the node list of *table*. Also, if the node list of *table* contains both discrete and continuous nodes, the discrete nodes must precede the continuous nodes in *order*.

If a chance node has experience or fading tables, then the order of the (discrete) parents in these tables must be the same as in the conditional probability table. This constraint is enforced by *h_table_reorder_nodes*. So, reordering one of these tables also reorders the other two tables (if they exist).

In the current implementation of the HUGIN API, reordering the nodes of a node table causes the affected domain to be uncompiled. (Except, if the node list of *table* is equal to *order*, then nothing is done.)

**Example 4.3** The following code creates four chance nodes, two discrete ($a$ and $b$) and two continuous ($x$ and $y$); $a$, $b$, and $x$ are made parents of $y$. Then the number of states of $a$ and $b$ and the conditional distributions of the nodes must be specified (this is not shown).

```
h_domain_t d = h_new_domain ();
h_node_t a = h_domain_new_node (d, h_category_chance,
                                   h_kind_discrete);
h_node_t b = h_domain_new_node (d, h_category_chance,
                                   h_kind_discrete);
h_node_t x = h_domain_new_node (d, h_category_chance,
                                   h_kind_continuous);
h_node_t y = h_domain_new_node (d, h_category_chance,
                                   h_kind_continuous);

h_node_add_parent (y, a);
h_node_add_parent (y, b);
h_node_add_parent (y, x);

...  /* set number of states,
        specify conditional distributions, etc. */
```

Now suppose we want to ensure that $a$ appears before $b$ in the node list of the conditional probability table of $y$. We make a list containing the desired order of $y$ and its parents, and then we call *h_table_reorder_nodes*.

```
h_node_t list[5];
h_table_t t = h_node_get_table (y);

list[0] = a; list[1] = b;
list[2] = y; list[3] = x;
list[4] = NULL;

h_table_reorder_nodes (t, list);
```

Note that since $y$ (the “child” node of the table) is continuous, it must be the first node among the continuous nodes in the node list of the table. Had $y$ been discrete, it should have been the last node in the node list of the table (in this case, all nodes would be discrete). ∎

# Chapter 5

# Generating Tables

This chapter describes how to specify a compact description of a node table. From this description, the contents of the table is generated.

Such a table description is called a *model*. A model consists of a list of discrete nodes and a set of *expressions* (one expression for each configuration of states of the nodes). The expressions are built using standard statistical distributions (such as Normal, Binomial, Beta, Gamma, etc.), arithmetic operators (such as addition, subtraction, etc.), standard functions (such as logarithms, exponential, trigonometric, and hyperbolic functions), logical operators (conjunction, disjunction, and conditional), and relations (such as less-than or equals).

Models are also used to specify the functions associated with function nodes. In this case, no tables are generated.

## 5.1 Subtyping of discrete nodes

In order to provide a rich language for specifying models, we introduce a classification of the discrete chance and decision nodes into four groups:

- *Labeled* nodes. These are discrete nodes that have a label associated with each state (and nothing else). Labels can be used in equality comparisons and to express deterministic relationships.
- *Boolean* nodes. Such nodes represent the truth values, ‘false’ and ‘true’ (in that order).
- *Numbered* nodes. The states of such nodes represent numbers (not necessarily integers).
- *Interval* nodes. The states of such nodes represent (disjoint) intervals on the real line.

The last two groups are collectively referred to as *numeric* nodes.

Recall that discrete nodes have a finite number of states. This implies that numbered nodes can only represent a finite subset of, e.g., the nonnegative integers (so special conventions are needed for discrete infinite distributions such as the Poisson — see Section 5.7.2).

The above classification of discrete nodes is represented by the enumeration type **h_node_subtype_t**. The constants of this enumeration type are: *h_subtype_label*, *h_subtype_boolean*, *h_subtype_number*, and *h_subtype_interval*. In addition, the constant *h_subtype_error* is defined for denoting errors.

▸ **h_status_t *h_node_set_subtype***
(**h_node_t** *node*, **h_node_subtype_t** *subtype*)

Set the subtype of *node* (a discrete chance or decision node) to *subtype*.

If *subtype* is *h_subtype_boolean* then *node* must have exactly two states. Moreover, when a node has subtype 'boolean', *h_node_set_number_of_states*(35) cannot change the state count of the node.

The default subtype (i.e., if it is not set explicitly using the above function) of a node is *h_subtype_label*.

The state labels and the state values (see *h_node_set_state_label*(76) and *h_node_set_state_value*(77)) are not affected by this function.

▸ **h_node_subtype_t *h_node_get_subtype*** (**h_node_t** *node*)

Return the subtype of *node* (which must be a discrete chance or decision node). If an error occurs, *h_subtype_error* is returned.

## 5.2 Expressions

Expressions are classified (typed) by what they denote. We have four different types: *labeled*, *boolean*, *numeric*, and *distribution*.[1]

We define an opaque pointer type **h_expression_t** to represent the expressions that constitute a model. Expressions can represent constants, variables, and composite expressions (i.e., expressions comprised of an operator applied to a list of arguments). The HUGIN API defines the following set of functions to construct expressions.

All these functions return NULL on error (e.g., out-of-memory).

▸ **h_expression_t *h_node_make_expression*** (**h_node_t** *node*)

This function constructs an expression that represents a variable. If *node* is a CG, a utility, or a function node, then the constructed expression is of numeric type. If *node* is a discrete node, then the type of the expression is either labeled, boolean, or numeric, depending on the subtype of *node*.

---

[1] In a conditional expression, it is possible to combine expressions of different types — resulting in a "union" type. See *h_operator_if*(71).

▸ **h_expression_t** ***h_label_make_expression*** (**h_string_t** *label*)

Construct an expression that represents a label constant. A copy is made of *label*.

▸ **h_expression_t** ***h_boolean_make_expression*** (**h_boolean_t** *b*)

Construct an expression that represents a boolean constant: 'true' if *b* is 1, and 'false' if *b* is 0.

▸ **h_expression_t** ***h_number_make_expression*** (**h_double_t** *number*)

Construct an expression representing the numeric constant *number*.

The expressions constructed using one of the above four functions are called *simple* expressions, whereas the following function constructs *composite* expressions.

▸ **h_expression_t** ***h_make_composite_expression***
(**h_operator_t** *operator*, **h_expression_t** ∗*arguments*)

This function constructs a composite expression representing *operator* applied to *arguments*; *arguments* must be a NULL-terminated list of expressions. The function allocates an internal array to hold the expressions, but it does not copy the expressions themselves.

The **h_operator_t** type referred to above is an enumeration type listing all possible operators, including statistical distributions.

The complete list is as follows:

- *h_operator_add*, *h_operator_subtract*, *h_operator_multiply*, *h_operator_divide*, *h_operator_power*

  These are binary operators that can be applied to numeric expressions.

- *h_operator_negate*

  A unary operator for negating a numeric expression.

- *h_operator_equals*, *h_operator_less_than*, *h_operator_greater_than*, *h_operator_not_equals*, *h_operator_less_than_or_equals*, *h_operator_greater_than_or_equals*

  These are binary comparison operators for comparing labels, numbers, and boolean values (both operands must be of the same type). Only the equality operators (i.e., *h_operator_equals* and *h_operator_not_equals*) can be applied to labels and boolean values.

- *h_operator_Normal*, *h_operator_LogNormal*, *h_operator_Beta*, *h_operator_Gamma*, *h_operator_Exponential*, *h_operator_Weibull*, *h_operator_Uniform*, *h_operator_Triangular*, *h_operator_PERT*

  Continuous statistical distributions — see Section 5.7.1.

- *h_operator_Binomial*, *h_operator_Poisson*, *h_operator_NegativeBinomial*, *h_operator_Geometric*, *h_operator_Distribution*, *h_operator_NoisyOR*

  Discrete statistical distributions — see Section 5.7.2.

- *h_operator_truncate*

  This operator can be applied to a continuous statistical distribution in order to form a truncated distribution. The operator takes either two or three arguments. When three arguments are specified, the first and third arguments must be numeric expressions denoting, respectively, the left and right truncation points, while the second argument must denote the distribution to be truncated.

  Either the first or the third argument can be omitted. Omitting the first argument results in a right-truncated distribution, and omitting the third argument results in a left-truncated distribution.

  **Example 5.1** Using the syntax described in Section 5.3, a truncated normal distribution can be expressed as follows:

  ```
  truncate (-4, Normal (0, 1), 4)
  ```

  This distribution is truncated at the left at $-4$ and at the right at $4$. A left-truncated (at $-4$) normal distribution is obtained by omitting the last argument:

  ```
  truncate (-4, Normal (0, 1))
  ```

  Similarly, a right-truncated normal distribution can be obtained by omitting the first argument. ∎

- *h_operator_min*, *h_operator_max*

  Compute the minimum or maximum of a list of numbers (the list must be non-empty).

- *h_operator_log*, *h_operator_log2*, *h_operator_log10*, *h_operator_exp*, *h_operator_sin*, *h_operator_cos*, *h_operator_tan*, *h_operator_sinh*, *h_operator_cosh*, *h_operator_tanh*, *h_operator_sqrt*, *h_operator_abs*

  Standard mathematical functions to compute the natural (i.e., base $e$) logarithm, base $2$ and base $10$ logarithms, exponential, trigonometric functions, hyperbolic functions, square root, and absolute value of a number.

- *h_operator_floor*, *h_operator_ceil*

  The “floor” and “ceiling” functions round real numbers to integers.

  *floor*$(x)$ (usually denoted $\lfloor x \rfloor$) is defined as the greatest integer less than or equal to $x$.

  *ceil*$(x)$ (usually denoted $\lceil x \rceil$) is defined as the least integer greater than or equal to $x$.

- *h_operator_mod*

  The "modulo" function gives the remainder of a division. It is defined as follows:

  $$mod(x, y) \equiv x - y\lfloor x/y \rfloor, \qquad y \neq 0.$$

  Note that $x$ and $y$ can be arbitrary real numbers (except that $y$ must be nonzero).

- *h_operator_if*

  Conditional expression (with three arguments): The first argument must denote a boolean value, and the second and third arguments must denote values of "compatible" types. The simple case is for these types to be identical: Then the result of the conditional expression is of that common type.

  Else, the type of the conditional expression is a "union" type. Let $T$ be any scalar type (i.e., either *labeled*, *boolean*, or *numeric*). The result of the conditional expression is of type $T$-*or-distribution* if the types of the last two arguments are any two distinct members of the list: $T$, *distribution*, and $T$-*or-distribution*.

  The union-type feature can be used to define relationships that are sometimes probabilistic and sometimes deterministic.

  **Example 5.2** Let $A$ be an interval node, and let $B$ be a boolean node and a parent of $A$. Assume that the following expression is used to define the conditional probability table of $A$.

  ```
  if (B, Normal (0,1), 0)
  ```

  If $B$ is true, then $A$ has a normal distribution. If $B$ is false, then $A$ is instantiated to the interval containing $0$. ∎

- *h_operator_and*, *h_operator_or*, *h_operator_not*

  Standard logical operators: 'not' takes exactly one boolean argument, while 'and' and 'or' take a list (possibly empty) of boolean arguments. Evaluation of an 'and' composite expression is done sequentially, and evaluation terminates when an argument that evaluates to 'false' is found. Similar for an 'or' expression (except that the evaluation terminates when an argument evaluating to 'true' is found).

In addition, a number of 'operators' are introduced to denote simple expressions and errors (for use by *h_expression_get_operator*⁽⁷²⁾):

- *h_operator_label* for expressions constructed using *h_label_make_expression*;

- *h_operator_number* for expressions constructed using *h_number_make_expression*;

- *h_operator_boolean* for expressions constructed using *h_boolean_make_expression*;
- *h_operator_node* for expressions constructed using *h_node_make_expression*;
- *h_operator_error* for illegal arguments to *h_expression_get_operator*.

Analogous to the constructor functions, we also need functions to test how a particular expression was constructed and to access the parts of an expression.

▸ **h_boolean_t** ***h_expression_is_composite*** (**h_expression_t** *e*)

Test whether the expression *e* was constructed using *h_make_composite_expression*.

▸ **h_operator_t** ***h_expression_get_operator*** (**h_expression_t** *e*)

Return the operator that was used when the expression *e* was constructed using *h_make_composite_expression*, or, if *e* is a simple expression, one of the special operators (see the above list).

▸ **h_expression_t** ∗***h_expression_get_operands*** (**h_expression_t** *e*)

Return the operand list that was used when the expression *e* was constructed using *h_make_composite_expression*. Note that the returned list is the real list stored inside *e*, so don't try to deallocate it after use.

▸ **h_node_t** ***h_expression_get_node*** (**h_expression_t** *e*)

Return the node that was used when the expression *e* was constructed using *h_node_make_expression*.

▸ **h_double_t** ***h_expression_get_number*** (**h_expression_t** *e*)

Return the number that was used when the expression *e* was constructed using *h_number_make_expression*. If an error occurs (e.g., *e* was not constructed using *h_number_make_expression*), a negative number is returned. However, since negative numbers are perfectly valid in this context, errors must be checked for using *h_error_code*(18) and friends.

▸ **h_string_t** ***h_expression_get_label*** (**h_expression_t** *e*)

Return the label that was used when the expression *e* was constructed using *h_label_make_expression*. Note that the real label inside *e* is returned, so don't try to deallocate it after use.

▸ **h_boolean_t** ***h_expression_get_boolean*** (**h_expression_t** *e*)

Return the boolean value that was used when the expression *e* was constructed using *h_boolean_make_expression*.

▸ **h_status_t** ***h_expression_delete*** (**h_expression_t** *e*)

Deallocate the expression *e*, including subexpressions (in case of composite expressions).

This function will also be called automatically in a number of circumstances: when a new expression is stored in a model (see *h_model_set_expression*(76)), when parents are removed, and when the number of states of a node is changed.

Note: The HUGIN API keeps track of the expressions stored in models. This means that if you delete an expression with a subexpression that is shared with some expression within some model, then that particular subexpression will not be deleted.

However, if you build two expressions with a shared subexpression (and that subexpression is not also part of some expression owned by HUGIN), then the shared subexpression will not be "protected" against deletion if you delete one of the expressions. For such uses, the following function can be used to explicitly create a copy of an expression.

▸ **h_expression_t** ***h_expression_clone*** (**h_expression_t** *e*)

Create a copy of *e*.

## 5.3 Syntax of expressions

In many situations, it is convenient to have a concrete syntax for presenting expressions (e.g., in the HUGIN GUI application). The syntax is also used in specifications written in the NET language (see Chapter 12).

⟨Expression⟩ → ⟨Simple expression⟩ ⟨Comparison⟩ ⟨Simple expression⟩
| ⟨Simple expression⟩

⟨Simple expression⟩
→ ⟨Simple expression⟩ ⟨Plus or minus⟩ ⟨Term⟩
| ⟨Plus or minus⟩ ⟨Term⟩
| ⟨Term⟩

⟨Term⟩ → ⟨Term⟩ ⟨Times or divide⟩ ⟨Exp factor⟩
| ⟨Exp factor⟩

⟨Exp factor⟩ → ⟨Factor⟩ ^ ⟨Exp factor⟩
| ⟨Factor⟩

⟨Factor⟩ → ⟨Unsigned number⟩
| ⟨Node name⟩
| ⟨String⟩

```
                     | false
                     | true
                     | ( ⟨Expression⟩ )
                     | ⟨Operator name⟩ ( ⟨Expression sequence⟩ )

⟨Expression sequence⟩
                     → ⟨Empty⟩
                     | ⟨Expression⟩ [ , ⟨Expression⟩ ]*

⟨Comparison⟩[2]      → == | = | != | <> | < | <= | > | >=

⟨Plus or minus⟩      → + | -

⟨Times or divide⟩    → * | /

⟨Operator name⟩      → truncate | Normal | LogNormal
                     | Beta | Gamma | Exponential | Weibull
                     | Uniform | Triangular | PERT
                     | Binomial | Poisson | NegativeBinomial
                     | Geometric | Distribution | NoisyOR
                     | min | max | log | log2 | log10 | exp
                     | sin | cos | tan | sinh | cosh | tanh
                     | sqrt | abs | floor | ceil | mod
                     | if | and | or | not
```

The operator names refer to the operators of the **h_operator_t**(69) type: prefix the operator name with *h_operator_* to get the corresponding constant of the **h_operator_t** type.

▸ **h_expression_t** ***h_string_parse_expression***
(**h_string_t** *s*, **h_model_t** *model*,
**void** (∗*error_handler*) (**h_location_t**, **h_string_t**, **void** ∗),
**void** ∗*data*)

Parse the expression in string *s* and construct an equivalent **h_expression_t** structure. Node names appearing in the expression must correspond to parents of the owner of *model*. If an error is detected, the *error_handler* function is called with the location (the character index) within *s* of the error, a string that describes the error, and *data*. The storage used to hold the error message string is reclaimed by *h_string_parse_expression*, when *error_handler* returns (so if the error message will be needed later, a copy must be made).

The user-specified *data* allows the error handler to access non-local data (and hence preserve state between calls) without having to use global variables.

The **h_location_t** type is an unsigned integer type (such as **unsigned long**).

---

[2] Note that both C and Pascal notations for equality/inequality operators are accepted.

If no error reports are desired (in this case, only the error indicator returned by *h_error_code*(18) will be available), then the *error_handler* argument may be NULL.

Note: The *error_handler* function may also be called in non-fatal situations (e.g., warnings).

▸ **h_string_t** ***h_expression_to_string*** (**h_expression_t** *e*)

Allocate a string and write into this string a representation of the expression *e* using the above described syntax.

Note that it is the responsibility of the user of the HUGIN API to deallocate the returned string when it is no longer needed.

## 5.4 Creating and maintaining models

The HUGIN API introduces the opaque pointer type **h_model_t** to represent models. Models must be explicitly created before they can be used.

▸ **h_model_t** ***h_node_new_model***
(**h_node_t** *node*, **h_node_t** ∗*model_nodes*)

Create and return a model for *node* (which must be a discrete, a utility, or a function node) using *model_nodes* (a NULL-terminated list of discrete nodes, comprising a subset of the parents of *node*) to define the configurations of the model. If *node* already has a model, it will be deleted before the new model is installed.

If a (non-function) node has a model, the contents of the node table will be generated from the model. This happens automatically as part of compilation, propagation, and reset-inference-engine operations, but it can also be explicitly done by the user (see *h_node_generate_table*(83)). It is possible to modify the contents generated from the model, but note that the inference engine will regenerate the table when certain parameters are changed (see Section 5.8 for precise details).

▸ **h_model_t** ***h_node_get_model*** (**h_node_t** *node*)

Return the model associated with *node*. If no model exists, NULL is returned.

▸ **h_status_t** ***h_model_delete*** (**h_model_t** *model*)

Delete *model* (i.e., deallocate the storage used by *model*). If *model* was a model for a node table, the table will again be the definitive source (i.e., the contents of the table will no longer be derived from a model).

▸ **h_node_t** ∗***h_model_get_nodes*** (**h_model_t** *model*)

Return the list of nodes of *model*.

▸ **size_t** ***h_model_get_size*** (**h_model_t** *model*)

Return the number of configurations of the nodes of *model*. If an error occurs, (**size_t**) −1 (i.e., the number '−1' cast to the type **size_t**) is returned.

▸ **h_status_t** ***h_model_set_expression***
(**h_model_t** *model*, **size_t** *index*, **h_expression_t** *e*)

Store the expression *e* at position *index* in *model*. It is an error if *model* is NULL, or *index* is out of range. If there is already an expression stored at the indicated position, then that expression is deleted — using *h_expression_delete*$^{(73)}$.

Now, let *node* be the "owner" of *model* (i.e., *model* is associated with *node*). If *node* is a utility or a function node, then the type of *e* must be numeric. Otherwise, *node* is a discrete node:

- If *e* is not a composite expression with the operator being one of the statistical distribution operators, the value of *node* is a deterministic function of the parents (for the configurations determined by *index*). In this case, the type of *e* must match the subtype of *node*.

- Otherwise (i.e., the type of *e* is 'distribution'), the statistical distribution denoted by *e* must be appropriate for the subtype of *node* — see Section 5.7.

In all cases, the subexpressions of *e* must not use *node* as a variable — only parents can be used as variables.

▸ **h_expression_t** ***h_model_get_expression***
(**h_model_t** *model*, **size_t** *index*)

Return the expression stored at position *index* within *model*. If *model* is NULL or no expression has been stored at the indicated position, NULL is returned (the first case is an error).

## 5.5 State labels

Labels assigned to states of discrete chance and decision nodes serve two purposes: (1) to identify states of labeled nodes in the table generator, and (2) to identify states in the HUGIN GUI application (for example, when beliefs or expected utilities are displayed).

▸ **h_status_t** ***h_node_set_state_label***
(**h_node_t** *node*, **size_t** *s*, **h_string_t** *label*)

Create a copy of *label* and assign it as the label of state *s* of *node* (which must be a discrete chance or decision node). The *label* can be any string (i.e., it is not restricted in the way that, e.g., node names are).

When *node* is used as a labeled node with the table generator facility, the states must be assigned unique labels.

▸ **h_string_t** ***h_node_get_state_label*** **(h_node_t** *node*, **size_t** *s*)

Return the label of state *s* of *node*. If no label has been assigned to the state, a default label is returned. The default label is the empty string, unless *node* is a boolean node in which case the default labels are `"false"` (for state 0) and `"true"` (for state 1).

If an error occurs (i.e., *node* is not a discrete chance/decision node, or *s* is an invalid state), NULL is returned.

Note that the string returned by *h_node_get_state_label* is not a copy. Thus, the application should not attempt to free it after use.

The following function provides the inverse functionality.

▸ **h_index_t** ***h_node_get_state_index_from_label***
**(h_node_t** *node*, **h_string_t** *label*)

Return the index of the state of *node* matching the specified *label*. If *node* is not a discrete chance or decision node, or *label* is NULL (these conditions are treated as errors), or no (unique) state matching the specified label exists, $-1$ is returned.

## 5.6 State values

Similar to the above functions for dealing with state labels, we need functions for associating states with points or intervals on the real line. We introduce a common set of functions for handling both of these purposes.

▸ **h_status_t** ***h_node_set_state_value***
**(h_node_t** *node*, **size_t** *s*, **h_double_t** *value*)

Associate *value* with state *s* of *node* (which must be a numeric node).

When *node* is used with the table generator facility, the state values must form an increasing sequence.

For *numbered* nodes, *value* indicates the specific number to be associated with the specified state.

For *interval* nodes, the values specified for state $\mathtt{i}$ and state $\mathtt{i}+1$ are the left and right endpoints of the interval denoted by state $\mathtt{i}$ (the dividing point between two neighboring intervals is taken to belong to the interval to the right of the dividing point). To indicate the right endpoint of the rightmost interval, specify *s* equal to the number of states of *node*.

To specify (semi)infinite intervals, the constant *h_infinity* is defined. The negative of this constant may be specified for the left endpoint of the first interval, and the positive of this constant may be specified for the right endpoint of the last interval.

▶ **h_double_t** ***h_node_get_state_value*** (**h_node_t** *node*, **size_t** *s*)

Return the value associated with state *s* for the numeric node *node*.

The following function provides the inverse functionality.

▶ **h_index_t** ***h_node_get_state_index_from_value***
(**h_node_t** *node*, **h_double_t** *value*)

Return the index of the state of *node* matching the specified *value*:

- If *node* is an interval node, the state index of the interval containing *value* is returned. If an error is detected (that is, if the state values of *node* do not form an increasing sequence), or no interval contains *value*, $-1$ is returned.
- If *node* is a numbered node, the index of the state having *value* as the associated state value is returned. If no (unique) state is found, $-1$ is returned.
- If *node* is not a numeric node (this is an error condition), $-1$ is returned.

## 5.7 Statistical distributions

This section defines the distributions that can be specified using the model feature of HUGIN.

### 5.7.1 Continuous distributions

Continuous distributions are relevant for interval nodes only.

**Normal** A random variable $X$ has a normal (or Gaussian) distribution with mean $\mu$ and variance $\sigma^2$ if its probability density function is of the form

$$p_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}(x-\mu)^2/\sigma^2} \qquad \sigma^2 > 0 \qquad -\infty < x < \infty$$

This distribution is denoted Normal$(\mu, \sigma^2)$.[3]

---

[3] The normal and log-normal distributions are often specified using the *standard deviation* $\sigma$ instead of the *variance* $\sigma^2$. To be consistent with the convention used for CG potentials, we have chosen to use the variance.

**Log-Normal** A random variable $X$ has a log-normal distribution if $\log X$ has a normal distribution with mean $\mu$ and variance $\sigma^2$:

$$p_X(x) = \frac{1}{x\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}(\log x - \mu)^2/\sigma^2} \qquad \sigma^2 > 0 \qquad x > 0$$

This distribution is denoted $\mathrm{LogNormal}(\mu, \sigma^2)$.[3]

An optional *location* parameter $c$ can be specified if $X - c$ has a log-normal distribution: $\mathrm{LogNormal}(\mu, \sigma^2, c)$.

**Gamma** A random variable $X$ has a gamma distribution if its probability density function is of the form

$$p_X(x) = \frac{(x/b)^{a-1} e^{-x/b}}{b\Gamma(a)} \qquad a > 0 \qquad b > 0 \qquad x \geq 0$$

$a$ is called the *shape* parameter, and $b$ is called the *scale* parameter. This distribution is denoted $\mathrm{Gamma}(a, b)$.

An optional *location* parameter $c$ can be specified if $X-c$ has a gamma distribution: $\mathrm{Gamma}(a, b, c)$.

**Beta** A random variable $X$ has a beta distribution if its probability density function is of the form

$$p_X(x) = \frac{1}{B(\alpha, \beta)} \frac{(x-a)^{\alpha-1}(b-x)^{\beta-1}}{(b-a)^{\alpha+\beta-1}} \qquad \alpha > 0 \quad \beta > 0 \quad a \leq x \leq b$$

This distribution is denoted $\mathrm{Beta}(\alpha, \beta, a, b)$. The *standard form* of the beta distribution is obtained by letting $a = 0$ and $b = 1$. This variant is denoted by $\mathrm{Beta}(\alpha, \beta)$ and has the density

$$p_X(x) = \frac{1}{B(\alpha, \beta)} x^{\alpha-1}(1-x)^{\beta-1} \qquad \alpha > 0 \qquad \beta > 0 \qquad 0 \leq x \leq 1$$

**Weibull** A random variable $X$ has a Weibull distribution if its probability density function is of the form

$$p_X(x) = \frac{a}{b}\left(\frac{x}{b}\right)^{a-1} e^{-(x/b)^a} \qquad a > 0 \qquad b > 0 \qquad x \geq 0$$

$a$ is called the *shape* parameter, and $b$ is called the *scale* parameter. This distribution is denoted $\mathrm{Weibull}(a, b)$.

An optional *location* parameter $c$ can be specified if $X-c$ has a Weibull distribution: $\mathrm{Weibull}(a, b, c)$.

**Exponential** A random variable $X$ has an exponential distribution if its probability density function is of the form

$$p_X(x) = e^{-x/b}/b \qquad b > 0 \qquad x \geq 0$$

This distribution is denoted Exponential$(b)$.

An optional *location* parameter $c$ can be specified if $X - c$ has an exponential distribution: Exponential$(b, c)$.

Note: This is a special case of the gamma and Weibull distributions (obtained by letting the shape parameter $a = 1$).

**Uniform** A random variable $X$ has a uniform distribution if its probability density function is of the form

$$p_X(x) = \frac{1}{b-a} \qquad a < b \qquad a \leq x \leq b$$

This distribution is denoted Uniform$(a, b)$.

**Triangular** A random variable $X$ has a triangular distribution if its probability density function is of the form

$$p_X(x) = \frac{2}{b-a} \times \begin{cases} \dfrac{x-a}{m-a} & \substack{a \leq x \leq m \\ a < m} \\[2ex] \dfrac{b-x}{b-m} & \substack{m \leq x \leq b \\ m < b} \end{cases} \qquad a \leq m \leq b \qquad a < b$$

This distribution is denoted Triangular$(a, m, b)$.

**PERT** A PERT distribution is a beta distribution specified using the parameters: $a$ (min), $m$ (mode), $b$ (max) ($a < m < b$), and an optional shape parameter $\lambda$ ($\lambda > 0$). The mean of the beta distribution is assumed to be $(a + \lambda m + b)/(\lambda + 2)$, and from this assumption, formulas for computing the $\alpha$ and $\beta$ parameters of the beta distribution can be derived:

$$\alpha = 1 + \lambda \frac{m-a}{b-a} \qquad \text{and} \qquad \beta = 1 + \lambda \frac{b-m}{b-a}$$

This beta distribution is denoted PERT$(a, m, b, \lambda)$. The 3-parameter variant PERT$(a, m, b)$ is obtained by letting the shape parameter $\lambda = 4$.

This method of specifying a beta distribution is described in [32].

When a continuous distribution is specified for an interval node, the intervals must include the domain specified in the definition of the distribution. The density is defined to be zero outside the intended domain of the distribution (e.g., the density for a gamma distributed random variable $X$ is zero for negative $x$).

Alternatively, the continuous distribution can be *truncated* to fit the intervals of the interval node — see *h_operator_truncate*(70).

Finally, it is recommended not to make the intervals larger than necessary. For example, if a node $A$ has a beta distribution, and $A$ has a child $B$ with a binomial distribution where $A$ is the probability parameter, then the intervals for $A$ should cover the interval $[0,1]$ and nothing more. Otherwise, there would be problems when computing the probabilities for $B$ (because the probability parameter would be out-of-range).

### 5.7.2 Discrete distributions

The following discrete distributions apply to numbered and interval nodes.

**Binomial** A random variable $X$ has a binomial distribution with parameters $n$ (a nonnegative integer) and $p$ (a probability) if

$$P(X=k) = \binom{n}{k} p^k (1-p)^{n-k} \qquad k = 0, 1, \ldots, n$$

This distribution is denoted $\text{Binomial}(n, p)$.

**Poisson** A random variable $X$ has a Poisson distribution with parameter $\lambda$ (a positive real number) if

$$P(X=k) = \frac{e^{-\lambda}\lambda^k}{k!} \qquad \lambda > 0 \qquad k = 0, 1, 2, \ldots$$

This distribution is denoted $\text{Poisson}(\lambda)$.

**Negative Binomial** A random variable $X$ has a negative binomial distribution with parameters $r$ (a positive real number) and $p$ (a probability) if

$$P(X=k) = \binom{k+r-1}{k} p^r (1-p)^k \qquad k = 0, 1, 2, \ldots$$

This distribution is denoted $\text{NegativeBinomial}(r, p)$.

**Geometric** A random variable $X$ that counts the number of failures in a sequence of Bernoulli trials before the first success has a geometric distribution. Let $p$ denote the probability of success, and let $q = 1-p$. Then

$$P(X=k) = pq^k \qquad k = 0, 1, 2, \ldots$$

This distribution is denoted $\text{Geometric}(p)$.

Note: The geometric distribution is a special case of the negative binomial distribution (corresponding to $r = 1$).

When one of the above distributions is specified for a numbered node, the state values for that node must form the sequence 0, 1, ..., $m$ for some $m$. If the distribution is a binomial distribution, all possible outcomes must be included (i.e., $m \geq n$). The other distributions have an infinite number of possible outcomes, so by convention the probability $P(X \geq m)$ is associated with the last state.

If any of the above distributions is specified for an interval node, the intervals must include all possible outcomes. Recall that the intervals of an interval node are taken to be of the form $[a, b)$ (except for the rightmost interval and for infinite endpoints), so, for example, the interval $[0, 2)$ contains the integers 0 and 1.

The "Distribution" operator specifies a user-defined distribution. It applies to all discrete nodes.

**Distribution** If a discrete node $A$ has $n$ states, then Distribution$(e_1, \ldots, e_n)$ means that expression $e_i$ will be evaluated and the result assigned as the probability of the $i$th state (the probabilities need not be normalized). The expressions must be of numeric type and must evaluate to nonnegative values (and at least one of them must be positive).

The Noisy-OR distribution applies to boolean nodes.

**Noisy-OR** Let $b_1, \ldots, b_n$ $(n \geq 1)$ be boolean values, and let $q_1, \ldots, q_n$ $(0 \leq q_i \leq 1)$ be probability values. A random (boolean) variable X has a Noisy-OR distribution if

$$P(X = \texttt{false}) = \prod_{i:b_i = \texttt{true}} q_i$$

This distribution is denoted NoisyOR$(b_1, q_1, \ldots, b_n, q_n)$.

The Noisy-OR distribution can be used to model an event that may be caused by any member of a set of conditions, and the likelihood of causing the event increases if more conditions are satisfied.

The assumptions underlying the Noisy-OR distribution are:

- If all the causing conditions $b_1, \ldots, b_n$ are `false`, then X is `false`.
- If some condition $b_i$ is satisfied then X is `true`, unless some *inhibitor* prevents it. The probability of the inhibitor for $b_i$ being active is denoted by $q_i$. If $b_i$ is the only satisfied condition, it follows that $P(X = \texttt{true}) = 1 - q_i$.
- The mechanism that inhibits one satisfied condition $b_i$ from causing X to be `true` is independent of the mechanism that inhibits another satisfied condition $b_j$ $(i \neq j)$ from causing X to be `true`.

- The causing conditions combine disjunctively, meaning that if a number of conditions are satisfied then X is `true`, unless all the corresponding inhibitors are active.

See [12, Section 3.3.2] and [26, Section 4.3.2] for further details.

Note: The boolean arguments of the NoisyOR operator can be arbitrary expressions — not just simple variables. For example, to introduce a condition that is always satisfied, specify `true` as the corresponding boolean expression.

## 5.8 Generating tables

Normally, the user doesn't need to worry about generating tables from their corresponding models. This is automatically taken care of by the compilation, propagation, and reset-inference-engine operations (by calling the functions described below).

However, it may sometimes be desirable to generate a single table from its model (for example, when deciding how to split a continuous range into subintervals). This is done using the following function.

▶ **h_status_t** ***h_node_generate_table*** (**h_node_t** *node*)

Generate the table of *node* from its model (a missing model is a usage error). Since function nodes do not have tables, *node* must not be a function node.

The table is only generated if the inference engine thinks it is necessary. The following conditions will cause the table to be generated: The model is new or one of its expressions is new (relative to the most recent generation of this table), the number of samples per interval (see *h_model_set_number_of_samples_per_interval*(85) below) for the model of *node* has changed, or a state label (if *node* is a labeled node), a state value, the number of states, or the subtype of *node* or one of its parents has changed since the most recent generation of this table. Removal of a parent of *node* can also cause the table to be generated (this happens if the parent is used in the model). Adding a parent, however, will *not* cause the table to be generated (because the contents of the table would not change).

If the operation fails, the contents of the table will be undefined. If a log-file has been specified (see *h_domain_set_log_file*(93)), then information about the computations (including reasons for failures) is written to the log-file.

Experience and fading tables (see Chapter 10) are not affected by *h_node_generate_table*.

▸ **h_status_t** ***h_domain_generate_tables*** (**h_domain_t** *domain*)

Generate tables for all relevant nodes of *domain*. This is done by calling *h_node_generate_table* for all (non-function) nodes having a model. Hence, the description of that function also applies here.

The operation is aborted if table generation fails for some node. This implies that some tables may have been successfully generated, some may not have been generated at all, and one table has been only partially generated.

The following function is identical to the above function, except that it operates on classes instead of domains.

▸ **h_status_t** ***h_class_generate_tables*** (**h_class_t** *class*)

Generate tables for all relevant nodes of *class*. See the above description of *h_domain_generate_tables* for further details.

As mentioned above, information about the computations performed when generating a table is written to the log-file. For classes, a log-file is specified using the following function.

▸ **h_status_t** ***h_class_set_log_file*** (**h_class_t** *class*, **FILE** ∗*log_file*)

Set the file to be used for logging by subsequent HUGIN API operations that apply to *class*. (Currently, only table generation information is written to log-files for classes.)

If *log_file* is NULL, no log will be produced. If *log_file* is not NULL, it must be a `stdio` text file, opened for writing (or appending). Writing is done sequentially (i.e., no seeking is done).

See also Section 6.4.

## 5.9 How the computations are done[4]

The most complex cases are nodes with interval parents. Assume, for simplicity, that we have a chance node with one interval parent (more than one interval parent is a trivial generalization of the simple case).

For a given interval of the parent (i.e., for a specific parent state configuration), we compute many probability distributions for the child, each distribution being obtained by instantiating the parent to a value in the interval under consideration.[5] The average of these distributions is used as the con-

---

[4] The information provided in this section only applies to generation of node tables from models. See Section 8.7 for information on the computations performed using models associated with function nodes.

[5] For semi-infinite intervals, only one value is used. This value is chosen to be close to the finite endpoint. Intervals that are infinite in both directions are discouraged — the behavior is unspecified.

ditional probability distribution for the child given the parent is in the interval state considered. (For this scheme to work well, the intervals should be chosen such that the discretised distributions corresponding to the chosen points in the parent interval are not "too different" from each other.)

By default, 25 values are taken within each bounded interval of an interval parent: The interval is divided into 25 subintervals, and the midpoints of these subintervals are then used in the computations. A large number of values gives high accuracy, and a small number of values results in fast computations. The number of values used can be changed by the following function.

▸ **h_status_t** ***h_model_set_number_of_samples_per_interval***
(**h_model_t** *model*, **size_t** *count*)

Specify that *count* values should be sampled from each bounded interval of an interval parent when generating a table from *model*.

▸ **h_count_t** ***h_model_get_number_of_samples_per_interval***
(**h_model_t** *model*)

Retrieve the count indicating the number of samples that would be used if a table were to be generated from *model* now.

### 5.9.1 Deterministic relationships

If the type of the expression for the parent state configuration under consideration is *not* 'distribution,' then we have a deterministic relationship.

The expression must then evaluate to something that matches one of the states of the child node. For labeled, boolean, and numbered nodes, the value must match exactly one of the state values or labels. For interval nodes, the value must belong to one of the intervals represented by the states of the child node.

If one or more of the parents are of interval subtype, then a number of samples (25 by default) within each (bounded) interval will be generated. Each of these samples will result in a "degenerate" distribution (i.e., all probability mass will be assigned to a single state) for the child node. The final distribution assigned to the child node is the average over all generated distributions. This amounts to counting the number of times a given child state appears when applying the deterministic relationship to the generated samples.

If all samples within a given parent state interval map to the same child state, then the resulting child distribution is independent of the number of samples generated. It is recommended that the intervals be chosen such that this is the case.

If this is not feasible, then the number of samples generated should be large in order to compensate for the sampling errors. Typically, some of the child states will have a frequency count one higher (or lower) than the "ideal" count.

**Example 5.3** Let $X$ be an interval node having $[0, 1)$ as one of its states (intervals). Let $Y$ be a child of $X$ having $[0, 1)$, $[1, 2)$, and $[2, 3)$ as some of its states. Assume that $Y$ is specified through the deterministic relation $Y = 3X$. If 25 samples for $X$ are taken within the interval $[0, 1)$, then 8, 9, and 8 of the computed values will fall in the intervals $[0, 1)$, $[1, 2)$, and $[2, 3)$ of $Y$, respectively. Ideally, the frequency counts should be the same, resulting in a uniform distribution over the three interval states. ∎

# Chapter 6

# Compilation

Before a belief network or a LIMID can be used for inference, it must be compiled.

This chapter describes functions to compile domains, control triangulations, and to perform other related tasks, such as approximation and compression.

## 6.1 What is compilation?

The compilation process involves the following sequence of steps:[1]

(1) The function nodes are removed.

(2) The network is converted into its *moral* graph: The parents of each node are "married" (i.e., links are added between them), and the directions of the links are dropped.

(3) The utility nodes are removed.

(4) The graph is triangulated. (This is described in detail below.)

(5) The *cliques* (maximal complete sets) of the triangulated graph are identified, and the collection of cliques is organized as a tree (with the cliques forming the nodes of the tree). Such a tree is called a *junction tree*. If the original network is disconnected, there will be a tree for each connected component.

(6) Finally, potentials are associated with the cliques and the links (the *separators*) of each junction tree. These potentials are initialized from the evidence and the conditional probability tables (and the policies and the utility tables in the case of LIMIDs), using a sum-propagation (see Section 9.1).

---

[1] The network presented for compilation is not actually modified by these steps. The compilation process should be thought of as working on a copy of the original network.

All steps, except the triangulation step, are quite straightforward. The triangulation problem is known to be $\mathcal{NP}$-hard for all reasonable criteria of optimality, so (especially for large networks) finding the optimal solution is not always feasible. The HUGIN API provides several methods for finding triangulations: five heuristic methods based on local cost measures, and a combined exact/heuristic method capable of minimizing the storage requirements (i.e., the sum of state space sizes) of the cliques of the triangulated graph, if sufficient computational resources are available.

Alternatively, a triangulation can be specified in the form of an elimination sequence.

An *elimination sequence* is an ordered list containing each node of the graph exactly once. An elimination sequence $\langle v_1, \dots, v_n \rangle$ generates a triangulated graph from an undirected graph as follows: Complete the set of neighbors of $v_1$ in the graph (i.e., for each pair of unconnected neighbors, add a *fill-in* edge between them). Then eliminate $v_1$ from the graph (i.e., delete $v_1$ and edges incident to $v_1$). Repeat this process for all nodes of the graph in the specified order. The input graph with the set of generated fill-in edges included is a triangulated graph.

The elimination sequence can be chosen arbitrarily, except for belief networks with both discrete and continuous nodes.

In order to ensure correct inference, the theory of CG belief networks (see [7, 16, 19]) requires the continuous nodes to be eliminated before the discrete nodes.

Let $\Delta$ denote the set of discrete nodes, and let $\Gamma$ denote the set of continuous nodes. A valid elimination sequence must contain the nodes of $\Gamma$ (in any order) followed by the nodes of $\Delta$ (in any order).

Let $x, y \in \Delta$. It is well-known that, for any valid elimination sequence, the following must hold for the corresponding triangulated graph: If between $x$ and $y$ there exists a path lying entirely in $\Gamma$ (except for the end-points), then $x$ and $y$ are connected. If $x$ and $y$ are not connected in the moral graph, we say that $x$ and $y$ are connected by a *necessary fill-in* edge. Conversely, it can be shown that a triangulated graph with this property has a valid elimination sequence.

Let $G$ be the moral graph extended with all necessary fill-in edges. The neighbors of a connected component of $G[\Gamma]$ form a complete separator of $G$ (unless there is exactly one connected component having all nodes of $\Delta$ as neighbors). A maximal subgraph that does not have a complete separator is called a *prime component* [25]. We note that a triangulation formed by the union of triangulations of the prime components of $G$ has the property described above.

We can now further specify the triangulation step. The input to this step is the moral graph (without utility nodes).

- Add all necessary fill-in links.
- Find the prime components of the graph.
- Triangulate the prime components. The union of these triangulations constitutes the triangulation of the input graph.
- Generate an elimination sequence for the triangulation.

See [7, 11, 12, 16] for further details on the compilation process.

## 6.2 Compilation

▸ **h_status_t** ***h_domain_compile*** (**h_domain_t** *domain*)

Compile *domain*, using the default triangulation method (unless *domain* is already triangulated — see Section 6.3); *domain* must contain at least one chance or decision node.

It is considered an error, if *domain* is already compiled.

The junction trees are initialized using the sum-propagation operation, incorporating all available evidence in "normal" mode. If this propagation fails, the compilation also fails.

The compilation process can use large amounts of memory (and time), depending on the size and structure of the network, and the choice of triangulation method. The application should be prepared to handle out-of-memory conditions.

▸ **h_boolean_t** ***h_domain_is_compiled*** (**h_domain_t** *domain*)

Test whether *domain* is compiled.

## 6.3 Triangulation

The choice of triangulation method in the compilation process can have a huge impact on the size of the compiled domain, especially if the network is large. If the default triangulation method used by *h_domain_compile*$^{(89)}$ does not give a good result, another option is available: The network may be triangulated in advance (before calling *h_domain_compile*). The HUGIN API provides a choice between several built-in triangulation methods, or, alternatively, a triangulation can be specified in the form of an elimination sequence.

The HUGIN API defines the enumeration type **h_triangulation_method_t**. This type contains the following six constants, denoting the built-in triangulation methods: *h_tm_clique_size*, *h_tm_clique_weight*, *h_tm_fill_in_size*, *h_tm_fill_in_weight*, *h_tm_best_greedy*, and *h_tm_total_weight*. The first four of these methods are simple elimination based heuristics, the fifth method is a combination of the first four methods, and the last method is a combined exact/heuristic method capable of producing an optimal triangulation, if sufficient computational resources (primarily storage) are available.

The elimination based heuristics follow a common scheme: Nodes are eliminated sequentially in a "greedy" manner (that is, with no "look-ahead") from the prime component being triangulated. If the current graph has a node with all its (uneliminated) neighbors forming a complete set, that node is eliminated next (this is optimal with respect to minimizing the size of the largest clique of the final triangulated graph). If no such node exists, a node with a best "score" according to the selected heuristic is chosen.

Let $C(v)$ denote the set comprised of $v$ and its (uneliminated) neighbors. The elimination based heuristics, implemented in HUGIN, define a score based on $C(v)$ for each node $v$ (and with "best" defined as "minimum"). The scores defined by the four basic heuristics are:

***h_tm_clique_size*** The score is equal to the cardinality of $C(v)$.

***h_tm_clique_weight*** The score is $s + sm(m + 3)/2$, where $s$ is the product of the number of states of the discrete nodes in $C(v)$, and $m$ is the number of continuous nodes in $C(v)$. This score is equal to the number of data elements stored in a table holding $C(v)$.[2]

***h_tm_fill_in_size*** The score is equal to the number of fill-in edges needed to complete $C(v)$.

***h_tm_fill_in_weight*** The score is equal to the sum of the weights of the fill-in edges needed to complete $C(v)$, where the weight of an edge is defined as the product of the number of states of the nodes connected by the edge (in this context, continuous nodes are equivalent to discrete nodes with only one state).

It is possible for these heuristics to produce a non-minimal triangulation (a triangulation is *minimal* if no proper subset of the fill-in edges produces a triangulated graph). In order to obtain a minimal triangulation, a maximal set of *redundant fill-in edges* is removed from the triangulation. This is done using a variant of an algorithm by Kjærulff [14].

The *h_tm_best_greedy* triangulation method combines the results of the four basic heuristics: It tries all of the heuristics and uses the best result. (Here,

---

[2] This is not always proportional to the actual storage requirements (measured in bytes) of the table. This is because the data elements can be of different types: The data elements associated with state configurations of discrete nodes are of type **h_number_t**, while the data elements associated with continuous nodes (such as mean and covariance values) are of type **h_double_t**. In a single-precision version of the HUGIN API, **h_number_t** is a 4-byte quantity, but in a double-precision version, it is an 8-byte quantity. In both versions, **h_double_t** is an 8-byte quantity.

"best result" refers to the triangulation with the smallest sum of clique weights — as defined by the *h_tm_clique_weight* heuristic.) Notice that different heuristics can produce the best result on different prime components.

The *h_tm_best_greedy* heuristic is the method used by *h_domain_compile*(89), if the domain being compiled has not been triangulated in advance.

The heuristic triangulation methods are very fast, but sometimes the generated triangulations are quite bad. As an alternative, the HUGIN API provides the *h_tm_total_weight* triangulation method. This method can produce an optimal triangulation, if sufficient computational resources are available. The method considers a triangulation to be optimal, if it is minimal and the sum of clique weights (as defined by the *h_tm_clique_weight* heuristic) is minimum.

For some large networks, use of the *h_tm_total_weight* triangulation method has improved time and space complexity of inference by an order of magnitude (sometimes even more), compared to the heuristic methods described above.

The *h_tm_total_weight* triangulation algorithm can be outlined as follows:

First, all minimal separators of the prime component being triangulated are identified (using an algorithm by Berry *et al* [2]). From this set of minimal separators, an initial triangulation is found using greedy selection of separators until the component is triangulated. The cost of this triangulation is then used as an upper bound on the cost of optimal triangulations, and all separators that are too expensive relative to this upper bound are discarded. Then the prime component is split using each of the remaining separators. The pieces resulting from such a split are called *fragments* (cf. Shoikhet and Geiger [29]; another term commonly used is 1-*block*). Every (minimal) triangulation of a fragment can be decomposed into a clique and corresponding triangulated subfragments. Using this fact, optimal triangulations are found for all fragments. (The clique generation process of this search makes use of a characterization of cliques in minimal triangulations given by Bouchitté and Todinca [3].) Finally, an optimal triangulation of the prime component is identified (by considering all possible splits of the component using minimal separators).

Some prime components have more minimal separators than the memory of a typical computer can hold. In order to handle such components, an upper bound on the number of separators can be specified: If the search for minimal separators determines that more than the specified maximum number of separators exist, then the component is split using one of the separators already found.[3] The fragments obtained are then recursively triangulated.

---

[3] The separator is selected using a heuristic method that considers the cost of the separator and the size of the largest fragment generated, when the component is split using the separator. The heuristic method used for this selection may change in a future version of the HUGIN API.

Experience suggests that 100 000 is a good number to use as an upper bound on the number of minimal separators.

▸ **h_status_t *h_domain_set_max_number_of_separators***
(**h_domain_t** *domain*, **size_t** *count*)

Specify *count* as the maximum number of minimal separators to generate when using the *h_tm_total_weight* method for triangulating *domain*. If *count* is zero, then the bound is set to "unlimited," which is also the default value.

▸ **h_count_t *h_domain_get_max_number_of_separators***
(**h_domain_t** *domain*)

Retrieve the current setting for *domain* of the maximum number of separators to generate for the *h_tm_total_weight* triangulation method. If an error occurs, a negative number is returned.

▸ **h_status_t *h_domain_triangulate***
(**h_domain_t** *domain*, **h_triangulation_method_t** *tm*)

Perform triangulation of *domain* using triangulation method *tm*. It is considered an error, if *domain* is already triangulated.

As mentioned above, it is also possible to supply a triangulation explicitly through an elimination sequence. This is convenient if a better triangulation is available from other sources.

▸ **h_status_t *h_domain_triangulate_with_order***
(**h_domain_t** *domain*, **h_node_t** ∗*order*)

Triangulate *domain* using the NULL-terminated list *order* of nodes as elimination sequence (*order* must contain each chance and each decision node of *domain* exactly once, and continuous nodes must precede discrete nodes — see Section 6.1).

It is considered an error, if *domain* is already triangulated.

The triangulation functions construct the cliques and the junction trees, but do not allocate storage for the data arrays of the clique and separator tables. This storage (which is the largest part of the storage consumed by a typical compilation) is allocated by the *h_domain_compile*(89) function. However, the total size of the junction tree tables can be queried before this storage is allocated — see *h_jt_get_total_size*(101) and *h_jt_get_total_cg_size*(101).

▸ **h_boolean_t *h_domain_is_triangulated*** (**h_domain_t** *domain*)

Test whether *domain* is triangulated.

▸ **h_node_t** ∗***h_domain_get_elimination_order*** (**h_domain_t** *domain*)

Return a NULL-terminated list of nodes in the order used to triangulate *domain*. If an error is detected (e.g., *domain* has not been triangulated), NULL is returned.

The list holding the elimination order is stored within the domain structure. It must not be deallocated by the application.

As indicated above, it can be a lot of work to find good triangulations. Therefore, it is convenient to store the corresponding elimination orders in separate files for later use. The following function helps in managing such files: It parses a text file holding a list of node names (separated by spaces, tabs, newlines, or comments — see Section 12.7).

▸ **h_node_t** ∗***h_domain_parse_nodes***
(**h_domain_t** *domain*, **h_string_t** *file_name*,
**void** (∗*error_handler*) (**h_location_t**, **h_string_t**, **void** ∗),
**void** ∗*data*)

This function parses the list of node names stored in the file with name *file_name*. The node names must identify nodes of *domain*; it is an error, if some node cannot be found. If no error is detected, a NULL-terminated dynamically allocated array holding the nodes is returned. If an error is detected, NULL is returned.

Note: Only the user application has a reference to the array, so the user application is responsible for deallocating the array when it is done using it.

The *error_handler* and *data* arguments are used for error handling. This is similar to the error handling done by the other parse functions. See Section 12.8 for further information.

The *h_domain_parse_nodes* function can be used to parse any file containing a node list (not just node lists representing elimination orders for triangulations). Therefore, for completeness, a similar parse function is provided for classes:

▸ **h_node_t** ∗***h_class_parse_nodes***
(**h_class_t** *class*, **h_string_t** *file_name*,
**void** (∗*error_handler*) (**h_location_t**, **h_string_t**, **void** ∗),
**void** ∗*data*)

## 6.4 Getting a compilation log

It is possible to get a log of the actions taken by the compilation process (the elimination order chosen, the fill-in edges created, the cliques, the junction trees, etc.). Such a log is useful for debugging purposes (e.g., to find out why the compiled version of the domain became so big).

▸ **h_status_t** ***h_domain_set_log_file*** (**h_domain_t** *domain*, **FILE** ∗*log_file*)

Set the file to be used for logging by subsequent calls to HUGIN API functions.

If *log_file* is NULL, no log will be produced. If *log_file* is not NULL, it must be a `stdio` text file, opened for writing (or appending). Writing is done sequentially (i.e., no seeking is done).

Note that if a log is wanted, and (some of) the nodes (that are mentioned in the log) have not been assigned names, then names will automatically be assigned (through calls to the *h_node_get_name*(39) function).

**Example 6.1** The following code fragment illustrates a typical compilation process.

```
h_domain_t d;
FILE *log;
...
log = fopen ("mydomain.log", "w");
h_domain_set_log_file (d, log);
h_domain_triangulate (d, h_tm_clique_weight);
h_domain_compile (d);
h_domain_set_log_file (d, NULL);
fclose (log);
```

A file (*log*) is opened for writing and assigned as log file to domain *d*. Next, triangulation, using the *h_tm_clique_weight* heuristic, is performed. Then the domain is compiled. When the compilation process has completed, the log file is closed. Note that further writing to the log file (by HUGIN API functions) is prevented by setting the log file of domain *d* to NULL. ∎

In addition to the compilation and triangulation functions, the *h_node_generate_table*(83), *h_domain_learn_structure*(155), *h_domain_learn_tables*(157), and *h_domain_learn_class_tables*(161) functions also use the log file to report errors, warnings, and other information. HUGIN API functions that use *h_node_generate_table* internally (for example, the propagation operations call this function when tables need to be regenerated from their models) also write to the log file (if it is non-NULL).

## 6.5 Uncompilation

▸ **h_status_t *h_domain_uncompile* (h_domain_t** *domain*)

Remove the data structures of *domain* produced by the *h_domain_compile*(89), *h_domain_triangulate*(92), and *h_domain_triangulate_with_order*(92) functions. If *domain* is not compiled or triangulated, nothing is done.

Note that any opaque references to objects within the compiled structure (e.g., clique and junction tree objects) are invalidated by an "uncompile" operation.

Also note that many of the editing functions described in Chapter 2 automatically perform an "uncompile" operation whenever something is changed about *domain* that invalidates the compiled structure. When this happens, the domain must be recompiled (using *h_domain_compile*(89)) before it can be used for inference.

## 6.6 Compression

Most of the memory consumed by a compiled domain is used for storing the data of the clique and separator tables. Many of the entries of these tables might be zero, reflecting the fact that these state combinations in the model are impossible. Zeros in the junction tree tables arise from logical relations within the model. Logical relations can be caused by deterministic nodes, approximation, or propagation of evidence. To conserve memory, the data elements with a value of zero can be removed, thereby making the tables smaller. This process is called *compression*.

▸ **h_double_t *h_domain_compress*** (**h_domain_t** *domain*)

Remove the zero entries from the clique and separator tables of the junction trees of *domain*.

Compression can only be applied to (compiled) ordinary belief networks. [Continuous nodes are allowed, but approximation only applies to configurations of states of the discrete nodes.]

If *domain* has a memory backup (see *h_domain_save_to_memory*(125)), it will be deleted as part of the compression operation.

The compression function returns a measure of the compression achieved. This measure should be less than 1, indicating that the compressed domain requires less memory than the uncompressed version. If the measure is larger than 1, the compressed domain will actually use more memory than the uncompressed version. This can happen if only a few elements of the junction tree tables are zero, so that the space savings achieved for the tables are outweighed by the extra space required to support the more complex table operations needed to do inference in compressed domains.

If an error occurs, *h_domain_compress* returns a negative number.

If a domain has been compressed, and more zeros have been introduced by new evidence or approximation (Section 6.7), then the domain can be compressed further to take advantage of the new zero entries.

Note that some operations, such as extracting marginal tables (with more than one node), cannot be done with a compressed domain. Those operations will fail with the error code *h_error_compressed*.

Note also that compression is only possible after compilation is completed. This means that enough memory to store the uncompressed compiled domain must be available. Compression is maintained in saved domains (when

HUGIN KB files are used), making it possible to use a computer with a large amount of (virtual) memory to compile and compress a domain and then load the compressed domain on computers with less memory.

The zero elements in the junction tree tables do not contribute anything to the beliefs computed by the HUGIN inference engine. Thus, their removal doesn't change the results of inference. The only effect of compression is to save memory and to speed up inference.

▶ **h_boolean_t** ***h_domain_is_compressed*** (**h_domain_t** *domain*)

Test whether *domain* is compressed.

## 6.7 Approximation

The discrete part of a clique potential consists of a joint probability distribution over the set of state configurations of the discrete nodes of the clique.

The approximation technique — implemented in the HUGIN API — is based on the assumption that very small probabilities in this probability distribution reflect (combinations of) events that will hardly ever occur in practice. Approximation is the process of setting all such "near-zero" probabilities to zero. The primary objective of this process is to minimize storage consumption through compression.

It must be emphasized that this approximation technique should only be used when one is *not* interested in the probabilities of unlikely states as the relative error — although small in absolute terms — for such probabilities can be huge. Approximation should be used only if all one is interested in is to identify the most probable state(s) for each node given evidence.

▶ **h_double_t** ***h_domain_approximate***
(**h_domain_t** *domain*, **h_double_t** $\varepsilon$)

The effect of this function is as follows: For each clique in *domain*, a value $\delta$ is computed such that the sum of all elements less than $\delta$ in (the discrete part of) the clique table is less than $\varepsilon$. These elements are then set to $0$. In effect, $\varepsilon$ specifies the maximum probability mass to remove from each clique.

Approximation can only be applied to (compiled) ordinary belief networks. [Continuous nodes are allowed, but approximation only applies to configurations of states of the discrete nodes.]

The type of equilibrium on the junction trees of *domain* must be 'sum,' incorporating all evidence (if any) specified for *domain* in 'normal' mode (Section 9.1). This condition holds right after a (successful) compilation, which is when an approximation is usually performed.

The approximation function returns the probability mass remaining in the entire domain, letting you know how much precision you have "lost." Note that this is not the same as $1 - \varepsilon$, as the $\varepsilon$ value is relative to each clique. Typically, the total amount of probability mass removed will be somewhat larger than $\varepsilon$.

If *h_domain_approximate* fails, a negative value is returned.

The annihilation of small probabilities within the clique potentials can be thought of as entering a special kind of evidence. As part of the approximation process, this evidence is propagated throughout the junction trees, thereby reaching an equilibrium state on all junction trees. The joint probability of the evidence is the value returned by *h_domain_approximate*.

An approximation operation should be followed by a compression operation. If not, the approximation will be lost when the inference engine is reset (which can, e.g., happen as part of a propagation operation when evidence has been retracted and/or some conditional probability tables have changed).

**Example 6.2** Example 6.1 can be extended with approximation and compression as follows.

```
h_domain_t d;
FILE *log;
...
log = fopen ("mydomain.log", "w");
h_domain_set_log_file (d, log);
h_domain_triangulate (d, h_tm_clique_weight);
h_domain_compile (d);
h_domain_set_log_file (d, NULL);
fclose (log);
h_domain_approximate (d, 1E-8);
h_domain_compress (d);
h_domain_save_as_kb (d, "mydomain.hkb", NULL);
```

Probability mass of 'weight' up to $10^{-8}$ is removed from each clique of the compiled domain using approximation. Then the zero elements are removed from the clique potentials using compression. Finally, the domain is saved as an HKB file (this is necessary in order to use the compressed domain on another computer with insufficient memory to create the uncompressed version of the domain). ∎

It is difficult to give a hard-and-fast rule for choosing a good value for $\varepsilon$ (i.e., one that achieves a high amount of compression and doesn't introduce unacceptable errors). In general, the ratio between the error introduced by the approximation process and the joint probability of the evidence obtained when using the approximated domain should not become too large. If it does, the evidence should be processed by the unapproximated version. A "threshold" value for this ratio should be determined through empirical tests for the given domain.

See [9] for an empirical analysis of the approximation method.

▸ **h_double_t** ***h_domain_get_approximation_constant***
(**h_domain_t** *domain*)

Return the approximation constant of the most recent (explicit or implicit) approximation operation. If an error occurs, a negative number is returned.

An implicit approximation takes place when you change some conditional probability tables of a compressed domain, and then perform a propagation operation. Since some (discrete) state configurations have been removed from a compressed domain, the probability mass of the remaining configurations will typically be less than 1; *h_domain_get_approximation_constant* returns that probability mass.

# Chapter 7

# Cliques and Junction Trees

The compilation process creates a secondary structure of the belief network or LIMID model. This structure, the *junction tree*, is used for inference. Actually, since the network may be disconnected, there can be more than one junction tree. In general, we get a forest of junction trees — except for a LIMID, which is compiled into a single junction tree (in order to ensure correct inference).

The cliques of the triangulated graph form the nodes of the junction trees. The connections (called *separators*) between the cliques (i.e., the edges of the junction trees) are the "communication channels" used by *Collect-Evidence* and *DistributeEvidence* (see Chapter 9). Associated with each clique and separator is a function from the state space of the clique/separator to the set of (nonnegative) real numbers — this is called the (probability) potential, and it is represented as a table (Chapter 4). If the input to the compilation process contains utilities, there will be an additional potential associated with the cliques and the separators: a utility potential which is similar to the probability potential (except that the numbers may be negative).

The triangulation process constructs the cliques and the junction trees, but does not allocate storage for the data arrays of the clique and separator tables. This storage (which is the largest part of the storage consumed by a typical compilation) is allocated by the *h_domain_compile*(89) function. However, the total size of the junction tree tables can be queried before this storage is allocated. This is useful for evaluating the quality of the triangulation.

The HUGIN API provides functions to access the junction forest and to traverse the trees of the forest.

## 7.1 Types

We introduce the opaque pointer types **h_clique_t** and **h_junction_tree_t**. They represent clique and junction tree objects, respectively.

## 7.2 Junction trees

The HUGIN API provides a pair of functions to access the junction trees of a triangulated domain.

▸ **h_junction_tree_t** ***h_domain_get_first_junction_tree***
(**h_domain_t** *domain*)

Return the first junction tree in the list of junction trees of *domain*. If an error is detected (e.g., *domain* is not triangulated), a NULL pointer is returned.

▸ **h_junction_tree_t** ***h_jt_get_next*** (**h_junction_tree_t** *jt*)

Return the successor of junction tree *jt*. If there is no successor, NULL is returned. If an error is detected (i.e., if *jt* is NULL), NULL is returned.

Another way to access junction trees is provided by the *h_clique_get_junction_tree* and *h_node_get_junction_tree* functions.

▸ **h_junction_tree_t** ***h_clique_get_junction_tree*** (**h_clique_t** *clique*)

Return the junction tree to which *clique* belongs. If an error is detected, NULL is returned.

▸ **h_junction_tree_t** ***h_node_get_junction_tree*** (**h_node_t** *node*)

Return the junction tree to which *node* belongs (*node* must not be a function node). If an error is detected, NULL is returned.

As explained in Section 6.1, utility (and function) nodes are not present in the junction trees. But because only one junction tree is constructed for a LIMID network, *h_node_get_junction_tree* returns that junction tree if *node* is a utility node.

We also provide a function to access the collection of cliques comprising a given junction tree.

▸ **h_clique_t** *∗**h_jt_get_cliques*** (**h_junction_tree_t** *jt*)

Return a NULL-terminated list of the cliques that form the set of vertices of junction tree *jt*. If an error is detected, a NULL pointer is returned.

The storage holding the list of cliques returned by *h_jt_get_cliques* is owned by the junction tree object, and must therefore not be deallocated by the application.

▸ **h_clique_t** ***h_jt_get_root*** (**h_junction_tree_t** *jt*)

Return the "root" of junction tree *jt*. If the junction tree is undirected (which it is unless there are continuous nodes involved), this is just an arbitrarily selected clique. If the junction tree is directed, a *strong* root (see [7, 16, 19]) is returned (there may be more than one of those). If an error is detected, NULL is returned.

▸ **size_t** ***h_jt_get_total_size*** (**h_junction_tree_t** *jt*)

Return the total size (i.e., the total number of discrete configurations) of all clique and separator tables associated with junction tree *jt*.

Each discrete table configuration has a numeric quantity of type **h_number_t** associated with it. In a single-precision version of the HUGIN API, this is a 4-byte quantity. In a double-precision version, this is an 8-byte quantity.

Note that both probability and utility tables are counted (that is, the discrete clique and separator configurations are counted twice if there are utility potentials in the junction tree).

If an error occurs (e.g., the total size of all tables exceeds the maximum value of the **size_t** type), "(**size_t**) $-1$" is returned.

▸ **size_t** ***h_jt_get_total_cg_size*** (**h_junction_tree_t** *jt*)

Return the total CG size of all clique and separator tables associated with junction tree *jt*. This counts the total number of CG data elements of all tables. Each such data element occupies 8 bytes.

If the junction tree contains no CG nodes, the tables contain no CG data. In this case (only), the function returns 0.

If an error occurs, "(**size_t**) $-1$" is returned.

## 7.3 Cliques

Each clique corresponds to a maximal complete set of nodes in the triangulated graph. The members of such a set can be retrieved from the corresponding clique object by the following function.

▸ **h_node_t** $*$***h_clique_get_members*** (**h_clique_t** *clique*)

Return a NULL-terminated list of the nodes comprising the members of *clique*. If an error is detected, NULL is returned.

The storage holding the list of nodes is the actual member list stored within *clique*, and must therefore not be freed by the application.

▸ **h_clique_t** $*$***h_clique_get_neighbors*** (**h_clique_t** *clique*)

Return a NULL-terminated list of cliques containing the neighbors of *clique* in the junction tree to which *clique* belongs. If an error is detected, NULL is returned.

The storage used for holding the clique list returned by *h_clique_get_neighbors* is owned by *clique*, and must therefore not be freed by the application.

## 7.4 Traversal of junction trees

The *h_jt_get_root*(100) and *h_clique_get_neighbors*(101) functions can be used to traverse a junction tree in a recursive fashion.

**Example 7.1** The following code outlines the structure of the *DistributeEvidence* function used by the propagation algorithm (see [11] for further details).

```
void distribute_evidence (h_clique_t self, h_clique_t parent)
{
    h_clique_t *neighbors = h_clique_get_neighbors (self);

    if (parent != 0)
        /* absorb from parent */ ;

    for (h_clique_t *n = neighbors; *n != 0; n++)
        if (*n != parent)
            distribute_evidence (*n, self);
}
...
{
    h_junction_tree_t jt;
    ...
    distribute_evidence (h_jt_get_root (jt), 0);
    ...
}
```

The *parent* argument of *distribute_evidence* indicates the origin of the invocation; this is used to avoid calling the caller. ∎

# Chapter 8

# Evidence and Beliefs

The first step of the inference process is to make the inference engine aware of the evidence. This is called *entering the evidence*, and it can be done before or after the compilation step (but note that the compilation process uses the available evidence to initialize the junction tree potentials). Moreover, an "uncompile" operation does not remove any already entered evidence (this is worth noting because many HUGIN API functions perform implicit "uncompile" operations).

Typically, an item of evidence has the form of a statement specifying the state (or value) of a variable. This type of evidence is entered to the HUGIN inference engine by the *h_node_select_state*(105) function for discrete nodes and by the *h_node_enter_value*(106) function for continuous nodes. However, for discrete chance nodes, more general items of evidence, called "likelihood" or "multi-state" evidence, can be specified: This type of evidence is entered using the *h_node_enter_finding*(106) function.

This chapter explains how to enter and retract evidence, how to determine independence properties induced by evidence and network structure, how to retrieve beliefs and expected utilities, how to compute values of function nodes, how to examine evidence, and how to save evidence as a case file for later use.

Chapter 9 explains how to propagate evidence in order to compute updated beliefs and other results.

## 8.1 Evidence

### 8.1.1 Discrete evidence

Associated with each discrete node in a HUGIN domain model is a function that assigns a nonnegative real number to each state of the node. We sometimes refer to such a function as an *evidence potential* or a *finding vector*.

Initially, before any evidence has been entered, all finding vectors consist of 1-elements only. Such evidence is termed *vacuous*.

If the finding vector for a node has exactly one positive element, the node is said to be *instantiated*. The function *h_node_select_state*$^{(105)}$ instantiates a node to a specific state (using 1 as the finding value of the specified state).

In general, specifying 0 as the finding value of a state declares the state to be impossible. All finding vectors must have at least one positive element. If not, inference will fail with an "impossible evidence" error code: *h_error_inconsistency_or_underflow*$^{(120)}$.

If a finding vector has two (or more) 1-elements, and the remaining elements are 0, we call the finding vector a *multi-state* finding.

If a finding vector has at least one element that is $\neq 0$ and $\neq 1$, the finding vector is called a *likelihood*. The following examples illustrate the use of likelihoods.

**Example 8.1** Let $A$ be the node that we wish to enter likelihood evidence for. Now, suppose we add a new node $B$ as a child of $A$ and specify the conditional probability table $P(B|A)$ as follows:

| | $a_1$ | $a_2$ |
|---|---|---|
| $b_1$ | 0.3 | 0.4 |
| $b_2$ | 0.7 | 0.6 |

Then, entering the observation $B = b_1$ in the modified network is equivalent to entering the likelihood $(0.3, 0.4)$ for $A$ in the original network.

This feature can be used for inexact observations. Suppose $A$ represents something that we cannot observe with 100% certainty, and $B$ represents our observation of $A$ (such that state $b_i$ of $B$ corresponds to state $a_i$ of $A$). If there is a 10% risk of making a wrong observation, then $P(B|A)$ would be:

| | $a_1$ | $a_2$ |
|---|---|---|
| $b_1$ | 0.9 | 0.1 |
| $b_2$ | 0.1 | 0.9 |

If $B$ is part of the network, then we would enter either $B = b_1$ or $B = b_2$ according to our actual observation. If $B$ is not part of the network, we would instead enter either $(0.9, 0.1)$ or $(0.1, 0.9)$ as likelihood for $A$. ∎

**Example 8.2** Suppose we want to make inference pretending that some "root" node has some other prior distribution than the specified $P(A)$. This can be done by specifying a likelihood equal to the quotient of the desired prior and the original prior. (This trick, of course, only works when division by zero is not involved.) ∎

An instantiated node is said to have *hard* evidence. All other types of (non-vacuous) evidence are called *soft* evidence.

### 8.1.2 Continuous evidence

Evidence for continuous nodes always take the form of a statement that a node is known to have a specific value. Such evidence is entered using the *h_node_enter_value*(106) function.

This type of evidence is an example of *hard* evidence.

### 8.1.3 Evidence in LIMIDs

Decision nodes can only have hard evidence (and the finding value must be 1). In addition, a chance node *with* evidence must not have a decision node *without* evidence as ancestor in the network obtained by ignoring information links (i.e., links pointing at decision nodes). Such an evidence scenario would amount to observing the consequences of a decision before the decision is made, and an attempt to perform inference given such evidence fails with an "invalid evidence" error code: *h_error_invalid_evidence*(120).

## 8.2 Entering evidence

The functions described in this section can be used to enter evidence for a given set of nodes (one node at a time). It is also possible to load the evidence for all nodes at once, when the evidence is stored in a case file (see *h_domain_parse_case*(115)) or as a case in main memory (see *h_domain_enter_case*(150)).

The following function handles evidence taking the form of instantiations of discrete variables.

▸ **h_status_t *h_node_select_state*** (**h_node_t** *node*, **size_t** *state*)

Select state *state* of node *node* (which must be a discrete chance or decision node). This is equivalent to specifying the finding value 1 for state *state* and 0 for all other states (see also *h_node_enter_finding* below).

The enumeration of the states of a node follows traditional C conventions; i.e., the first state has index 0, the second state has index 1, etc. So, if *node* has $n$ states, then *state* must be a nonnegative integer smaller than $n$.

**Example 8.3** The following code

```
h_domain_t d = h_kb_load_domain ("mydomain.hkb", NULL);
h_node_t n = h_domain_get_node_by_name (d, "input");

h_node_select_state (n, 0);
...
```

loads a domain and enters the observation that node `input` is in state 0. ∎

If the evidence is not a simple instantiation, then the function *h_node_enter_finding* should be called, once for each state of the node, specifying the finding value for the state.

▸ **h_status_t *h_node_enter_finding***
(**h_node_t** *node*, **size_t** *state*, **h_number_t** *value*)

Specify *value* as the finding value for state *state* of *node* (which must be a discrete chance or decision[1] node); *value* must be nonnegative, and *state* must specify a valid state of *node*.

If you have several independent observations to be presented as likelihoods to HUGIN for the same node, you have to multiply them yourself; each call to *h_node_enter_finding* overrides the previous finding value stored for the indicated state. The *h_node_get_entered_finding*(113) function can be conveniently used for the accumulation of a set of likelihoods.

To specify evidence for a continuous node, the following function must be used.

▸ **h_status_t *h_node_enter_value*** (**h_node_t** *node*, **h_double_t** *value*)

Specify that the continuous node *node* has the value *value*.

Note that inference is not automatically performed when evidence is entered (not even when the domain is compiled). To get the updated beliefs, you must explicitly call a propagation function (see Section 9.2).

## 8.3 Retracting evidence

If an already entered observation is found to be invalid, it can be retracted by the following function.

▸ **h_status_t *h_node_retract_findings*** (**h_node_t** *node*)

Retract all findings for *node*. [If *node* is discrete, this is equivalent to setting the finding value to 1 for all states of *node*.]

▸ **h_status_t *h_domain_retract_findings*** (**h_domain_t** *domain*)

Retract findings for all nodes of *domain*. This is useful when, e.g., a new set of observations should be entered; see also *h_domain_initialize*(126).

In addition to *h_node_retract_findings*, *h_domain_retract_findings*, and *h_domain_initialize* (and deletion of domains and nodes, of course), the *h_node_set_number_of_states*(35) function deletes evidence when the number of states of a (discrete) node is changed.

---

[1] Because evidence for a decision node must always be an instantiation, it is usually better to use *h_node_select_state* to specify evidence for decisions.

**Example 8.4** The code

```
...
d = h_kb_load_domain ("mydomain.hkb", NULL);
n = h_domain_get_node_by_name (d, "input");
h_node_select_state (n, 0);
...
h_node_retract_findings (n);
```

enters the observation that the discrete node `input` is in state $0$; later, that observation is retracted, returning the node `input` to its initial status. ∎

## 8.4 Determining independence properties

Independence properties in belief networks are often determined using the concept of *d-separation* [7, 12, 15, 18, 26, 27, 31]:

> Given three sets of nodes: $A$, $B$, and $S$. A *trail* $\tau$ (a sequence of nodes that are connected by arcs, of any directionality) from a node $a \in A$ to a node $b \in B$ in an acyclic directed graph is said to be *blocked* by $S$, if there is a node $n \in \tau$ such that either
>
> - $n \in S$ and arcs of $\tau$ do not meet head-to-head at $n$, or
> - $n$ and all its descendants are not in $S$, and arcs of $\tau$ meet head-to-head at $n$.
>
> A trail that is not blocked by $S$ is said to be *active*.
>
> If all trails between $A$ and $B$ are blocked, then $A$ and $B$ are said to be *d-separated* given $S$. We denote this as $A \perp B | S$.

Notice that, in the above definition, the sets $A$, $B$, and $S$ are not assumed to be disjoint. Instead, we have, for $x \in S$ and any $y$, that $\{x\} \perp \{y\} | S$.

Intuitively, when we have beliefs computed conditional on evidence on some set $E$ of nodes, and we receive new evidence on a set $E'$ of nodes, then nodes $X$ such that $\{X\} \perp E' | E$ will not be affected by the new evidence (that is, after propagation of $E'$, the beliefs of $X$ will be unchanged).

Lauritzen *et al* [18] describes a different (but equivalent) criterion for determining the independence properties in belief networks.

> Given three sets of nodes: $A$, $B$, and $S$. Construct the induced subgraph consisting of $A \cup B \cup C$ and their ancestors, and then form the moral graph corresponding to this subgraph. If $S$ separates $A \setminus S$ and $B \setminus S$ in this moral graph, we have $A \perp B | S$.

This criterion forms the basis for the algorithm in the HUGIN API for determining independence properties.

The following functions determine the (maximal) sets of nodes that are *d*-connected to (respectively *d*-separated from) the specified source nodes given evidence nodes.

- ▸ **h_node_t** ∗***h_domain_get_d_connected_nodes***
  (**h_domain_t** *domain*, **h_node_t** ∗*source*, **h_node_t** ∗*hard*,
  **h_node_t** ∗*soft*)

- ▸ **h_node_t** ∗***h_domain_get_d_separated_nodes***
  (**h_domain_t** *domain*, **h_node_t** ∗*source*, **h_node_t** ∗*hard*,
  **h_node_t** ∗*soft*)

Return the maximal set of nodes that is *d*-connected to (respectively *d*-separated from) the *source* nodes given the evidence nodes.

The evidence nodes are specified using the *hard* and *soft* node lists: *hard* must contain the nodes with hard evidence, and *soft* must contain the nodes with soft evidence (see Section 8.1). These functions treat nodes with soft evidence as having no evidence themselves but instead as having (imaginary) instantiated child nodes — see Example 8.1.

In LIMIDs, for the purpose of performing *d*-separation analysis, links pointing at decision nodes are treated as follows: Links pointing at instantiated decision nodes are ignored, but links pointing at uninstantiated decision nodes are not. Also, the evidence specified in the *hard* and *soft* node lists must form a valid evidence scenario (see Section 8.1.3), and, moreover, adding the *source* nodes to the set of nodes with hard evidence must also yield a valid evidence scenario. (An "invalid evidence" error code is set if these constraints are not satisfied. The constraints also imply that only chance and decision nodes can appear in the *source*, *hard*, and *soft* node lists.)

The functions return their results in a permanently allocated node list stored inside *domain*, so the application must not free the list. The list is overwritten by each (successful) call to any of the functions.

If an error occurs, the functions return NULL.

## 8.5 Retrieving beliefs

When the domain has been compiled (see Chapter 6) and the evidence propagated (see Chapter 9), the calculated beliefs can be retrieved using the functions described below.

- ▸ **h_number_t** ***h_node_get_belief*** (**h_node_t** *node*, **size_t** *state*)

The belief in state *state* of the discrete node *node* is returned. If an error is detected (e.g., *state* is an invalid state, or *node* is not a discrete node), a negative number is returned.

Note that if evidence has been entered since the most recent propagation, the beliefs returned by this function may not be up-to-date.

The beliefs in the states of a node usually form a probability distribution for the node. However, other possibilities exist, determined by the propagation method used; see Chapter 9 for details.

**Example 8.5** A sample use of a domain could be

```
h_domain_t d;
h_node_t n;
int i, k;
...
n = h_domain_get_node_by_name (d, "input");
h_node_select_state (node, 0);
h_domain_propagate (d, h_equilibrium_sum, h_mode_normal);
n = h_domain_get_node_by_name (d, "output");
k = h_node_get_number_of_states (n);
for (i = 0; i < k; i++)
    printf ("P(output=%d|input=0) = %g\n", i,
             h_node_get_belief (n, i));
...
```

This code enters the observation that node `input` is in state 0. This observation is then propagated to the remaining nodes, using *h_domain_propagate*(119). Finally, the revised beliefs (the conditional probabilities given the observation) for node `output` are displayed. ∎

For continuous nodes, the beliefs computed take the form of the mean and variance of the distribution of the node given the evidence.

▸ **h_double_t** ***h_node_get_mean*** (**h_node_t** *node*)

Return the mean of the marginal distribution of the continuous node *node*.

▸ **h_double_t** ***h_node_get_variance*** (**h_node_t** *node*)

Return the variance of the marginal distribution of the continuous node *node*.

The marginal distribution of *node* is not necessarily a Gaussian distribution. In general, it is a mixture of several Gaussians. See the description of the *h_node_get_distribution*(110) function for instructions on how to access the individual components of the mixture.

Sometimes, the joint distribution over a set of nodes is desired:

▸ **h_table_t** ***h_domain_get_marginal***
(**h_domain_t** *domain*, **h_node_t** ∗*nodes*)

Compute the marginal table for the specified list *nodes* of nodes with respect to the (imaginary) joint potential, determined by the current potentials on the junction tree(s) of *domain*. The nodes must be distinct chance or decision nodes and must belong to *domain*.[2] If the *nodes* list contains continuous nodes, they must be last in the list. This operation is not allowed on compressed domains. If an error occurs, a NULL pointer is returned.

The fact that the marginal is computed based on the current junction tree potentials implies that the "equilibrium" and "evidence incorporation mode" (see Section 9.1) for the marginal will be as specified in the propagation that produced the current junction tree potentials.

If the *nodes* list contains continuous nodes, the marginal will in general be a so-called *weak marginal* [7, 16, 19]. This means that only the means and the (co)variances are computed, not the full distribution. In other words, the marginal is not necessarily a multi-variate normal distribution with the indicated means and (co)variances (in general, it is a mixture of such distributions). Also note that if the discrete probability is zero, then the mean and (co)variances are essentially random numbers (the inference engine doesn't bother computing zero components of a distribution).

The table returned by *h_domain_get_marginal* is owned by the application, and it is the responsibility of the application to deallocate it (using *h_table_delete*(63)) after use.

See Chapter 4 for information on how to manipulate **h_table_t** objects.

▸ **h_table_t** ***h_node_get_distribution*** (**h_node_t** *node*)

This function computes the distribution for the CG node *node*. No value must be propagated for *node*. If an error occurs, a NULL pointer is returned.

The distribution for a CG node is in general a mixture of several Gaussian distributions. What *h_node_get_distribution* really computes is a joint distribution for *node* and a set of discrete nodes. The set of discrete nodes is chosen such that the computed marginal is a *strong marginal* [7, 16, 19], but the set is not necessarily minimal.

As is the case for the *h_domain_get_marginal* function, the means and variances corresponding to zero probability components are arbitrary numbers.

The table returned by *h_node_get_distribution* is owned by the application, and it is the responsibility of the application to deallocate it (using *h_table_delete*(63)) after use.

**Example 8.6** The following code prints out the components that form the (mixture) distribution for a continuous node. Each component is a (one-dimensional) Gaussian distribution.

```
h_node_t n;
...
printf ("Distribution for %s:\n", h_node_get_name (n));
```

---

[2] In the current implementation, all nodes must belong to the same junction tree.

```
{
    h_table_t t = h_node_get_distribution (n);
    size_t k, s = h_table_get_size (t);
    h_number_t *p = h_table_get_data (t);

    for (k = 0; k < s; k++)
        if (p[k] > (h_number_t) 0)
            printf ("%g * Normal (%g, %g)\n",
                    (double) p[k],
                    (double) h_table_get_mean (t, k, n),
                (double) h_table_get_variance (t, k, n));

    (void) h_table_delete (t);
}
```

Note that we ignore the zero components of the distribution. (The table functions used are described in Chapter 4.) ∎

## 8.6 Retrieving expected utilities

In LIMIDs, we will want to retrieve the expected utilities associated with the states of a node (usually a decision node). We might also be interested in the overall expected utility of a decision problem.

First, evidence must be entered and propagated. Then, the functions below can be used to retrieve expected utilities.

▸ **h_number_t** ***h_node_get_expected_utility*** (**h_node_t** *node*, **size_t** *state*)

If *node* is a discrete decision or chance node, then the expected utility associated with state *state* is returned. If *node* is a utility node, then the contribution of *node* to the overall expected utility is returned (in this case, specify 0 as the value for *state*).

▸ **h_number_t** ***h_domain_get_expected_utility*** (**h_domain_t** *domain*)

The overall expected utility of the decision problem represented by *domain* is returned.

For both functions, a negative value is returned if an error occurs. But this is of little use for error detection, since any real value is (in general) a valid utility. Thus, errors must be checked for using the *h_error_code*(18) function.

## 8.7 Computing function values

The results of inference can be used as input to the functions associated with function nodes.

▸ **h_double_t** ***h_node_get_value*** (**h_node_t** *node*)

Evaluate the function associated with *node* (which must be a function node), and return the result.

If an error occurs, a negative number is returned. However, since a negative value can be a valid result, error conditions must, in general, be checked for using *h_error_code*(18) and related functions.

In order to successfully evaluate the function, the following conditions must be satisfied:

- *node* must have a model;
- all model nodes must be instantiated (as determined by the results of inference);
- the expression identified by the configuration of the model nodes must be valid:
  - the expression must be non-NULL,
  - all subexpressions must have correct types (i.e., the types of the operands must match the types expected by the operator), and
  - interval nodes must not appear in the expression;
- there must be no computation failures (e.g., division by zero).

If non-function nodes appear in the expression, their values are determined from the results of inference. In this case, the underlying domain must be compiled, and, moreover, the equilibrium must be 'sum,' and the evidence incorporation mode must be 'normal' (see Section 9.1). These conditions must also be satisfied, if the model has a non-empty set of model nodes. (The conditions are not required in all other cases.)

The values of non-function nodes are determined as follows:

- the value of a labeled node is the label associated with the state to which the node is instantiated (it is an error, if the node is not instantiated);
- the value of a boolean node is the truth value associated with the state to which the node is instantiated (it is an error, if the node is not instantiated);
- the value of a numbered or a CG node is the mean value of the node;
- the value of a utility node is the expected utility of the node.

If the expression refers to the values of other function nodes, then they must also be evaluated. This is done (recursively) according to the procedure described above.

It is also possible to evaluate function nodes using the results of simulation as input — see *h_node_get_sampled_value*(129).

## 8.8 Examining the evidence

The HUGIN API provides functions to access the evidence currently entered to the nodes of a domain. Functions to determine the type of evidence (nonvacuous or likelihood) are also provided.

The *node* argument of the functions described in this section must be a chance or a decision node. The functions having "*propagated*" in their names require the underlying domain to be compiled. The functions having "*entered*" in their names do not.

▸ **h_number_t** ***h_node_get_entered_finding*** (**h_node_t** *node*, **size_t** *state*)

Retrieve the finding value currently registered at the discrete node *node* for state *state*. If an error is detected, a negative value is returned.

▸ **h_number_t** ***h_node_get_propagated_finding***
(**h_node_t** *node*, **size_t** *state*)

Retrieve the finding value incorporated within the current junction tree potentials for state *state* of the discrete node *node*. If an error is detected, a negative value is returned.

▸ **h_double_t** ***h_node_get_entered_value*** (**h_node_t** *node*)

Retrieve the entered value for the continuous node *node*. If no value has been entered, a usage error code is set and a negative number is returned. However, since a negative number is a valid value for a continuous node, checking for errors must be done using *h_error_code*(18) and friends.

▸ **h_double_t** ***h_node_get_propagated_value*** (**h_node_t** *node*)

Retrieve the value that has been propagated for the continuous node *node*. If no value has been propagated, a usage error code is set and a negative number is returned. However, since a negative number is a valid value for a continuous node, checking for errors must be done using *h_error_code*(18) and friends.

▸ **h_boolean_t** ***h_node_evidence_is_entered*** (**h_node_t** *node*)

Is the evidence potential, currently registered at node *node* (which must be a chance or a decision node), non-vacuous?

▸ **h_boolean_t** ***h_node_likelihood_is_entered*** (**h_node_t** *node*)

Is the evidence potential, currently registered at node *node* (which must be a chance or a decision node), a likelihood?

▸ **h_boolean_t** ***h_node_evidence_is_propagated*** (**h_node_t** *node*)

Is the evidence potential for node *node* (which must be a chance or a decision node), incorporated within the current junction tree potentials, nonvacuous?

▸ **h_boolean_t** ***h_node_likelihood_is_propagated*** (**h_node_t** *node*)

Is the evidence potential for node *node* (which must be a chance or a decision node), incorporated within the current junction tree potentials, a likelihood?

## 8.9 Case files

When evidence has been entered to a set of nodes, it can be saved to a file. Such a file is known as a *case file*. The HUGIN API provides functions for reading and writing case files.

A case file is a text file. The format (i.e., syntax) of a case file can be described by the following grammar.

⟨Case file⟩ → ⟨Node finding⟩*
⟨Node finding⟩ → ⟨Node name⟩ `:` ⟨Value⟩
⟨Value⟩ → ⟨State index⟩ | ⟨Likelihood⟩
| ⟨Label⟩ | ⟨Real number⟩ | `true` | `false`
⟨State index⟩ → `#`⟨Integer⟩
⟨Likelihood⟩ → `(` ⟨Nonnegative real number⟩* `)`

where:

- ⟨State index⟩ is a valid specification for any discrete (chance or decision) node. The index is interpreted as if specified as the last argument to *h_node_select_state*(105) for the named node.
- ⟨Likelihood⟩ is also a valid specification for all discrete nodes. A nonnegative real number must be specified for each state of the named node (and at least one of the numbers must be positive).
- ⟨Real number⟩ is a valid specification for CG, numbered, and interval nodes. For numbered and interval nodes, the acceptable values are defined by the state values of the named node.
- ⟨Label⟩ is a valid specification for labeled nodes. The label (a double-quoted string) must match a unique state label of the named node. If the contents of the string conform to the definition of a name (see Section 12.7), the quotes can be omitted.

- `true` and `false` are valid specifications for boolean nodes.

Comments can be included in the file. Comments are specified using the `%` character and extends to the end of the line. Comments are ignored by the case file parser.

**Example 8.7** The following case file demonstrates the different ways to specify evidence: `A`, `B`, and `C` are labeled nodes with states `yes` and `no`; `D` is a boolean node; `E` is a numbered node; `F` is an interval node; and `G` is a CG node.

```
A: "yes"
B: #1          % equivalent to "no"
C: (.3 1.2)    % likelihood
D: true
E: 2
F: 3.5
G: -1.4
```

Because `yes` is a valid name, the finding for `A` can instead be specified simply as:

```
A: yes
```

∎

▸ **h_status_t** ***h_domain_save_case***
(**h_domain_t** *domain*, **h_string_t** *file_name*)

Create a case file named *file_name*. (Note: If a file named *file_name* already exists and is not write-protected, it is overwritten.) The case file will contain the evidence currently entered in *domain*. The contents is text conforming to the above described format.

Note that if (some of) the nodes with evidence have not been assigned names, then names will automatically be assigned (through calls to the *h_node_get_name*(39) function).

▸ **h_status_t** ***h_domain_parse_case***
(**h_domain_t** *domain*, **h_string_t** *file_name*,
**void** (∗*error_handler*) (**h_location_t**, **h_string_t**, **void** ∗),
**void** ∗*data*)

This function parses the case stored in the file with name *file_name*. The evidence stored in the case is entered into *domain*. All existing evidence in *domain* is retracted before entering the new evidence.

The *error_handler* and *data* arguments are used for error handling. This is similar to the error handling done by the other parse functions. See Section 12.8 for further information.

In case of errors, no evidence will be entered.

# Chapter 9

# Inference

When evidence has been entered and the domain has been compiled, we want to compute revised beliefs for the nodes of the domain. This process is called *inference*. In HUGIN, this is done by a two-pass propagation operation on the junction tree(s). The two passes are known as *CollectEvidence* and *DistributeEvidence*, respectively. The *CollectEvidence* operation proceeds inwards from the leaves of the junction tree to a root clique, which has been selected in advance. The *DistributeEvidence* operation proceeds outwards from the root to the leaves.

This inference scheme is described in many places. See, for example, [7, 8, 11, 12, 13, 16].

## 9.1 Propagation methods

The collect/distribute propagation scheme can be used to compute many different kinds of information.

### 9.1.1 Summation and maximization

One can think of a propagation as the computation of certain marginals of the full joint probability distribution over all variables. As is well-known, the distribution of an individual variable can be found by summing/integrating out all other variables of this joint probability distribution.

However, we might also be interested in the probability, for each state of a given variable, of the most probable configuration of all other variables. Again, we can compute these probabilities from the joint probability distribution over all variables. But this time, we “max out” the other variables (i.e., we take the maximum value over the set of relevant configurations).

It turns out that both kinds of marginals can be computed by the collect/distribute propagation method by a simple parametrization of the marginalization method.

When a propagation has been successfully completed, we have a situation where the potentials on the cliques and the separators of the junction tree are *consistent*, meaning that the marginal on a set S of variables can be computed from the potential of any clique or separator containing S. We also say that we have established *equilibrium* on the junction tree. The equilibria, discussed above, are called *sum-equilibrium* and *max-equilibrium*, respectively.

The HUGIN API introduces an enumeration type to represent the equilibrium. This type is called **h_equilibrium_t**. The values of this type are denoted by *h_equilibrium_sum* and *h_equilibrium_max*.

### 9.1.2 Evidence incorporation mode

The traditional way to incorporate (discrete) evidence into a junction tree is to first multiply each evidence potential onto the potential of some clique; when this has been done, the actual propagation is performed. This mode of evidence incorporation is called the *normal* mode.

An alternative way to incorporate evidence is to multiply the evidence potentials onto the clique potentials *during* the propagation. If this is done in the correct places, the equilibrium achieved will have the following property. Assuming a sum-propagation, the resulting potential on a set V of variables (clique, separator, or a single variable) will be the marginal probability for V given evidence on all variables *except* the variables in V itself. Since this is similar to the retraction of evidence (and accompanying propagation) for each variable, this mode is known as the *fast-retraction* mode of evidence incorporation.

A fast-retraction propagation can be useful to identify suspicious findings. If the observation made on a variable has a very small probability in the probability distribution obtained by incorporation of evidence on the other variables, then quite likely something is wrong with the observation. (Another way to identify suspicious findings is to use the notion of *conflict*; see Section 9.4).

If each item of evidence is a single-state observation of a single variable, then the equilibrium achieved with a normal mode propagation will give no useful information about the observed variables. In such cases, it would be tempting to always choose a fast-retraction propagation. However, one should be aware of the following facts: (1) a fast-retraction propagation may fail due to logical relations in the domain model; (2) fast-retraction propagations are not available for compressed domains, domains with continuous variables, or LIMIDs.

The fast-retraction propagation method is described in [6].

The HUGIN API introduces an enumeration type to represent the evidence incorporation mode. This type is called **h_evidence_mode_t**. The values of this type are denoted by *h_mode_normal* and *h_mode_fast_retraction*.

## 9.2 Propagation

Using the HUGIN API for inference involves several steps:

(1) Specify the evidence.

(2) Perform inference (i.e., update the junction tree potentials using the specified evidence, and compute derived results such as beliefs and expected utilities for all relevant nodes).

(3) Retrieve the results of inference.

The first and third steps are described in Chapter 8. The second step is performed using the following function.

▸ **h_status_t** ***h_domain_propagate***
(**h_domain_t** *domain*, **h_equilibrium_t** *equilibrium*,
**h_evidence_mode_t** *evidence_mode*)

Establish the specified *equilibrium* on all junction trees of *domain* using *evidence_mode* for incorporation of evidence. Revised beliefs are computed for all (chance and decision) nodes. And if *domain* is a LIMID, revised expected utilities are also computed for all (chance, decision, and utility) nodes.

If *domain* is a LIMID, then the specified evidence must form a *valid evidence scenario* — see Section 8.1.3.

If *domain* is compressed, *evidence_mode* must be *h_mode_normal*. Unless all nodes of *domain* are discrete chance nodes, then *equilibrium* must be *h_equilibrium_sum* and *evidence_mode* must be *h_mode_normal*.

The propagation operation requires a compiled domain (see Chapter 6).

If models (see Chapter 5) are used to specify tables, and the tables are not up-to-date with respect to the models, then the tables are regenerated (using *h_node_generate_table*$^{(83)}$).

If some tables (either because of direct modification or because of regeneration from a model) have changed since the most recent propagation operation, the equilibrium is computed (from scratch) using the updated tables. In this case, conditional probability and policy tables are normalized (so that the conditional probabilities corresponding to a given parent state configuration sum to 1). Note that the initial propagation (i.e., the one performed by the compilation operation) considers all tables to be "new."

In general, *h_domain_propagate* does not perform any unnecessary work. For example, if the current evidence is a superset of the evidence propagated in the most recent propagation (and the equilibrium and evidence incorporation mode of that propagation was equal to *equilibrium* and *evidence_mode*, and the set of node tables is unchanged), then an incremental propagation is performed — using *d*-separation analysis (see Section 8.4) to reduce the extent of the *DistributeEvidence* pass.

However, if some currently propagated evidence has been retracted (and the current evidence incorporation mode is 'normal'), then the new equilibrium must be computed either from scratch using the node tables or using a memory backup of the junction tree tables (see *h_domain_save_to_memory*$^{(125)}$). Using a memory backup can significantly speed up the propagation in this case.

If an error is detected during the propagation, the initial (that is, with no evidence incorporated) distribution is established. This might fail, and if it does, subsequent requests for beliefs and other quantities computed by a propagation will fail.

The set of evidence is never changed by any of the propagation functions.

If the propagation fails, the error code (in addition to the general error conditions such as 'usage' and 'out-of-memory') can be:

***h_error_inconsistency_or_underflow*** Some probability potential has degenerated into a zero-potential (i.e., with all values equal to zero). This is almost always due to evidence that is considered "impossible" (i.e., its probability is zero) by the domain model. (In theory, it can also be caused by underflow in the propagation process, but this rarely happens in practice.)

***h_error_overflow*** Overflow (caused by operations the purpose of which was to avoid underflow) has occurred in the propagation process. This is an unlikely error (but it is not entirely impossible — it can occur in very large "naive Bayes models," for example). The error never occurs if only one item of evidence is propagated at a time.

***h_error_fast_retraction*** A fast-retraction propagation has failed due to logical relations within the domain model.

***h_error_invalid_evidence*** (This only applies to LIMIDs.) The evidence scenario is invalid — see Section 8.1.3.

It is also possible to perform inference on individual junction trees:

▸ **h_status_t** ***h_jt_propagate***
(**h_junction_tree_t** *jt*, **h_equilibrium_t** *equilibrium*,
**h_evidence_mode_t** *evidence_mode*)

The meanings of the arguments and return value are similar to the meanings of the arguments and return value of the *h_domain_propagate*(119) function.

## 9.3 Inference in LIMIDs: Computing optimal policies

The propagation operation described above computes beliefs and expected utilities based on the tables specified as part of the domain model: the conditional probability tables (associated with the chance nodes), the utility tables (associated with the utility nodes), and the policy tables (associated with the uninstantiated decision nodes). Notice that the propagation operation doesn't change the policy tables — it simply uses the policy tables in the computations.

However, the pre-specified policy tables might not be optimal — or, in other words, changing the policy tables might improve the overall expected utility of the decision problem.

The HUGIN API provides a function to compute policy tables that maximize the overall expected utility. This function uses the SPU (Single Policy Updating) algorithm [20] to compute the policies. This is an iterative algorithm that updates one policy at a time and terminates when all policies have converged (i.e., more iterations change nothing). The algorithm usually finds the globally optimal policies, but it is possible that the algorithm may get stuck at a local maximum.

The parents specified for the decision nodes determine which observations should be taken into account when decisions are made. Ideally, we would specify all observations to be taken into account, but this may not be practical because the size of a policy table is exponential in the number of parents. We therefore often don't specify less important observations as parents of decision nodes (for example, old observations are typically less important than new ones) in order to reduce the size of the policy tables.

Unless all relevant information has been specified as parents, then it can be useful to recompute policies whenever new information becomes available. This is because the computations take all existing observations (in addition to future observations specified as parents of decision nodes) into account when policies are computed.

▸ **h_status_t** ***h_domain_update_policies*** (**h_domain_t** *domain*)

This function updates (using the SPU algorithm) the policies of uninstantiated decisions in *domain* such that the overall expected utility of the decision problem is maximized (but note that this maximum is not guaranteed to be a global maximum — it is possible that it is a local maximum). Evidence is taken into account (so the updated policies are effectively conditioned on the evidence).

The junction tree potentials must be up-to-date with respect to the evidence, the node tables (i.e., the CPTs, the current policies, and the utility tables) and their models (if any). The equilibrium must be 'sum,' and the evidence incorporation mode must be 'normal' (see Section 9.1).

The function only performs updates yielding improvements of the overall expected utility. This implies that a policy is only changed for parent configurations matching the observations.

As the final step, the function performs a propagation using the new policies: The beliefs of nodes that depend on uninstantiated decision nodes, the expected utilities of all nodes as well as the overall expected utility are updated.

## 9.4 Conflict of evidence

An alternative to a fast-retraction propagation to identify suspicious findings is to use the concept of *conflict of evidence*.

Given $n$ items of evidence, $e_1, \ldots, e_n$, we define the conflict of this set of findings as

$$\frac{P(e_1) \times \cdots \times P(e_n)}{P(e_1, \ldots, e_n)}$$

This quantity can be computed as part of the propagation process with virtually no overhead.

More details can be found in [10]. Note that in the paper, the definition of conflict of evidence includes the logarithm of the above ratio.

The current implementation of the HUGIN API does not support calculation of conflict of evidence in LIMIDs and when evidence has been specified for CG nodes.

The steps required for computing conflict of evidence are similar to those for performing inference:

(1) Specify the evidence.

(2) Make sure the inference engine is in the initial state (i.e., sum-equilibrium with no evidence incorporated). This can be accomplished using *h_domain_reset_inference_engine*$^{(126)}$.

(3) Perform a sum-propagation (Section 9.2). This computes the conflict of evidence.

(4) Retrieve the conflict of evidence.

The conflict of evidence is retrieved using the following function.

▸ **h_double_t** ***h_domain_get_conflict*** (**h_domain_t** *domain*)

Return the conflict of evidence for *domain* computed during the most recent propagation. If no evidence has been propagated, 1 is returned. In case of errors, a negative number is returned.

The conflict of evidence for a domain is the product of the conflict of evidence for the individual junction trees of the domain. The following function returns the conflict of evidence for a single junction tree.

▸ **h_double_t** ***h_jt_get_conflict*** (**h_junction_tree_t** *jt*)

Return the conflict of evidence for junction tree *jt* computed during the most recent propagation. If no evidence has been propagated, 1 is returned. In case of errors, a negative number is returned.

**Example 9.1** The following code outlines the proper way of computing conflicts.

```
h_domain_t d;
...
/* enter evidence */
h_domain_reset_inference_engine (d);
h_domain_propagate (d, h_equilibrium_sum, h_mode_normal);
printf ("Conflict of evidence: %g\n",
         h_domain_get_conflict (d));
```

This code first enters evidence into the domain, then ensures that the inference engine is in the initial state. Next, a propagation is performed. After the propagation, the conflict of evidence is retrieved and printed. ∎

## 9.5 The normalization constant

The final step of the collection phase of a propagation operation is to normalize the root clique potential.[1] For a sum-propagation, the normalization constant $\mu$ of this potential is the probability of the propagated evidence $\mathcal{E}$:

$$\mu = \mathrm{P}(\mathcal{E})$$

---

[1] In order to avoid underflow, local normalizations are performed in the separators as part of *CollectEvidence*. The normalization constant $\mu$ also includes the constants used in the local normalizations.

For a max-propagation, the normalization constant $\mu$ is the probability of the most probable configuration $x^{\mathcal{E}}$ consistent with the propagated evidence $\mathcal{E}$:

$$\mu = P(x^{\mathcal{E}})$$

This information is useful in many applications, so the HUGIN API provides functions to access the normalization constant and its logarithm:

▸ **h_double_t** ***h_domain_get_normalization_constant***
(**h_domain_t** *domain*)

▸ **h_double_t** ***h_domain_get_log_normalization_constant***
(**h_domain_t** *domain*)

If an error occurs, then *h_domain_get_normalization_constant* returns a negative number, and *h_domain_get_log_normalization_constant* returns a positive number.

The values returned by these functions are the normalization and the log-normalization constants for all of the propagated evidence. If the evidence consists of several items of evidence, it makes no difference whether all items are propagated together, or whether each item is entered and propagated in an incremental fashion.

If the evidence is "impossible" (that is, the evidence has zero probability), the propagation operation fails. This implies that the true normalization constant is always positive.[2] However, if many findings are propagated, the normalization constant can become very small. If the normalization constant becomes smaller than the smallest positive floating-point number representable within the **h_double_t** type, it underflows and *h_domain_get_normalization_constant* returns 0. In this case, *h_domain_get_log_normalization_constant* can be used (this function returns the correct value for all successful propagations).[3]

If approximation is used, the normalization constant should be compared to the error introduced by the approximation process — see *h_domain_get_approximation_constant*(98). If the probability of the evidence is smaller than the approximation error, propagation within the original (unapproximated) model should be considered (in order to get more accurate answers). See also Section 6.7.

If likelihood evidence has been propagated, the normalization constant cannot, in general, be interpreted as a probability. As an example, consider a binary variable: The likelihoods $\langle \frac{1}{2}, 1 \rangle$ and $\langle 1, 2 \rangle$ yield the same beliefs, but *not* the same normalization constant. However, see Example 8.1 and

---

[2] After a failed propagation operation, *h_domain_get_normalization_constant* returns 1.

[3] If the log-normalization constant cannot be correctly computed, the propagation operation fails. But the inference engine tries very hard to avoid that.

Example 8.2 for cases where it makes sense to interpret the normalization constant as a probability.

If CG evidence has been propagated, then the normalization constant is proportional to the density at the observed values of the continuous nodes (the proportionality constant is the conditional probability of the discrete evidence given the CG evidence). The density depends directly on the scale chosen for the continuous variables: Suppose that the scale of some continuous variable is changed from centimeter [cm] to millimeter [mm]. This causes the density values for that variable to be reduced by a factor of 10. Hence, the normalization constant should only be used to compare different sets of findings.

## 9.6 Initializing the inference engine

As described in Section 9.2, the propagation operation does not, in general, perform any unnecessary work: If possible, an incremental propagation is performed. However, if evidence has been changed or retracted, then this is not always possible. In this case, the inference engine must establish junction tree potentials equivalent to the initial state of the inference engine (i.e., with no evidence incorporated). This can be done by initializing the inference engine from the node potentials (i.e., the conditional probability, the policy, and the utility potentials).

As an alternative, a copy (often referred to as a "memory backup") of the initial state of the inference engine can be created. This memory backup is then subsequently used for initializing the inference engine for the cases where the initial state is needed. Using a memory backup can significantly speed up inference.

▸ **h_status_t** ***h_domain_save_to_memory*** (**h_domain_t** *domain*)

Create a memory backup of the belief and junction tree potentials of *domain*. (This approximately doubles the memory consumption of *domain*.) The equilibrium of the junction tree potentials must be 'sum' (see Section 9.1) with no evidence incorporated.

The operation is not supported for LIMIDs (because, in a LIMID, the initial distribution is generally not useful as a starting point for an incremental propagation).

The memory backup is kept up-to-date with respect to the node potentials: If a node potential changes (either because of direct modification or because of regeneration from a model), the memory backup is automatically refreshed. On the other hand, the memory backup is deleted as part of the "uncompilation" and compression operations. Thus, it is necessary to recreate the memory backup after a compilation or a compression operation.

The *h_domain_save_to_memory* function is typically called immediately after a compilation (assuming there is no evidence) or a reset-inference-engine operation.

▶ **h_status_t** ***h_domain_reset_inference_engine*** (**h_domain_t** *domain*)

Establish the initial state of the inference engine of *domain*: sum-equilibrium with no evidence incorporated.

This operation does not change the evidence (if any) specified for the nodes of *domain*, but the beliefs of the nodes computed by the operation do not take this evidence into account (that is, the beliefs are not conditioned on the evidence).

▶ **h_status_t** ***h_domain_initialize*** (**h_domain_t** *domain*)

Retract all evidence specified for the nodes of *domain* and establish the initial state of the inference engine.

This operation is equivalent to a *h_domain_retract_findings*(106) operation followed by either a *h_domain_reset_inference_engine*(126) operation or a sum-propagation.

**Example 9.2** The following code loads a domain and then repeatedly performs some experiment on the domain, using the application-defined function *perform_experiment*, until some condition is satisfied. The domain is initialized before each experiment starts, so that each experiment is carried out with the inference engine in the initial state.

```
h_domain_t d = h_kb_load_domain ("mydomain.hkb", NULL);
int done = 0;

while (!done)
{
    h_domain_initialize (d);
    done = perform_experiment (d);
}
...
```

∎

## 9.7 Querying the state of the inference engine

The HUGIN API provides several functions that enables the application to determine the exact state of the inference engine. The following queries can be answered:

- Which type of equilibrium is the junction tree(s) currently in?
- Which evidence incorporation mode was used to obtain the equilibrium?

- Has evidence been propagated?
- Has any likelihood evidence been propagated?
- Is there any unpropagated evidence?

▸ **h_boolean_t** ***h_domain_equilibrium_is***
(**h_domain_t** *domain*, **h_equilibrium_t** *equilibrium*)

Test if the equilibrium of all junction trees of *domain* is *equilibrium*.

▸ **h_boolean_t** ***h_jt_equilibrium_is***
(**h_junction_tree_t** *jt*, **h_equilibrium_t** *equilibrium*)

Test if the equilibrium of junction tree *jt* is *equilibrium*.

▸ **h_boolean_t** ***h_domain_evidence_mode_is***
(**h_domain_t** *domain*, **h_evidence_mode_t** *mode*)

Test if the equilibrium of all junction trees of *domain* could have been obtained through a propagation using *mode* as the evidence incorporation mode.

Note that without evidence, there is no difference between the equilibria produced via 'normal' or 'fast-retraction' propagations.

▸ **h_boolean_t** ***h_jt_evidence_mode_is***
(**h_junction_tree_t** *jt*, **h_evidence_mode_t** *mode*)

Test if the equilibrium of junction tree *jt* could have been obtained through a propagation using *mode* as the evidence incorporation mode.

▸ **h_boolean_t** ***h_domain_evidence_is_propagated*** (**h_domain_t** *domain*)

Test if evidence has been propagated for *domain*.

▸ **h_boolean_t** ***h_jt_evidence_is_propagated*** (**h_junction_tree_t** *jt*)

Test if evidence has been propagated for junction tree *jt*.

▸ **h_boolean_t** ***h_domain_likelihood_is_propagated***
(**h_domain_t** *domain*)

Test if likelihood evidence has been propagated for *domain*.

▸ **h_boolean_t** ***h_jt_likelihood_is_propagated*** (**h_junction_tree_t** *jt*)

Test if likelihood evidence has been propagated for junction tree *jt*.

▸ **h_boolean_t** ***h_domain_cg_evidence_is_propagated***
(**h_domain_t** *domain*)

Test if CG evidence has been propagated for *domain*.

▶ **h_boolean_t** ***h_jt_cg_evidence_is_propagated*** **(h_junction_tree_t** *jt*)

Test if CG evidence has been propagated for junction tree *jt*.

▶ **h_boolean_t** ***h_domain_evidence_to_propagate*** **(h_domain_t** *domain*)

Test if there is any node with changed evidence compared to the most recent propagation (if any). If there was no previous propagation, this is equivalent to testing if there is any node in *domain* with evidence.

▶ **h_boolean_t** ***h_jt_evidence_to_propagate*** **(h_junction_tree_t** *jt*)

Test if there is any node in the junction tree *jt* with new evidence as compared to the evidence propagated.

▶ **h_boolean_t** ***h_node_evidence_to_propagate*** **(h_node_t** *node*)

Is the entered and propagated evidence for *node* (which must be a chance or a decision node) different?

▶ **h_boolean_t** ***h_domain_tables_to_propagate*** **(h_domain_t** *domain*)

Are there any nodes in *domain* having a (conditional probability, policy, or utility) table that has changed since the most recent compilation or propagation operation?

▶ **h_boolean_t** ***h_jt_tables_to_propagate*** **(h_junction_tree_t** *jt*)

Similar to *h_domain_tables_to_propagate*, but specific to the junction tree *jt*.

## 9.8 Simulation

Often, we are interested in generating (sampling) configurations (i.e., vectors of values over the set of variables in the network) with respect to the conditional distribution given the evidence.

▶ **h_status_t** ***h_domain_simulate*** **(h_domain_t** *domain*)

If *domain* is compiled, sample a configuration with respect to the current distribution on the junction tree(s). This distribution must be in sum-equilibrium with evidence incorporated in 'normal' mode. Only propagated evidence is taken into account: Unpropagated evidence, and models and tables that have changed since the most recent propagation operation are ignored.

If *domain* is not compiled, sample a configuration with respect to the distribution determined by the tables associated with the nodes of *domain*. All uninstantiated chance and decision nodes must have valid tables (see Section 2.6), and the set of nodes with evidence must form an *ancestral set* of instantiated nodes (i.e., no likelihood or multi-state evidence, and if a chance node is instantiated, so are all of its parents). Tables that are not up-to-date with respect to their models (see Chapter 5) are *not* regenerated.

The sampled configuration is obtained using the following functions.[4]

▶ **h_index_t** ***h_node_get_sampled_state*** (**h_node_t** *node*)

Retrieve the state index of *node* (which must be a discrete node) within the configuration generated by the most recent call to *h_domain_simulate*. If an error occurs, a negative number is returned.

▶ **h_double_t** ***h_node_get_sampled_value*** (**h_node_t** *node*)

If *node* is a continuous node, then the value of *node* within the configuration generated by the most recent call to *h_domain_simulate* is returned.

If *node* is a function node, the function associated with *node* is evaluated using the configuration generated by *h_domain_simulate* as input (that is, if the function refers to a parent in an expression, then the sampled value of that parent is used in the evaluation), and the result is returned.

If an error occurs, a negative number is returned. However, since negative numbers can be valid in both cases, error conditions must be checked for using *h_error_code*(18) and related functions.

▶ **h_number_t** ***h_node_get_sampled_utility*** (**h_node_t** *node*)

Return the utility value in the potential table of *node* (which must be a utility node) identified by the configuration generated by the most recent call to *h_domain_simulate*. If an error occurs, a negative number is returned. But since negative numbers are valid utility values, errors must be checked for using *h_error_code*(18) and related functions.

The configurations generated by *h_domain_simulate* are not really random. They are generated using a pseudorandom number generator producing a sequence of numbers that although it appears random is actually completely deterministic. To change the starting point for the generator, use the following function.

▶ **h_status_t** ***h_domain_seed_random***
(**h_domain_t** *domain*, **unsigned int** *seed*)

Seed the pseudorandom number generator used by *h_domain_simulate* with *seed*.

The pseudorandom number generator implemented in Hugin can also be used directly through the following functions.

▶ **h_double_t** ***h_domain_get_uniform_deviate*** (**h_domain_t** *domain*)

Use the pseudorandom number generator for *domain* to sample a real number from the uniform distribution over the interval $[0, 1)$.

---

[4] In order to avoid returning invalid values, simulation results are automatically invalidated when an "uncompile" operation is performed. It is an error to request values derived from invalid simulation results.

▸ **h_double_t** ***h_domain_get_normal_deviate***
(**h_domain_t** *domain*, **h_double_t** *mean*, **h_double_t** *variance*)

Use the pseudorandom number generator for *domain* to sample a real number from the normal (also known as the Gaussian) distribution with mean *mean* and variance *variance*.

## 9.9 Value of information analysis

Consider the situation where a decision maker has to make a decision based on the probability distribution of a hypothesis variable. It could, for instance, be a physician deciding on a treatment of a patient given the probability distribution of a disease variable. For instance, if the probability of the patient suffering from the disease is above a certain threshold, then the patient should be treated immediately. Prior to deciding on a treatment, the physician may have the option to gather additional information about the patient such as performing a test or asking a certain question. Given a range of options, which option should the physician choose next? That is, which of the given options will produce the most information? These questions can be answered by a value of information analysis.

Given a Bayesian network model and a hypothesis variable, the task is to identify the variable which is most informative with respect to the hypothesis variable.

### Entropy and Mutual Information

The main reason for acquiring additional information is to reduce the uncertainty about the hypothesis under consideration. The selection of the variable to observe next (for example, the question to ask next) can be based on the notion of entropy. Entropy is a measure of how much the probability mass is scattered around on the states of a variable (the degree of chaos in the distribution of the variable). In other words, entropy is a measure of randomness. The more random a variable is, the higher its entropy will be.

The *entropy* $H(X)$ of a discrete random variable $X$ is defined as follows:

$$H(X) = -\sum_{x} p(x) \log p(x)$$

The maximum entropy $\log n$ (assuming $X$ has $n$ states) is achieved when the probability distribution of $X$ is uniform, while the minimum entropy $0$ is achieved when all the probability mass is located on a single state. Thus, $0 \leq H(X) \leq \log n$.

Since entropy can be used as a measure of the uncertainty in the distribution of a variable, we can determine how the entropy of a variable changes as observations are made. In particular, we can identify the most informative observation.

If $Y$ is a random variable, then the *conditional entropy* $H(X|Y)$ of $X$ given $Y$ is computed as follows:

$$H(X|Y) = \sum_{y} p(y) H(X|Y{=}y) = H(X) - I(X;Y)$$

where

$$I(X;Y) = \sum_{x,y} p(x,y) \log \frac{p(x,y)}{p(x)p(y)}$$

is the *mutual information* (also known as the *cross entropy*) of $X$ and $Y$. The conditional entropy $H(X|Y)$ is a measure of the uncertainty of $X$ given an observation of $Y$, while the mutual information $I(X;Y)$ is a measure of the information shared by $X$ and $Y$ (i.e., the reduction in the entropy of $X$ obtained by observing $Y$). If $X$ is the variable of interest, then $I(X;Y)$ is a measure of the value of observing $Y$. It follows that the variable $Y$ to observe next should be chosen such that $I(X;Y)$ is maximized.

The HUGIN API provides functions for computing entropy and mutual information for discrete nodes. These functions use the natural logarithm as the “log” function in the computations. The computations are done with respect to the (joint) distribution computed by the most recent propagation operation (implying that the underlying domain must be compiled). This distribution must have been obtained by a sum-propagation using ‘normal’ evidence incorporation mode.

▸ **h_double_t** ***h_node_get_entropy*** (**h_node_t** *node*)

Compute the entropy of *node* (which must be a discrete node).

▸ **h_double_t** ***h_node_get_mutual_information***
(**h_node_t** *node*, **h_node_t** *other*)

Compute the mutual information of *node* and *other* (which must be discrete nodes). The nodes must belong to the same domain, and this domain must be compiled but not compressed.

## 9.10 Sensitivity analysis

Often there are one or more nodes (and associated beliefs) in a belief network that are regarded as “outputs” of the model. For example, the probability (belief) of a disease in a medical domain model.

In order to improve the “robustness” of the belief network model, it should be investigated how sensitive the output probabilities are to variations in the numerical parameters of the model (because these numerical parameters are often imprecisely specified). The most influential parameters should be identified, and effort should be directed towards reducing the impreciseness of those parameters.

The process of identifying the most influential parameters of a belief network model and analyzing their effects on the output probabilities of the model is known as *sensitivity analysis* [4, 5].

Let $A$ be a (discrete chance) node in a belief network model being subjected to sensitivity analysis, and let $a$ be a state of $A$. For given evidence $\mathcal{E}$, the probability $P(A=a|\mathcal{E})$ can be considered as a function of the conditional probabilities in the CPTs of the chance nodes in the network. If the network is a LIMID, $P(A=a|\mathcal{E})$ can also be considered as a function of the conditional probabilities in the policies of the decision nodes.

Let $B$ be a (discrete) node in the belief network, and let $x$ be a conditional probability parameter associated with $B$ — we shall refer to $x$ as an *input parameter*. This parameter corresponds to some state $b$ of $B$ and some state configuration $\pi$ of the parents of $B$.

We wish to express $P(A=a|\mathcal{E})$ as a function of $x$. When $x$ varies, the other conditional probabilities associated with $B$ for parent configuration $\pi$ must also vary (in order to satisfy the "sum-to-1" constraint). The most common assumption is that the other conditional probabilities vary according to a proportional scheme:

$$P(b'|\pi) = \begin{cases} x & \text{if } b' = b \\ \dfrac{1-x}{1-\theta_{b|\pi}}\theta_{b'|\pi} & \text{if } b' \neq b \end{cases}$$

Here, $\theta_{b|\pi}$ is the initial (user-specified) value of $x$ (and similarly for the other input parameters).

Under this assumption, it can be shown that the probability of the evidence $\mathcal{E}$ is a linear function of $x$:

$$P(\mathcal{E})(x) = \gamma x + \delta \tag{9.1}$$

Likewise, the joint probability of the evidence $\mathcal{E}$ and the event "$A=a$" is a linear function of $x$:

$$P(A=a, \mathcal{E})(x) = \alpha x + \beta \tag{9.2}$$

We then conclude:

$$P(A=a|\mathcal{E})(x) = \frac{\alpha x + \beta}{\gamma x + \delta} \tag{9.3}$$

This function is known as a *sensitivity function*.

If we know the constants (i.e., $\alpha$, $\beta$, $\gamma$, and $\delta$) of this function, we can determine whether our estimate for $x$ is precise enough: For example, we might know that $x$ belongs to some interval, and if the variations of $P(A = a \mid \mathcal{E})(x)$ for $x$ ranging over that interval are acceptable, then we accept the estimate.

It turns out that the $\langle\alpha, \beta, \gamma, \delta\rangle$-constants can be computed for all input parameters using only two propagations. The most efficient way to organize the computations is therefore to provide two functions: One function to perform the propagations and to store the results of these propagations, and another function to request the $\langle\alpha, \beta, \gamma, \delta\rangle$-constants of a specific sensitivity function (i.e., corresponding to a specific input parameter).

▸ **h_status_t** ***h_node_compute_sensitivity_data***
(**h_node_t** *node*, **size_t** *state*)

This function computes the $\langle\alpha, \beta, \gamma, \delta\rangle$-constants (or rather, compact data from which these constants can be efficiently computed) of all sensitivity functions (i.e., for all input parameters $x$) of the output probability

$$P(\mathit{node} = \mathit{state} \mid \mathcal{E})(x),$$

where *node* is a discrete chance node in a compiled domain, and $\mathcal{E}$ is the current evidence. No evidence must be specified for *node*.

The junction tree potentials must be up-to-date with respect to the evidence, the node tables (i.e., the CPTs, the policies, and the utility tables) and their models (if any). The equilibrium must be 'sum,' and the evidence incorporation mode must be 'normal' (see Section 9.1).

After calling the above function for producing sensitivity data, the following function is called for each input parameter to be analyzed.

▸ **h_status_t** ***h_node_get_sensitivity_constants***
(**h_node_t** *node*, **size_t** *input*, **h_number_t** $*\alpha$, **h_number_t** $*\beta$,
**h_number_t** $*\gamma$, **h_number_t** $*\delta$)

This function retrieves the $\langle\alpha, \beta, \gamma, \delta\rangle$-constants of the sensitivity function corresponding to the specified input parameter: *node* must be a discrete node, and *input* must be the index of an entry of the table of *node*. The output probability of the sensitivity function was specified in the preceding call to *h_node_compute_sensitivity_data*(133).

The $\langle\alpha, \beta, \gamma, \delta\rangle$-constants of the sensitivity function are returned in the **h_number_t** $*$ arguments.

The user-specified value of the input parameter must not be 0 or 1 (because the method used for computing the $\langle\alpha, \beta, \gamma, \delta\rangle$-constants does not work in these cases).

This function uses the sensitivity data produced by the preceding call to *h_node_compute_sensitivity_data*. Certain operations on the underlying domain invalidate these data: An operation that "uncompiles" (Section 6.5) the domain also invalidates the sensitivity data. Also, changing node tables (i.e., the CPTs, the policies, and the utility tables) or their models (if any) invalidate the sensitivity data.

If these conditions are not satisfied, the function returns an error.

Note that the $\langle\alpha, \beta, \gamma, \delta\rangle$-constants returned by this function correctly define the sensitivity function in (9.3), but the constants do not define the probabilities in (9.1) and (9.2) correctly (unless $\mathcal{E}$ is empty). One reason is that the constants are scaled to avoid floating-point underflow: The scaling factor is $P(\mathcal{E})^{-1}$ (assuming the user-specified values for all input parameters). Also, if $\alpha = p\gamma$ and $\beta = p\delta$ for some constant $p$, then the function might return $\alpha = \gamma = 0$ and $\beta = p$ and $\delta = 1$. This is the case if the specified input parameter is not associated with a node in the sensitivity set (see below).

**Example 9.3** This example uses the "chest clinic" network [21]. We shall determine the probability of lung cancer (L) given shortness of breath (D) as a function of the prior probability of the patient being a smoker (S).

```
h_domain_t d = h_kb_load_domain ("asia.hkb", NULL);
h_number_t alpha, beta, gamma, delta;
h_node_t L, D, S;
... /* code to find the nodes */
h_domain_compile (d);
h_node_select_state (D, 0);  /* set D to "yes" */
h_domain_propagate (d, h_equilibrium_sum, h_mode_normal);
h_node_compute_sensitivity_data (L, 0);
h_node_get_sensitivity_constants
       (S, 0, &alpha, &beta, &gamma, &delta);
printf ("P(L=yes|D=yes)(x)"
        " = (%g * x + %g) / (%g * x + %g)\n",
        alpha, beta, gamma, delta);
```

This code produces the following output (reformatted to fit the width of the page):

```
P(L=yes|D=yes)(x) = (0.170654 * x + 0.0174324)
                      / (0.535988 * x + 0.732006)
```

In this output, $x$ refers to the prior probability of the patient being a smoker. ∎

In many cases, only a small subset of the input parameters can influence the output probability $P(A = a|\mathcal{E})$. The set of nodes to which these input parameters are associated is known as the *sensitivity set* for $A$ given $\mathcal{E}$ [5]. This set of nodes can be identified using *d*-separation analyses (Section 8.4): Suppose that we add an extra parent $X$ to node $B$. If $X$ is *d*-connected to $A$ given $\mathcal{E}$, then $B$ belongs to the sensitivity set.

The *h_node_compute_sensitivity_data*(133) function identifies the sensitivity set for the specified output probability given the available evidence. The following function can be used to retrieve the sensitivity set.

▸ **h_node_t** ∗***h_domain_get_sensitivity_set*** (**h_domain_t** *domain*)

Return the sensitivity set computed by the most recent call to *h_node_compute_sensitivity_data*, provided that the results of that call are still valid — see *h_node_get_sensitivity_constants*(133) for the list of conditions that invalidate the sensitivity data.

The sensitivity set is returned as a list of nodes. This list is stored inside *domain*, so the application must not attempt to deallocate it. Also note that the list is invalidated by a subsequent call to *h_node_compute_sensitivity_data* (i.e., it must be retrieved again, because it may have been reallocated).

Approximated domains (this typically includes compressed domains) are treated specially. Because approximation can be seen as entering special evidence in all cliques, we conservatively assume that all input parameters may influence the specified output probability. In other words, the sensitivity set is taken to be the set of all nodes in the domain (regardless of what the *d*-separation criterion tells us).

See Section 6.6 and Section 6.7 for more information regarding compression and approximation.

### Parameter tuning

In order to tune the behavior of a belief network, we often need to impose constraints on the output probabilities of the network.

We recall the equation for the output probability $P(A = a|\mathcal{E})$ as a function of some input parameter $x$:

$$P(A = a|\mathcal{E})(x) = \frac{\alpha x + \beta}{\gamma x + \delta}.$$

Using this equation, we can find the values of $x$ that satisfy a given constraint. In [4], the following types of constraints are considered:

(1) $P(A = a|\mathcal{E}) - P(A' = a'|\mathcal{E}) \geq \varepsilon$ (DIFFERENCE)

(2) $P(A = a|\mathcal{E}) \,/\, P(A' = a'|\mathcal{E}) \geq \varepsilon$ (RATIO)

(3) $P(A = a|\mathcal{E}) \geq \varepsilon$ (SIMPLE)

where $P(A = a|\mathcal{E})$ and $P(A' = a'|\mathcal{E})$ are output probabilities (referring to the same variable or to two different variables). The relation can also be '$\leq$'.

If $x$ is associated with a node belonging to the sensitivity sets of both output probabilities, the sensitivity functions (as computed by *h_node_compute_sensitivity_data*(133) or *h_domain_compute_sensitivity_data*(136)) are:

$$P(A = a|\mathcal{E})(x) = \frac{\alpha_1 x + \beta_1}{\gamma x + \delta} \qquad \text{and} \qquad P(A' = a'|\mathcal{E})(x) = \frac{\alpha_2 x + \beta_2}{\gamma x + \delta}$$

For the DIFFERENCE constraint, we get (because $\gamma x + \delta$ is nonnegative):

$$P(A = a | \mathcal{E}) - P(A' = a' | \mathcal{E}) \geq \varepsilon \quad \Longleftrightarrow \quad (\alpha_1 - \alpha_2 - \gamma\varepsilon)x \geq -\beta_1 + \beta_2 + \delta\varepsilon$$

If $\alpha_1 - \alpha_2 - \gamma\varepsilon$ is positive, we get a lower bound for $x$. If $\alpha_1 - \alpha_2 - \gamma\varepsilon$ is negative, we get an upper bound.

If $x$ is not associated with a node belonging to the sensitivity sets of both output probabilities, then at least one of the probabilities is a constant. If both are constants, the solution is trivial. If only one is a constant, the DIFFERENCE constraint reduces to a SIMPLE constraint.

For the RATIO constraint, we get:

$$P(A = a | \mathcal{E}) \,/\, P(A' = a' | \mathcal{E}) \geq \varepsilon \quad \Longleftrightarrow \quad (\alpha_1 - \alpha_2\varepsilon)x \geq -\beta_1 + \beta_2\varepsilon$$

Depending on the sign of $\alpha_1 - \alpha_2\varepsilon$, we get either a lower or an upper bound.

Finally, for the SIMPLE constraint, we get:

$$P(A = a | \mathcal{E}) \geq \varepsilon \quad \Longleftrightarrow \quad (\alpha_1 - \gamma\varepsilon)x \geq -\beta_1 + \delta\varepsilon$$

Again, we get either a lower or an upper bound.

This can be generalized to any number of constraints (by forming the intersection of solution intervals) and any type of "solvable" constraint. Solving a set of parameter constraints is done in two stages.

The first stage consists of computing sensitivity data corresponding to all of the output probabilities involved in the constraints. Let the output probabilities be $P(A_0 = a_0 | \mathcal{E})$, $P(A_1 = a_1 | \mathcal{E})$, ..., $P(A_{n-1} = a_{n-1} | \mathcal{E})$. These probabilities are encoded using two lists: a NULL-terminated list of nodes and a list of state values, such that the state value at position $i$ in the second list is associated with the node at position $i$ in the first list. A given output probability is then later referred to by its position in these lists.

The second stage computes the solution to the constraints.

▸ **h_status_t** ***h_domain_compute_sensitivity_data***
(**h_domain_t** *domain*, **h_node_t** ∗*nodes*, **const size_t** ∗*states*)

Compute sensitivity data for all of the specified output probabilities. All of the *nodes* must belong to *domain*. The usage conditions of *h_node_compute_sensitivity_data*(133) must be satisfied for all output probabilities.

Note: *h_node_compute_sensitivity_data* is implemented in terms of this function (implying that each function overwrites the sensitivity data computed by the other function).

▸ **h_status_t** ***h_node_get_sensitivity_constants_by_output***
(**h_node_t** *node*, **size_t** *input*, **size_t** *output*, **h_number_t** $*\alpha$,
**h_number_t** $*\beta$, **h_number_t** $*\gamma$, **h_number_t** $*\delta$)

This function retrieves the $\langle \alpha, \beta, \gamma, \delta \rangle$-constants of the sensitivity function corresponding to the specified input parameter and the specified output probability: *node* must be a discrete node, *input* must be the index of an entry of the table of *node*, and *output* must identify an output probability specified in the preceding call to *h_domain_compute_sensitivity_data*(136).

See *h_node_get_sensitivity_constants*(133) for the remaining usage conditions and the semantics of this function.

Note: *h_node_get_sensitivity_constants* is implemented in terms of this function.

▶ **h_node_t** ∗***h_domain_get_sensitivity_set_by_output***
(**h_domain_t** *domain*, **size_t** *output*)

Return the sensitivity set corresponding to the specified output probability: *output* refers to one of the output probabilities specified in the most recent call to *h_domain_compute_sensitivity_data*(136).

See also *h_domain_get_sensitivity_set*(135).

Note: *h_domain_get_sensitivity_set* is implemented in terms of this function.

## 9.11 Most probable configurations

Steffen Lauritzen has proposed a Monte Carlo algorithm for finding the most probable configurations of a set of discrete nodes given evidence on some of the remaining nodes.

Let $A$ be the set of nodes for which we seek the most probable configurations. The goal is to identify all configurations with probability at least $p_{\text{min}}$. The algorithm can be outlined as follows.

- A sequence of configurations of $A$ is sampled from the junction tree. [Configurations of $A$ are obtained by ignoring the sampled values of the remaining nodes.] Repetitions are discarded.

- The probability of each (unique) configuration in the sequence is computed. Let $p_{\text{total}}$ be the total probability of all (unique) configurations. If $1 - p_{\text{total}} < p_{\text{min}}$, the algorithm terminates.

The most probable configuration is also known as the *maximum a posteriori* (MAP) configuration. The above algorithm thus solves a more general form of the MAP configuration problem.

The basic assumption of the algorithm is that most of the probability mass is represented by a small set of configurations. If this is not the case, the algorithm could run for a long time.

If the size of $A$ is small, it is probably more efficient to compute the joint probability table of $A$ using *h_domain_get_marginal*(109). From this table, it is easy to identify the most probable configurations.

▸ **h_status_t** ***h_domain_find_map_configurations***
(**h_domain_t** *domain*, **h_node_t** $*$*nodes*, **h_double_t** $p_{\min}$)

Find all configurations of *nodes* with probability at least $p_{\min}$ ($0 < p_{\min} \leq 1$). The *nodes* list must be a nonempty list of distinct discrete nodes belonging to *domain*. The domain must be compiled, and the distribution on the junction tree(s) must be in sum-equilibrium with evidence incorporated in 'normal' mode (see Section 9.1).

The current implementation imposes the following additional restrictions.[5]

- LIMIDs are not supported (that is, *domain* must not contain decisions).
- Evidence must not be specified for any of the nodes in *nodes*.
- The junction tree potentials must be up-to-date with respect to the evidence, the CPTs and their models (if any).

The probability $p_{\min}$ should be "reasonable" (like $0.01$ or higher). Otherwise, the algorithm could run for a long time.

The results of a call to *h_domain_find_map_configurations* are retrieved using the functions described below. The results remain available until *domain* is uncompiled.

▸ **h_count_t** ***h_domain_get_number_of_map_configurations***
(**h_domain_t** *domain*)

This function returns the number of configurations found by the most recent successful call to *h_domain_find_map_configurations* (with *domain* as argument). If no such call has been made (or the results of the call are no longer available), a negative number is returned.

Let $n$ be the number of configurations with probability at least $p_{\min}$ — as specified in the call to *h_domain_find_map_configurations*. The configurations are identified by integer indexes $0, \ldots, n-1$, where index $0$ identifies the most probable configuration, index $1$ identifies the second-most probable configuration, ..., and index $n-1$ identifies the least probable of the $n$ configurations.

▸ **size_t** $*$***h_domain_get_map_configuration***
(**h_domain_t** *domain*, **size_t** *index*)

This function returns the configuration identified by *index* among the configurations with probability at least $p_{\min}$ — as specified in the most recent successful call to *h_domain_find_map_configurations* (as explained above). If an error occurs, NULL is returned.

---

[5] An alternative implementation is under consideration. This implementation might impose a different set of restrictions (e.g., requiring that *domain* not being compressed, but removing the other additional restrictions).

A configuration is stored as a list of state indexes (an array of **size_t** values): The $k$th element of a configuration is a state index for the $k$th node in the *nodes* list — as specified in the call to *h_domain_find_map_configurations*.

The configuration is stored within the *domain* data structure, so the application must not deallocate or modify the configuration. The HUGIN API deallocates the configurations when *domain* is uncompiled.

▶ **h_double_t *h_domain_get_probability_of_map_configuration***
(**h_domain_t** *domain*, **size_t** *index*)

Return the probability of the configuration returned by *h_domain_get_map_configuration* (*domain*, *index*). If an error occurs, a negative number is returned.

# Chapter 10

# Sequential Updating of Conditional Probability Tables

This chapter describes the facilities for using data to sequentially update the conditional probability tables for a domain when the graphical structure and an initial specification of the conditional probability distributions have been given in advance.

Sequential updating makes it possible to update and improve these conditional probability distributions as observations are made. This is especially important if the model is incomplete, the modeled domain is drifting over time, or the model quite simply does not reflect the modeled domain properly.

The sequential learning method implemented (also referred to as *adaptation*) was developed by Spiegelhalter and Lauritzen [30]. See also Cowell *et al* [7] and Olesen *et al* [24].

Spiegelhalter and Lauritzen introduced the notion of *experience*. The experience is the quantitative memory which can be based both on quantitative expert judgment and past cases. *Dissemination of experience* refers to the process of calculating prior conditional distributions for the variables in the belief network. *Retrieval of experience* refers to the process of calculating updated distributions for the parameters that determine the conditional distributions for the variables in the belief network.

## 10.1 Experience counts and fading factors

The adaptation algorithm updates the conditional probability distributions of a belief network in the light of inserted and propagated evidence. Adaptation can be applied to discrete chance variables only.

The experience for a given node is represented as a set of experience counts $\alpha_0, \ldots, \alpha_{n-1}$ — one count for each configuration of the (discrete) parents of the node: $\alpha_i$ represents the number of times the (discrete) parents have been observed to be in the $i$th configuration. However, note that the "counts" don't have to be integers (non-integer counts can arise because of incomplete case data).

The experience counts are stored in a table:

▸ **h_table_t *h_node_get_experience_table*** (**h_node_t** *node*)

This function returns the experience table of *node* (which must be a chance node).[1] If *node* doesn't have an experience table, then one will be created. The nodes of the experience table are the discrete parents of *node*, and the order of the nodes in the experience table is the same as the order in the conditional probability table of *node*.

When an experience table is created, it is filled with zeros. Zero is an invalid experience count for the adaptation algorithm, so positive values must be stored in the table before adaptation can take place. If a database of cases is available, the EM algorithm can be used to get initial experience counts.

The adaptation algorithm only adapts conditional distributions corresponding to parent configurations with positive experience counts. All other configurations (including all configurations for nodes with no experience table) are ignored. This convention can be used to turn on/off adaptation at the level of individual parent configurations: Setting an experience count to a positive number turns on adaptation for the associated parent configuration; setting the experience count to zero or a negative number turns it off.

Note that the table returned by *h_node_get_experience_table* is the table stored within *node* (and not a copy of that table). This implies that the experience counts for *node* can be modified using functions that provide access to the internal data structures of tables (see Chapter 4).

Experience tables can be deleted using the *h_table_delete*<sup>(63)</sup> function. Deleting an experience table turns off adaptation for the node associated with the table.

▸ **h_boolean_t *h_node_has_experience_table*** (**h_node_t** *node*)

This function tests whether *node* has an experience table.

The adaptation algorithm also provides an optional *fading* feature. This feature reduces the influence of past (and possibly outdated) experience in order to let the domain model adapt to changing environments. This is done by discounting the experience count $\alpha_i$ by a *fading factor* $\lambda_i$, which is

---

[1] Although adaptation is only possible for discrete chance nodes, experience tables are also used to control the EM algorithm, and the EM algorithm applies to both discrete and continuous chance nodes.

a positive real number less than but typically close to 1. The true fading amount is made proportional to the probability of the parent configuration in question. To be precise: If the probability of the $i$th parent configuration given the propagated evidence is $p_i$, then $\alpha_i$ is multiplied by $(1 - p_i) + p_i\lambda_i$ *before* adaptation takes place. Note that experience counts corresponding to parent configurations that are inconsistent with the propagated evidence (i.e., configurations with $p_i = 0$) remain unchanged.

The fading factor $\lambda_i$ can be set to 1: this implies that cases are accumulated (that is, no fading takes place). Setting $\lambda_i$ to a value greater than 1 or less than or equal to 0 will disable adaptation for the $i$th parent configuration (just as setting $\alpha_i$ to an invalid value will).

The fading factors are also stored in a table:

▸ **h_table_t** ***h_node_get_fading_table*** (**h_node_t** *node*)

This function returns the fading table of *node* (which must be a discrete chance node). If *node* doesn't have a fading table, then one will be created. The order of the nodes in the fading table is the same as the order of the parents of *node* in the conditional probability table of *node* (which is identical to the order of nodes in the experience table).

When a fading table is created, all entries are initially set to 1. This has the same affect as if no fading table were present. To get fading, values between 0 and 1 must be stored in the table.

The table returned by *h_node_get_fading_table* is the table stored within *node* (and not a copy of that table). This implies that the fading factors for *node* can be modified using functions that provide access to the internal data structures of tables (see Chapter 4).

Like experience tables, fading tables can also be deleted using the *h_table_delete*(63) function. Note that fading tables are *not* automatically deleted when the corresponding experience tables are deleted. The fading tables must be explicitly deleted.

▸ **h_boolean_t** ***h_node_has_fading_table*** (**h_node_t** *node*)

This function tests whether *node* has a fading table.

**Example 10.1** The following code loads the `Asia` domain [21], enables adaptation for all nodes except `E` (we delete the experience table of `E`, if it has one): If some node (besides `E`) doesn't have an experience table, we create one and set all entries of the table to 10.

```
h_domain_t d = h_kb_load_domain ("Asia.hkb", NULL);
h_node_t E = h_domain_get_node_by_name (d, "E");
h_node_t n = h_domain_get_first_node (d);

for (; n != 0; n = h_node_get_next (n))
```

```
    if (n != E && !h_node_has_experience_table (n))
    {
        h_table_t t = h_node_get_experience_table (n);
        h_number_t *data = h_table_get_data (t);
        size_t k = h_table_get_size (t);

        for (; k > 0; k--, data++)
            *data = 10.0;
    }

if (h_node_has_experience_table (E))
    h_table_delete (h_node_get_experience_table (E));
```

■

## 10.2 Updating tables

When experience tables (and optionally fading tables) have been created and their contents specified, then the model is ready for adaptation.

An adaptation step consists of entering evidence, propagating it, and, finally, updating (adapting) the conditional probability and experience tables. The last substep is performed using the following function.

▸ **h_status_t** ***h_domain_adapt*** (**h_domain_t** domain)

This function updates (adapts), for all discrete chance nodes of *domain*, the experience count (retrieval of experience) and the conditional probability distribution (dissemination of experience) for all parent configurations having a valid experience count and a valid fading factor.

This adaptation is based on the propagated evidence (hence *domain* must be compiled). The evidence must have been propagated with equilibrium equal to ‘sum’ and evidence-incorporation-mode equal to ‘normal’ (Section 9.1). Moreover, the junction tree potentials must be up-to-date with respect to the node tables and their models (the *h_domain_tables_to_propagate*(128) function tests this condition). Note that the latter condition implies that the *h_domain_adapt* function cannot be (successfully) called before the updated distributions have been incorporated into the junction tree potentials (by either a propagation or a reset-inference-engine operation).

**Example 10.2** The following code

```
h_domain_t d = h_kb_load_domain ("Asia.hkb", NULL);
h_node_t n = h_domain_get_node_by_name (d, "S");

h_node_select_state (n, 0);

h_domain_propagate (d, h_equilibrium_sum, h_mode_normal);
```

```
  h_domain_adapt (d);
  ...
```

loads the `Asia` domain [21], enters the observation that node `S` is in state 0 (“yes”), propagates the evidence, and updates the experience and conditional probability tables of the domain (using *h_domain_adapt*). We assume that suitable experience (and possibly fading) tables have already been specified. ∎

If the experience count for some parent configuration is (or can be expected to be) very large ($10^4$ or more) or the fading factor is very close to 1 ($1-10^{-4}$ or closer), then it is recommended that a double-precision version of the HUGIN API is used.

# Chapter 11

# Learning Network Structure and Conditional Probability Tables

Chapter 10 describes the facilities for adapting the conditional probability distributions of a domain as new observations are made. This is known as *sequential* learning.

However, the sequential learning method requires that a complete belief network model including an initial assessment of the conditional probabilities is given. This chapter describes the HUGIN API facilities for using data (a set of cases) to learn the network structure as well as the conditional probability distributions of a belief network model. This is known as *batch* learning. Batch learning requires that all data is available when the learning process starts, whereas sequential learning allows the knowledge to be updated incrementally.

The method for learning the network structure is the *PC algorithm*, developed by Spirtes and Glymour [31]. A similar algorithm (the *IC algorithm*) was independently developed by Verma and Pearl [27].

The method for learning the conditional probability distributions (a method based on the *EM algorithm*) was developed by Lauritzen [17]. See also Cowell *et al* [7].

## 11.1 Data

An assignment of values to some or all of the nodes of a domain is called a *case*. If values have been assigned to all nodes, the case is said to be *complete*; otherwise, it is said to be *incomplete*. The data used by the learning procedure is comprised of a set of cases.

Note that the mechanism for entering cases described in this section is intended for case sets that fit in main memory. The learning algorithms currently provided by the HUGIN API assume that the data is stored in main memory. Also note that case data is not saved as part of the HUGIN KB file produced by *h_domain_save_as_kb*⁽⁴⁵⁾.

The cases are numbered sequentially from $0$ to $N - 1$, where $N$ is the total number of cases. The first case gets the number $0$, the second case gets the number $1$, etc.

▶ **h_status_t** ***h_domain_set_number_of_cases***
(**h_domain_t** *domain*, **size_t** *count*)

This function adjusts the storage capacity for cases of *domain* to *count*. Let the current capacity be $m$. The contents of the cases numbered $0$ up to $\min(count, m) - 1$ are unchanged. If $count > m$, then the contents of the cases numbered $m$ to $count - 1$ are set to 'unknown'. If $count < m$, then the cases numbered $count$ to $m - 1$ are deleted. In particular, setting *count* to $0$ deletes all cases.

The following function is provided for convenience (e.g., for use when reading a file of cases where the number of cases is not known in advance).

▶ **h_index_t** ***h_domain_new_case*** (**h_domain_t** *domain*)

Allocate storage within *domain* to hold a new case. If successful, the function returns the index of the new case. If an error occurs, a negative number is returned.

▶ **h_count_t** ***h_domain_get_number_of_cases*** (**h_domain_t** *domain*)

Returns the number of cases currently allocated for *domain*.

▶ **h_status_t** ***h_node_set_case_state***
(**h_node_t** *node*, **size_t** *case_index*, **size_t** *state_index*)

Specify the state value of the discrete node *node* associated with case *case_index* to be *state_index*; *state_index* must be an integer identifying a state of *node* (similar to the last argument of the function *h_node_select_state*⁽¹⁰⁵⁾). If the number of states of *node* is subsequently decreased (such that *state_index* becomes invalid), then the learning algorithms will treat the data as unknown/missing.

In order to reduce memory consumption, the value of *state_index* must be less than or equal to 32767 (that is, a signed 2-byte quantity is allocated for each case state index).

▶ **h_index_t** ***h_node_get_case_state*** (**h_node_t** *node*, **size_t** *case_index*)

Retrieve the state value of the discrete node *node* associated with case *case_index*. If an error occurs or no state value (or an invalid state value) has been specified, a negative number is returned.

Case data for continuous nodes is specified using the following functions.

▸ **h_status_t** ***h_node_set_case_value***
(**h_node_t** *node*, **size_t** *case_index*, **h_double_t** *value*)

Set the value associated with the continuous node *node* in case *case_index* to *value*.

▸ **h_double_t** ***h_node_get_case_value*** (**h_node_t** *node*, **size_t** *case_index*)

Retrieve the value of the continuous node *node* associated with case *case_index*. If an error occurs, a negative number is returned, but this cannot be used for error detection, since any real value is a valid value. Instead, the *h_error_code*(18) function must be used.

The next two functions apply to both discrete and continuous nodes.

▸ **h_status_t** ***h_node_unset_case*** (**h_node_t** *node*, **size_t** *case_index*)

Specify that the value of *node* in case *case_index* is 'unknown'.

Note that this function should rarely be needed, since the state values for all nodes of a newly created case are 'unknown', and also the value of a newly created node will be 'unknown' in all cases.

▸ **h_boolean_t** ***h_node_case_is_set*** (**h_node_t** *node*, **size_t** *case_index*)

Test whether the value of *node* in case *case_index* is currently set.

In large data sets, some cases may appear more than once. Instead of entering the case each time it appears, a count may be associated with the case. This count must be nonnegative (a zero case-count means that the case will be ignored by the learning algorithms), but it doesn't have to be an integer.

▸ **h_status_t** ***h_domain_set_case_count***
(**h_domain_t** *domain*, **size_t** *case_index*, **h_number_t** *case_count*)

Set the case-count for the case with index *case_index* in *domain* to *case_count*.

▸ **h_number_t** ***h_domain_get_case_count***
(**h_domain_t** *domain*, **size_t** *case_index*)

Retrieve the case-count associated case *case_index* in *domain*.

If no case-count has been associated with a case, the count defaults to 1.

The case-counts have no influence on the value returned by *h_domain_get_number_of_cases*(148).

The learning algorithms operate on the data set as a whole. But sometimes it can be useful to perform inference (or some other task) using a specific case in the data set. The following function assists with such tasks.

▸ **h_status_t** ***h_domain_enter_case***
(**h_domain_t** *domain*, **size_t** *case_index*)

Enter the case data from case *case_index* as evidence into *domain*. All existing evidence in *domain* is retracted before entering the case.

## 11.2 Scoring of graphical models

When we learn a graphical model from a set of cases, we want the model that best describes the data. We want to express this "goodness" using a single number, so that we can easily compare different models. We call such a number a *score*.

Several different scoring measures have been proposed. The HUGIN API provides the following scores:

- The log-likelihood of the data given the model. This is simply the sum of the log-likelihoods of the individual cases.
- Akaike's Information Criterion (AIC): This is computed as $l - \kappa$, where $l$ is the log-likelihood and $\kappa$ is the number of free parameters in the model.
- The Jeffreys–Schwarz criterion, also called the Bayesian Information Criterion (BIC): This is computed as $l - \frac{1}{2}\kappa \log n$, where $l$ and $\kappa$ are defined as above, and $n$ is the number of cases.

The log-likelihood score doesn't take model complexity into account, whereas the AIC and BIC scores do.

The following functions assume that case data has been specified, that *domain* is compiled (because inference will be performed), and that the junction tree potentials are up-to-date with respect to the node tables and their models (if any). If *domain* is a LIMID, then each case must specify a valid evidence scenario (see Section 8.1.3).

▸ **h_double_t** ***h_domain_get_log_likelihood*** (**h_domain_t** *domain*)

Get the log-likelihood of the case data with respect to the graphical model of *domain*.[1] This is computed using the current conditional probability tables. If this function is called immediately after the EM algorithm has been run (for example, using *h_domain_learn_tables*$^{(157)}$), the log-likelihood will be

---

[1] Prior to version 6.5 of the HUGIN API, this function could only be used after the EM algorithm had run, and the function returned the log-likelihood computed with respect to the conditional probability tables *before* the final iteration of the EM algorithm — i.e., *not* the final tables.

computed with respect to the final tables computed by the EM algorithm. But the function can also be used without the EM algorithm.

▶ **h_double_t** ***h_domain_get_AIC*** (**h_domain_t** *domain*)

Get the AIC score of the case data with respect to the graphical model of *domain*.

▶ **h_double_t** ***h_domain_get_BIC*** (**h_domain_t** *domain*)

Get the BIC score of the case data with respect to the graphical model of *domain*.

## 11.3 Data files

When a set of cases has been entered as described in the previous section, it can be saved to a file. Such a file is known as a *data file*. The HUGIN API provides functions for reading and writing data files.

A data file is a text file. The format (i.e., syntax) of a data file can be described by the following grammar.

⟨Data file⟩ → ⟨Header⟩⟨Case⟩$^*$
⟨Header⟩ → `#`⟨Separator⟩⟨Node list⟩ | ⟨Node list⟩
⟨Separator⟩ → `,` | ⟨Empty⟩
⟨Node list⟩ → ⟨Node name⟩ | ⟨Node list⟩⟨Separator⟩⟨Node name⟩
⟨Case⟩ → ⟨Case count⟩⟨Separator⟩⟨Data list⟩ | ⟨Data list⟩
⟨Case count⟩ → ⟨Nonnegative real number⟩
⟨Data list⟩ → ⟨Data⟩ | ⟨Data list⟩⟨Separator⟩⟨Data⟩
⟨Data⟩ → ⟨Value⟩ | `*` | `?` | ⟨Empty⟩

where:

- The ⟨Header⟩ must occupy a single line in the file. Likewise, each ⟨Case⟩ must occupy a single line.
- If `#` is the first element of ⟨Header⟩, then each ⟨Case⟩ must include a ⟨Case count⟩.
- Each ⟨Case⟩ must contain a ⟨Data⟩ item for each node specified in the ⟨Header⟩. The $i$th ⟨Data⟩ item (if it is a ⟨Value⟩) in the ⟨Data list⟩ must be valid (as explained in Section 8.9) for the $i$th node in the ⟨Node list⟩ of the ⟨Header⟩.
- If ⟨Data⟩ is `*`, `?`, or ⟨Empty⟩, then the data is taken as 'missing'.

- If ⟨Separator⟩ is ⟨Empty⟩, then none of the separated items is allowed to be ⟨Empty⟩.
- ⟨Value⟩ is as defined in Section 8.9, except that the ⟨Likelihood⟩ alternative is not allowed.

Comments can be included in the file. Comments are specified using the `%` character and extends to the end of the line. Comments behave like newline characters. Empty lines (after removal of blanks, tabs, and comments) are ignored by the data file parser (i.e., they do not represent "empty" cases).

**Example 11.1** Here is a small set of cases for the `Asia` domain [21].

```
#       A       S       D       X

1       "yes"   "no"    "no"    "no"
1       "no"    "yes"   "yes"   "no"
1       "no"    "yes"   "yes"   "yes"
1       "no"    "no"    "yes"   "yes"
2       "yes"   "yes"   *       "no"
1       "yes"   "no"    "no"    *
1       "yes"   "yes"   "yes"   "yes"
1       "no"    "no"    "no"    *
```

The first line lists the nodes, and the remaining lines each describe a case. The first case corresponds to a non-smoking patient, that has been to Asia recently, does not have shortness of breath, and the X-ray doesn't show anything. The last case corresponds to a non-smoking patient, that has not (recently) been to Asia, does not have shortness of breath, and the X-ray is not available. Similarly for the other cases.

Note the extra (optional) initial column of numbers: These numbers are case counts. The number 2 for the fifth case indicates that this case has been observed twice; the other cases have only been observed once. The presence of case counts is indicated by the # character in the header line. ∎

Note the distinction between case files (Section 8.9) and data files: A case file contains exactly one case, it may contain likelihood data, and reading a case file means loading the case data as evidence. A data file, on the other hand, can contain arbitrarily many cases, but likelihood data is not allowed, and reading a data file (using *h_domain_parse_cases* described below) loads the case data using the facilities described in Section 11.1.

▸ **h_status_t** ***h_domain_parse_cases***
(**h_domain_t** *domain*, **h_string_t** *file_name*,
**void** (∗*error_handler*) (**h_location_t**, **h_string_t**, **void** ∗),
**void** ∗*data*)

This function parses the cases stored in the file with name *file_name*. The cases will be stored in *domain* using the facilities described in Section 11.1. Existing cases in *domain* are not modified.

The *error_handler* and *data* arguments are used for error handling. This is similar to the error handling done by the other parse functions. See Section 12.8 for further information.

If an error occurs, no cases will be added to *domain*.

The *h_domain_save_cases* function saves case data stored in a domain.

▶ **h_status_t** ***h_domain_save_cases***
(**h_domain_t** *domain*, **h_string_t** *file_name*, **h_node_t** ∗*nodes*,
**h_index_t** ∗*cases*, **h_boolean_t** *include_case_counts*,
**h_string_t** *separator*, **h_string_t** *missing_data*)

Save (some of) the case data stored in *domain* as a data file named *file_name*. (Note: If a file named *file_name* already exists and is not write-protected, it is overwritten.) The format and contents of the file are controlled by several arguments:

***nodes*** is a non-empty NULL-terminated list of (distinct) nodes. Moreover, all *nodes* must be chance or decision nodes belonging to *domain*. The list specifies which nodes (and their order) that are saved.

Note: If (some of) the *nodes* do not have names, they will be assigned names (through calls to the *h_node_get_name*(39) function).

***cases*** is a list of case indexes (which must all be valid), terminated by −1. The list specifies which cases (and their order in the file) that must be included. Duplicates are allowed (the case will be output for each occurrence of its index in the list).

When a case is output, the associated case count is output unmodified: If the case has case count $n$, then it is also $n$ in the generated file (not $n/2$ or something like that).

NULL can be passed for *cases*. This will cause all cases to be output (in the same order as stored in *domain*).

***include_case_counts*** is a boolean controlling the presence of case counts in the generated file:

- If it is *true*, case counts will be included (even if they are all 1).
- If it is *false*, they are not included. This is only allowed if all the selected cases have integer-valued case counts — because, instead of writing the case count to the file, the case is repeated as many times as specified by the case count. Note: If the case count is zero, the case will be omitted from the file.

***separator*** is a string that is output between node names in the header and between data items (and after the case count) in a case. If the generated file is to be read by *h_domain_parse_cases*, then *separator* must be non-empty, must contain at most one comma, and the remaining characters must be blanks or tabs.[2]

***missing_data*** is a string that is output if no data is specified for a given node in a given case. If the generated file is to be read by *h_domain_parse_cases*, then the following restrictions apply: *missing_data* must contain at most one of the `*` or `?` characters; if *missing_data* does *not* contain one of these characters, then *separator* must contain a comma; the remaining characters must be blanks or tabs.[2]

**Example 11.2** Let `Asia.dat` be a data file with contents as shown in Example 11.1. The following code loads the `Asia` domain [21], parses the `Asia.dat` data file, and generates a new data file (`New.dat`) containing a subset of the data. [See Example 12.24 for an appropriate definition of the *error_handler* used by the parse function.]

```
h_domain_t d = h_kb_load_domain ("Asia.hkb", NULL);
h_index_t cases[5] = { 0, 2, 4, 6, -1 };
h_node_t nodes[4];

nodes[0] = h_domain_get_node_by_name (d, "S");
nodes[1] = h_domain_get_node_by_name (d, "D");
nodes[2] = h_domain_get_node_by_name (d, "X");
nodes[3] = NULL;

h_domain_parse_cases
        (d, "Asia.dat", error_handler, "Asia.dat");

h_domain_save_cases
        (d, "New.dat", nodes, cases, 0, ",\t", "");
```

When this code is executed, a new data file (`New.dat`) is generated. It has the following contents:

```
S,      D,      X

"no",   "no",   "no"
"yes",  "yes",  "yes"
"yes",  ,       "no"
"yes",  ,       "no"
"yes",  "yes",  "yes"
```

Note that the case with index 4 (the fifth case) from the input data file is repeated twice in the output data file. This is because that case has case count 2 in the input data. ∎

---

[2] If the data file is to be read by other applications, it can be useful to use a different separator and/or a different missing data indicator. Therefore, these restrictions are not enforced.

## 11.4 Learning network structure

The algorithm used by HUGIN for learning the network structure is the *PC algorithm* [31]. The current implementation is limited to domains of discrete chance nodes. Domain knowledge (i.e., knowledge of which edges to include or exclude, directions of the edges, or both) is taken into account. Such knowledge is specified as a set of edge constraints (see Section 11.5).

An outline of the algorithm is as follows:

- Case data is specified using the functions described in Section 11.1.
- Statistical tests for conditional independence of pairs of nodes $(X, Y)$ given sets of other nodes $S_{XY}$ (with the size of $S_{XY}$ varying from $0$ to $3$) are performed.
- An undirected graph (called the *skeleton*) is constructed: $X$ and $Y$ are connected with an edge if and only if (1) the edge is required by the edge constraints, or (2) the edge is permitted by the edge constraints and no conditional independence relation for $(X, Y)$ given a set $S_{XY}$ was found in the previous step.
- Edges for which directions are specified by the edge constraints are directed according to the constraints (unless the constraints impose directed cycles).
- Colliders (also known as *v*-structures) (i.e., edges directed at a common node) and derived directions are identified. Edges are directed such that no directed cycles are created.
- The previous step results in a partially directed graph. The remaining edges are arbitrarily directed (one at a time, each edge directed is followed by a step identifying derived directions).

▸ **h_status_t** ***h_domain_learn_structure*** (**h_domain_t** *domain*)

This function creates directed links (found by the PC algorithm) between the nodes of *domain*, which is assumed to contain only discrete chance nodes and no edges. Data must have been entered (using the functions described in Section 11.1), and the number of states for each node must have been set appropriately. The learned network respects the edge constraints specified (see Section 11.5) — unless the edge constraints impose directed cyles.

The PC algorithm only determines the structure of the network. It does *not* calculate the conditional probability tables. This can be done using the *h_domain_learn_tables*(157) function (see Section 11.6).

If a log-file has been specified (using *h_domain_set_log_file*(93)), then a log of the actions taken by the PC algorithm is produced. Such a log is useful for debugging and validation purposes (e.g., to determine which edge directions were determined from data and which were selected at random).

The dependency tests calculate a test statistic which is asymptotically $\chi^2$-distributed assuming (conditional) independence. If the test statistic is large, we reject the independence hypothesis; otherwise, we accept. The probability of rejecting a true independence hypothesis is set using the following function.

▸ **h_status_t** ***h_domain_set_significance_level***
(**h_domain_t** *domain*, **h_double_t** *probability*)

Set the significance level (i.e., the probability of rejecting a true independence hypothesis) to *probability* (a value between 0 and 1) for *domain*. The default value is 0.05.

In general, increasing the significance level will result in more edges, whereas reducing it will result in fewer edges. With fewer edges, the number of arbitrarily directed edges will decrease.

Reducing the significance level will also reduce the running time of *h_domain_learn_structure*.

▸ **h_double_t** ***h_domain_get_significance_level*** (**h_domain_t** *domain*)

Retrieve the current significance level for *domain*.

## 11.5 Domain knowledge

Background knowledge about the domain can be used to constrain the set of networks that can be learned. Such knowledge can be used by the learning algorithm to resolve ambiguities (e.g., deciding the direction of an edge).

Domain knowledge can be knowledge of the direction of an edge, the presence or absence of an edge, or both.

The enumeration type **h_edge_constraint_t** is introduced to represent the set of possible items of knowledge about a particular edge $a - b$. The possibilities are:

- *h_constraint_none* indicates that no constraints are imposed on the learning process. Unless any of the below constraints has been specified, *h_constraint_none* is assumed.
- *h_constraint_edge_required* indicates that an edge must be present, but the direction of the edge is unspecified.
- *h_constraint_edge_forbidden* indicates that no edge is permitted.
- *h_constraint_forward_edge_required* indicates that an edge is required, and that it must be directed from $a$ to $b$.

- *h_constraint_backward_edge_required* indicates that an edge is required, and that it must be directed from $b$ to $a$.
- *h_constraint_forward_edge_forbidden* indicates that, if an edge is present, it must be directed from $b$ to $a$.
- *h_constraint_backward_edge_forbidden* indicates that, if an edge is present, it must be directed from $a$ to $b$.

Moreover, the constant *h_constraint_error* is used to denote error returns from the *h_node_get_edge_constraint* function below.

▸ **h_status_t** ***h_node_set_edge_constraint***
(**h_node_t** $a$, **h_node_t** $b$, **h_edge_constraint_t** *constraint*)

Specify *constraint* as the learning constraint for the edge $a - b$. Note that this also specifies a symmetric constraint for the edge $b - a$ (e.g., specifying *h_constraint_forward_edge_required* for $a - b$ also entails specifying *h_constraint_backward_edge_required* for $b - a$).

▸ **h_edge_constraint_t** ***h_node_get_edge_constraint***
(**h_node_t** $a$, **h_node_t** $b$)

Retrieve the learning constraint specified for the edge $a - b$. If an error occurs, *h_constraint_error* is returned.

## 11.6 Learning conditional probability tables

Before learning of conditional probability tables can take place, the data set and the set of nodes for which conditional probability distributions should be learned must be specified. This set of nodes is specified as the nodes having experience tables. Experience tables are created by the *h_node_get_experience_table*(142) function, and they are deleted by the *h_table_delete*(63) function.

▸ **h_status_t** ***h_domain_learn_tables*** (**h_domain_t** *domain*)

Learn the conditional probability tables for all nodes of *domain* having experience tables. The input to the learning procedure is the case data set specified using the functions described in Section 11.1. If *domain* is a LIMID, then each case must specify a valid evidence scenario (see Section 8.1.3).

In general, the learning algorithm needs to do inference, so *domain* must be compiled before *h_domain_learn_tables* is called. If the computer has sufficient main memory, inference can be speeded up by saving the junction tree tables to memory (using *h_domain_save_to_memory*(125)) prior to the call of *h_domain_learn_tables*. Also, the junction tree potentials must be up-to-date with respect to the node tables (i.e., the CPTs, the policies, and the utility tables) and their models (if any), and the equilibrium must be 'sum' with no evidence incorporated.

As a "sanity" check, the function checks that case data has been entered, and that learning has been enabled for at least one node (i.e., that at least one node has an experience table containing at least one nonnegative count).

If successful, *h_domain_learn_tables* updates the conditional probability table and the experience table for each node of *domain* that has an experience table. Moreover, the inference engine is reset using the new conditional probability tables (see *h_domain_reset_inference_engine*(126)). A retract-evidence operation is implicitly performed as the final step of *h_domain_learn_tables*.

If an error occurs, the state of the conditional probability tables and the inference engine is unspecified.

The method used is the EM algorithm. If the experience count corresponding to some discrete parent configuration is zero, then *h_domain_learn_tables* computes the best ("maximum likelihood") conditional probability distribution for that configuration. If the experience count is positive, then this count is combined with the pre-specified contents of the conditional probability table to form *prior experience* (this is known as "penalized EM"):

- If the node is discrete, then the conditional probability distribution is multiplied by the experience count, resulting in a set of *prior counts*. These counts are added to the counts derived from the data set.
- If the node is continuous, then the joint weak marginal over the parents of the node is computed (based on the pre-specified conditional probability tables).[3] From this potential (and the experience count), the prior probability distribution over the parameters of the CG distribution (Section 2.6) is computed. The details are too complex to cover in this manual.

If the experience count is negative, then learning is disabled for the corresponding parent state configuration.

For the discrete nodes, the starting point of the EM algorithm consists of the pre-specified conditional probability tables. If no tables have been specified, uniform distributions are assumed. Sometimes, it is desirable to enforce zeros in the joint probability distribution. This is done by specifying zeros in the conditional probability tables for the configurations that should be impossible (i.e., have zero probability). However, note that presence of cases in the data set which are impossible according to the initial joint distribution will cause the learning operation to fail.

For the continuous nodes, a suitable starting point is computed from the case data.

---

[3] Because this computation does not involve evidence, the result is affected by the conditional probability tables of the node and all of its ancestors (but no other nodes).

**Example 11.3** The following code loads the `Asia` domain [21] and makes sure that all nodes except `E` have experience tables. All entries of these experience tables are then set to 0 (because we want to compute maximum likelihood estimates of the conditional probability tables). Note that newly created experience tables are already filled with zeros.

```
h_domain_t d = h_kb_load_domain ("Asia.hkb", NULL);
h_node_t E = h_domain_get_node_by_name (d, "E");
h_node_t n = h_domain_get_first_node (d);

for (; n != NULL; n = h_node_get_next (n))
    if (n != E)
    {
        h_boolean_t b = h_node_has_experience_table (n);
        h_table_t t = h_node_get_experience_table (n);

        if (b)
        {
            h_number_t *data = h_table_get_data (t);
            size_t k = h_table_get_size (t);

            for (; k > 0; k--, data++)
                *data = 0.0;
        }
    }

if (h_node_has_experience_table (E))
    h_table_delete (h_node_get_experience_table (E));
```

Now we read and enter into the domain a file of cases (*data_file*). This is done using the *h_domain_parse_cases*(152) function (see Example 12.24 for an appropriate definition of *error_handler*). After having ensured that the domain is compiled, we call *h_domain_learn_tables* in order to learn conditional probability tables for all nodes except `E`. [We assume that the correct conditional probability table has already been specified for `E`, and that the other conditional probability tables contain nonzero values.]

```
h_domain_parse_cases
        (d, data_file, error_handler, data_file);

if (!h_domain_is_compiled (d))
    h_domain_compile (d);

h_domain_learn_tables (d);
```

The *h_domain_learn_tables* operation will also update the experience tables with the counts derived from the file of cases. These experience counts can then form the basis for the sequential learning feature. (But note that if some parent state configurations are absent from the data set, then the corresponding experience counts will be zero.) ∎

The EM algorithm performs a number of iterations. The main purpose of each iteration is to produce an updated set of conditional probability tables. This updated set of tables is either used as input to the next iteration, or it is (part of) the final output of the EM algorithm.

The set of tables produced by an iteration of the EM algorithm is at least as good as the set given at the start of the iteration. The measure of "goodness" is the logarithm of the probability of the case data with respect to the probability distribution determined by the set of tables at the start of the iteration. This quantity is known as the *log-likelihood*, and the EM algorithm attempts to maximize this quantity.

If a log-file has been specified (using *h_domain_set_log_file*(93)), then the log-likelihood computed in each iteration of the EM algorithm is reported to the log-file, as well as the log-likelihood, AIC, and BIC scores of the final model. (See Section 11.2.)

The EM algorithm terminates when the relative difference between the log-likelihood for two successive iterations becomes sufficiently small. This criterion is controlled by the following function.

▸ **h_status_t** ***h_domain_set_log_likelihood_tolerance***
(**h_domain_t** *domain*, **h_double_t** *tolerance*)

Specify that the EM algorithm used by *h_domain_learn_tables* should terminate when the relative difference between the log-likelihood for two successive iterations becomes less than *tolerance* (which must be a positive number).

If this function is not called, a *tolerance* value of $10^{-4}$ is assumed by the EM algorithm.

▸ **h_double_t** ***h_domain_get_log_likelihood_tolerance***
(**h_domain_t** *domain*)

Retrieve the current setting of the log-likelihood tolerance for *domain*. If an error occurs, a negative number is returned.

It is also possible to specify directly the maximum number of iterations performed.

▸ **h_status_t** ***h_domain_set_max_number_of_em_iterations***
(**h_domain_t** *domain*, **size_t** *count*)

Specify that the EM algorithm used by *h_domain_learn_tables* should terminate when *count* number of iterations have been performed (if *count* is positive). If *count* is zero, no limit on the number of iterations is imposed (this is also the initial setting).

▶ **h_count_t** ***h_domain_get_max_number_of_em_iterations***
(**h_domain_t** *domain*)

Retrieve the current setting for *domain* of the maximum number of iterations for the EM algorithm used by *h_domain_learn_tables*. If an error occurs, a negative number is returned.

The EM algorithm terminates when at least one of the conditions described above becomes true.

### Learning CPTs in classes

If we have a runtime domain constructed from some class in an object-oriented model using *h_class_create_domain*(54), then several nodes in the runtime domain will typically be copies of the same class node (i.e., the last element of their source lists will be a common class node — in the following, we shall refer to that class node as the *source node*). Such nodes should be assigned the same conditional probability table by the EM algorithm.

The following function can (only) be used when *domain* is a runtime domain constructed from some class in an object-oriented model (i.e., *domain* must be a domain returned from *h_class_create_domain*(54) — it must *not* be a domain loaded from a file or a domain constructed in some other way).

▶ **h_status_t** ***h_domain_learn_class_tables*** (**h_domain_t** *domain*)

For each node of *domain* such that both the node and its source node have experience tables, the conditional probability and experience tables of both nodes are learned/updated, and the tables of the *domain* node will be identical to those of its source node.

Learning takes place in the object-oriented model (i.e., the conditional probability and experience tables of nodes in the object-oriented model are modified), but the inference part ("the expectation step," or "E-step" for short) of the EM algorithm takes place in the runtime domain. The results of the E-step are combined to produce new conditional probability tables for the nodes in the object-oriented model. These tables are then copied back to the runtime domain so that they will be used in the next E-step. As the final step of the EM algorithm, the conditional probability and experience tables are copied from the object-oriented model to the runtime domain.

The initial contents of the experience tables of nodes in the object-oriented model form the basis for the computation of "prior counts." (See explanation concerning "prior counts" above.) However, in the current implementation, prior experience is not supported for continuous nodes in object-oriented models (that is, the experience tables for such nodes must not contain positive values).

The contents of the updated experience tables reflect the fact that many runtime nodes contribute to the learning of the same source node (i.e., the experience counts will be higher than the number of cases in the data set).

Otherwise, everything specified for the *h_domain_learn_tables*(157) function also apply to the *h_domain_learn_class_tables* function.

Note that *h_domain_learn_class_tables* updates tables of nodes in the runtime domain as well as tables of nodes in the classes comprising the object-oriented model. In fact, the set of updated tables in the classes is typically the desired outcome of calling the function.

The general procedure for learning class tables is as follows:

(1) Make sure that experience tables have been created for the set of nodes in the object-oriented model for which EM learning is desired.

(2) Create a runtime domain. (If the source node corresponding to a runtime node has an experience table, then an experience table will automatically be created for the runtime node.)

(3) Enter case data into the runtime domain.

(4) Compile the runtime domain.

(5) Call *h_domain_learn_class_tables* on the runtime domain.

# Chapter 12

# The NET Language

When a belief network or LIMID model has been constructed using the HUGIN API functions described in Chapter 2 and Chapter 3, it can be filed for later use.

A non-object-oriented model (i.e., a domain) can be stored in a file using a portable (but undocumented) binary format — known as the HUGIN KB format (see Section 2.10).

As an alternative to a binary format, a textual format can be used. As opposed to a binary format, a textual format has the advantage that it can be read, modified, and even created from scratch by means of a standard text editor.

The HUGIN system uses a special-purpose language — called the NET language — for textual specifications of belief networks and LIMID models, object-oriented as well as non-object-oriented. The HUGIN API provides functions to parse (Section 12.8) and generate (Section 12.9) text files containing such specifications.

When new features are added to the HUGIN system, the NET language is similarly extended. However, NET files generated by older versions of the software can always be read by newer versions of the software. Also, the NET language is extended in such a way that unless newer features are being used in a NET file, then the file can also be read by an older version of the HUGIN API (provided it is version 2 or newer[1]).

---

[1] The first revision of the NET language (used by versions 1.$x$ of the HUGIN API) had a fixed format (i.e., the semantics of the different elements were determined by their positions within the specification). This format could not (easily) be extended to support new features, so a completely different format had to be developed.

## 12.1 Overview of the NET language

A domain or class specification in the NET language is conceptually comprised of the following parts:

- Information pertaining to the domain or class as a whole.
- Specification of basic nodes (category, kind, states, label, etc).
- Specification of the relationships between the nodes (i.e., the network structure, and the potentials and functions associated with the links).
- [Classes only] Specification of class instances, including bindings of interface nodes.

The last three parts can be overlapping, except that nodes must be defined before they can be used in specifications of structure or quantitative data.

A specification of a domain in the NET language has the following form:

⟨domain definition⟩ → ⟨domain header⟩ ⟨domain element⟩$^*$
⟨domain header⟩ → `net {` ⟨attribute⟩$^*$ `}`
⟨domain element⟩ → ⟨basic node⟩ | ⟨potential⟩
⟨attribute⟩ → ⟨attribute name⟩ `=` ⟨attribute value⟩ `;`

A specification of a class in the NET language has the following form:

⟨class definition⟩ → `class` ⟨class name⟩ `{` ⟨class element⟩$^*$ `}`
⟨class element⟩ → ⟨domain element⟩ | ⟨attribute⟩ | ⟨class instance⟩

A NET file can contain several class definitions. The only restriction is that classes must be defined before they are instantiated.

Names (⟨class name⟩, ⟨attribute name⟩, etc.) are specified in Section 12.7.

The following sections describe the syntax and semantics of the remaining elements of the grammar: ⟨basic node⟩ (Section 12.2), ⟨class instance⟩ (Section 12.3), and ⟨potential⟩ (Section 12.4 and Section 12.5),

## 12.2 Basic nodes

In ordinary belief networks, a node represents a random variable (either discrete or continuous) — this is known as a *chance* node. In LIMIDs, a node can also represent a (discrete) *decision*, controlled by the decision maker, or a *utility* function, which is used to assign preferences to different configurations of variables. Both types of networks can be extended with *function* nodes. Function nodes can be used to perform computations using the values of other nodes as input. However, it is not possible to specify evidence for function nodes (function nodes are ignored by the inference engine).

This section explains how to specify these different types of nodes in a NET specification. [In object-oriented models, nodes are also used to represent class instances. Class instances are described in Section 12.3.]

**Example 12.1** The following node specification is taken from the "Chest Clinic" example [21].

```
node T
{
    states = ("yes" "no");
    label = "Has tuberculosis?";
    position = (25 275);
}
```

This specifies a binary random variable named T, with states labeled `"yes"` and `"no"`. The specification also provides the label and position, which are used by the HUGIN GUI tool. ∎

A node specification is introduced by one of the keywords: [⟨prefix⟩] **node** (for specifying a chance node), **decision**, **utility**, or **function**, where the optional ⟨prefix⟩ on **node** is either **discrete** or **continuous** (omitting the ⟨prefix⟩ causes **discrete** to be used as default). The keyword is followed by a name that must be unique within the model. See Section 12.7 for the rules of forming valid node names.

A LIMID model (i.e., a model containing decision or utility nodes) must not contain continuous (chance) nodes.

Example 12.1 shows three of the attributes currently defined in the NET language for nodes: *states*, *label*, *position*, *subtype*, and *state_values*. All of these attributes are optional: If an attribute is absent, a default value is used.

- *states* specifies the states of the node: This is a non-empty list of strings, comprising the labels of the states. If the node is used as a labeled node with the table generator facility (Chapter 5), then the labels *must* be unique; otherwise, the labels need not be unique (and can even be empty strings). The length of the list defines the number of states of the node, which is the only quantity needed by the HUGIN inference engine.

  The default value is a list of length one, containing an empty string (i.e., the node has only one state).

  The *states* attribute is only allowed for discrete (chance and decision) nodes.

- *label* is a string that is used by the HUGIN GUI tool when displaying the nodes. The label is not used by the inference engine. The default value is the empty string.

- *position* is a list of integers (the list must have length two). It indicates the position within the graphical display of the network by the HUGIN

GUI tool. The position is not used by the inference engine. The default position is at $(0, 0)$.

- *subtype* specifies the subtype of a discrete (chance or decision) node. The value must be one of the following name tokens: *label*, *boolean*, *number*, or *interval*. See Section 5.1 for more information.

  The default value is *label*.

- *state_values* is a list of numbers, the state values of the node. These values are used by the table generator facility (Chapter 5). This attribute must only appear for nodes of subtypes *number* or *interval* (and must appear after the *subtype* and *states* attributes). If the subtype is *number*, the list must have the same length as the *states* list; if the subtype is *interval*, the list must have one more element than the *states* list.

  The list of numbers must form an increasing sequence.

  If the subtype is *interval*, the first element can be '*–infinity*,' and the last element can be '*infinity*.'

In addition to the standard attributes, an application can introduce its own attributes.

**Example 12.2** Here, the node T has been given the application-specific attribute *MY_APPL_my_attr*.

```
node T
{
    states = ("yes" "no");
    label = "Has tuberculosis?";
    position = (25 275);
    MY_APPL_my_attr = "1000";
}
```

∎

The values of such application-specific attributes must be text strings (see Section 12.7 for a precise definition of text strings).

The names of application-specific attributes should have a common prefix in order to avoid name clashes with attributes defined by HUGIN or other applications (in Example 12.2 the *MY_APPL_* prefix is used).

When a NET specification has been parsed, the values of application-specific attributes can be accessed using the *h_node_get_attribute*(43) and *h_node_set_attribute*(42) functions.

**Example 12.3** In the HUGIN GUI tool, some extra attributes are used to save descriptions of both nodes and their states. These are the attributes prefixed with *HR_*.

```
node T
{
    states = ("yes" "no");
```

```
        label = "Has tuberculosis?";
        position = (25 275);
        HR_State_0 = "The patient has tuberculosis.";
        HR_State_1 = "The patient does not have\
    tuberculosis.";
        HR_Desc = "Represents whether the patient has\
    tuberculosis.";
    }
```

The HUGIN GUI tool uses the UTF-8 encoding for storing arbitrary text in attributes. If you need to have your networks loaded within that tool, you should use the UTF-8 encoding for non-ASCII text. ∎

## 12.3 Class instances

In object-oriented models, a node can also represent a *class instance*. Such a node is introduced using the **instance** keyword:

`instance` ⟨instance name⟩ `:` ⟨class name⟩
`(` [⟨input bindings⟩] [ `;` ⟨output bindings⟩] `)` `{` ⟨node attributes⟩ `}`

This defines an instance of the class with name ⟨class name⟩. Currently, ⟨node attributes⟩ for a class instance can only be a *label*, *position*, or a user-defined attribute.

The ⟨input bindings⟩ specify how formal input nodes of the class instance are associated with actual input nodes. The syntax is as follows:

⟨input bindings⟩ → ⟨input binding⟩ `,` ⟨input bindings⟩
⟨input binding⟩ → ⟨formal input name⟩ = ⟨actual input name⟩

The ⟨formal input name⟩ must refer to a node listed in the *inputs* attribute (see Section 12.6) of the class with name ⟨class name⟩. The node referred to by the ⟨actual input name⟩ must be defined somewhere in the class containing the class instance.

The ⟨input bindings⟩ need not specify bindings for all the formal input nodes of the class (but at most one binding can be specified for each input node).

The ⟨output bindings⟩ are used to give names to output clones. The syntax is similar to that of the input bindings:

⟨output bindings⟩ → ⟨output binding⟩ `,` ⟨output bindings⟩
⟨output binding⟩ → ⟨actual output name⟩ = ⟨formal output name⟩

The ⟨actual output name⟩ is the name assigned to the output clone that corresponds to the output node with name ⟨formal output name⟩ for this particular class instance. An ⟨actual output name⟩ may appear in the *outputs* attribute (see Section 12.6) of a class definition and as a parent in ⟨potential⟩ specifications.

**Example 12.4** The following fragment of a NET specification defines an instance $I_1$ of class C.

```
instance I1 : C (X=X1, Y=Y1; Z1=Z) {...}
```

Class C must have (at least) two input nodes: X and Y. For instance $I_1$, X corresponds to node $X_1$, and Y corresponds to node $Y_1$. Class C must also have (at least) one output node: Z. The output clone corresponding to Z for instance $I_1$ is given the name $Z_1$. ∎

A NET file can contain several class definitions, but the classes must be ordered such that instantiations of a class follow its definition. Often, a NET file will be *self-contained* (i.e., no class instances refer to classes not defined in the file), but it is also possible to store the classes in individual files. When a NET file is parsed, classes will be "looked up" whenever they are instantiated. If the class is already loaded, the loaded class will be used. If no such class is known, it must be created (for example, by calling the parser recursively). See Section 12.8 for further details.

## 12.4 The structure of the model

The structure (i.e., the links of the underlying graph) is specified as part of the ⟨potential⟩ specifications. We have two kinds of links: *directed* and *undirected* links. We denote a directed link from A to B as $A \rightarrow B$, and we denote an undirected link between A and B as $A \sim B$. If there is a directed link from A to B, we say that A is a *parent* of B and that B is a *child* of A.

A network model containing undirected links is called a *chain graph* model.

A ⟨potential⟩ specification is introduced by the **potential** keyword. In the following, we explain how to specify links through a series of examples.

**Example 12.5** This is a typical specification of directed links:

```
potential ( A | B C ) { }
```

This specifies that node A has two parents: B and C. That is, there is a directed link from B to A, and there is also a directed link from C to A. ∎

**Example 12.6** This is a typical specification of undirected links:

```
potential ( A B ) { }
```

This specifies that there is an undirected link between A and B. (If there are no parents, the vertical bar may be omitted.) ∎

**Example 12.7** Directed and undirected links can also be specified together:

```
potential ( A B | C D ) { }
```

This specifies that there is an undirected link between A and B, and it also specifies that A and B both have C and D as parents. ∎

A maximal set of nodes, connected by undirected links, is called a *chain graph component*. If a chain graph component has more than one member, then all members must be discrete chance nodes (i.e., undirected links are only allowed between discrete chance nodes).

The graph must not contain a directed cycle: A directed cycle is a sequence of nodes $X_1, \ldots, X_n$ ($n > 1$ and $X_1 = X_n$), such that either $X_i \rightarrow X_{i+1}$ or $X_i \sim X_{i+1}$ ($1 \leq i < n$), and at least one of the links is directed.

**Example 12.8** The following specification is not allowed, because of the directed cycle $A \rightarrow B \rightarrow C \rightarrow A$.

```
potential ( B | A ) { }
potential ( C | B ) { }
potential ( A | C ) { }
```

However, the following specification is legal.

```
potential ( B | A ) { }
potential ( C | B ) { }
potential ( C | A ) { }
```

∎

**Example 12.9** The following specification is not allowed either, since there is a directed cycle $A \rightarrow B \rightarrow C \sim A$.

```
potential ( B | A ) { }
potential ( C | B ) { }
potential ( A C ) { }
```

However, the following specification is legal.

```
potential ( A | B ) { }
potential ( C | B ) { }
potential ( A C ) { }
```

∎

Chance, decision, and utility nodes can only have chance and decision nodes as parents.

Discrete nodes can only have discrete nodes as parents.

The links directed at decision nodes are called *information* links. The parents of a decision node are exactly those nodes that are assumed to be known when the decision is made.

**Example 12.10** Assume that we want to specify a LIMID with two decisions, $D_1$ and $D_2$, and with three discrete chance variables, $A$, $B$, and $C$. First, $A$ is observed; then, decision $D_1$ is made; then, $B$ is observed; finally, decision $D_2$ is made. This sequence of events can be specified as follows:

```
potential ( D1 | A ) { }
potential ( D2 | D1 B ) { }
```

The last line specifies that the decision maker “forgets” the observation of A before he makes decision $D_2$. If this is not desired, then A should be included in the ⟨potential⟩ specification for $D_2$:

```
potential ( D2 | D1 A B ) { }
```

However, this makes the policy of $D_2$ more complex. Reducing the comlexity of decision making by ignoring less important observations can often be an acceptable (or even necessary) trade-off. ∎

Finally, a node must not be referenced in a ⟨potential⟩ specification before it has been defined by a **node**-, **decision**-, **utility**-, or a **function**-specification.

## 12.5 Potentials

We also need to specify the quantitative part of the model. This part consists of conditional probability potentials for chance nodes, policies for decision nodes, and utility functions for utility nodes. We distinguish between *discrete probability*, *continuous probability*, and *utility* potentials. Discrete probability potentials are used for both chance and decision nodes.

Function nodes do not have potentials, but it is convenient to use ⟨potential⟩ specifications to define the functions represented by the nodes. In this case, we use expressions to specify the functions (see Section 12.5.2).

There are two ways to specify the discrete probability and the utility potentials: (1) by listing the numbers making up the potentials (Section 12.5.1), and (2) by using the table generation facility (Section 12.5.2).

### 12.5.1 Direct specification of the numbers

Direct specification of the quantitative part of the relationship between a group of nodes and their parents is done using the *data* attribute of the ⟨potential⟩ specification.

**Example 12.11** The following specification is taken from the “Chest Clinic” example [21] and specifies the conditional probability table of the discrete variable T.

```
potential ( T | A )
{
    data = (( 0.05 0.95 )          %  A=yes
            ( 0.01 0.99 ));        %  A=no
}
```

This specifies that the probability of tuberculosis given a trip to Asia is 5%, whereas it is only 1% if the subject has not been to Asia.

The *data* attribute may also be specified as an unstructured list of numbers:

```
potential ( T | A )
{
    data = ( 0.05 0.95           %  A=yes
             0.01 0.99 );        %  A=no
}
```

∎

As the example shows, the numerical data is specified through the *data* attribute of a ⟨potential⟩ specification. This data has the form of a list of real numbers. The structure of the list must either correspond to that of a multi-dimensional table with dimension list comprised of the parent nodes followed by the child nodes, or it must be a flat list with no structure at all. In both cases, the 'layout' of the *data* list is *row-major* (see Section 4.1).

**Example 12.12**

```
potential ( D E F | A B C ) { }
```

The *data* attribute of this ⟨potential⟩ specification corresponds to a multi-dimensional table with dimension list $\langle A, B, C, D, E, F\rangle$. ∎

The *data* attribute of a utility potential corresponds to a multi-dimensional table with dimension list comprised of the nodes to the right of the vertical bar.

**Example 12.13** The following specification is taken from the "Oil Wildcatter" example and shows a utility potential. `DrillProfit` is a utility node, while `Oil` is a discrete chance node with three states, and `Drill` is a decision node with two states.

```
potential (DrillProfit | Drill Oil)
{
    data = (( -70        %  Drill=yes  Oil=dry
               50        %  Drill=yes  Oil=wet
              200 )      %  Drill=yes  Oil=soaking
            (   0        %  Drill=no   Oil=dry
                0        %  Drill=no   Oil=wet
                0 ));    %  Drill=no   Oil=soaking
}
```

The *data* attribute of this ⟨potential⟩ specification corresponds to a multi-dimensional table with dimension list ⟨`Drill`, `Oil`⟩. ∎

The (multi-dimensional) table corresponding to the *data* attribute of a continuous probability potential has dimension list comprised of the discrete parent nodes of the ⟨potential⟩ specification (in the given order). These nodes must be listed first on the right side of the vertical bar, followed by the continuous parent nodes. However, the items in the table are no longer numbers but instead *continuous distribution functions*; only normal (i.e., Gaussian) distributions can be used. A normal distribution is specified by its mean and variance. In the following example, a continuous probability potential is specified.

**Example 12.14** Suppose A is a continuous node with parents B and C, which are both discrete. Also, both B and C have two states: B has states $b_1$ and $b_2$ while C has states $c_1$ and $c_2$.

```
potential (A | B C)
{
    data = (( normal (   0, 1   )       %  B=b1  C=c1
              normal (  -1, 1   ) )     %  B=b1  C=c2
            ( normal (   1, 1   )       %  B=b2  C=c1
              normal ( 2.5, 1.5 ) ));   %  B=b2  C=c2
}
```

The *data* attribute of this ⟨potential⟩ specification corresponds to a table with dimension list $\langle B, C \rangle$. Each entry contains a probability distribution for the continuous node A. ∎

All entries in the above example contain a specification of a normal distribution. A normal distribution is specified with the keyword **normal** followed by a list of two parameters. The first parameter is the mean and the second parameter is the variance of the normal distribution.

**Example 12.15** Let A be a continuous node with one discrete parent B (with states $b_1$ and $b_2$) and one continuous parent C.

```
potential (A | B C)
{
    data = ( normal ( 1 + C, 1 )                %  B=b1
             normal ( 1 + 1.5 * C, 2.5 ) );     %  B=b2
}
```

The *data* attribute of this ⟨potential⟩ specification corresponds to a table with dimension list $\langle B \rangle$ (B is the only discrete parent, and it must therefore be listed first on the right side of the vertical bar). Each entry again contains a continuous distribution function for A. The influence of C on A now comes from the use of C in the expressions specifying the mean parameters of the normal distributions. ∎

Only the mean parameter of a normal distribution can be specified as an expression. The variance parameter must be a numeric constant. The expression for the mean parameter must be a linear function of the continuous parents: each term of the expression must be (1) a numeric constant, (2) the name of a continuous parent, or (3) a numeric constant followed by '*' followed by the name of a continuous parent.

If the *data* attribute of a ⟨potential⟩ specification is missing, a list of 1s is assumed for discrete probability potentials, whereas a list of 0s is assumed for utility potentials. For a continuous probability potential, a list of normal distributions with mean set to 0 and variance set to 1 is assumed.

The values of the *data* attribute of discrete probability potentials must only contain nonnegative numbers. In the specification of a normal distribution for a continuous probability potential, only nonnegative numbers are allowed for the variance parameter. There is no such restriction on the values of utility potentials or the mean parameter of a normal distribution.

### 12.5.2 Using the table generation facility

For potentials that do not involve CG variables, a different method for specifying the quantitative part of the relationship for a single node and its parents is provided.

**Example 12.16** Let A denote the number of 1s in a throw with B (possibly biased) dice, where the probability of getting a 1 in a throw with a single die is C. The specification of the conditional probability potential for A given B and C can be given using the table generation facility described in Chapter 5 as follows:

```
potential (A | B C)
{
    model_nodes = ();
    samples_per_interval = 50;
    model_data = ( Binomial (B, C) );
}
```

First, we list the *model_nodes* attribute: This defines the set of configurations for the *model_data* attribute. In this case, the list is empty, meaning that there is just one configuration. The expression for that configuration is the binomial distribution expression shown in the *model_data* attribute.

C will typically be an interval node (i.e., its states represent intervals). However, when computing the binomial distribution, a specific value for C is needed. This is handled by choosing 50 distinct values within the given interval and computing the distributions corresponding to those values. The average of these distributions is then taken as the conditional distribution for A given the value of B and the interval (i.e., state) of C. The number 50 is specified by the *samples_per_interval* attribute. See Section 5.9 for further details. ∎

**Example 12.17** In the "Chest Clinic" example [21], the node E is specified as a logical OR of its parents, T and L. Assuming that all three nodes are of labeled subtype with states *yes* and *no* (in that order), the potential for E can be specified as follows:

```
potential (E | T L)
{
    model_nodes = (T L);
    model_data = ( "yes", "yes", "yes", "no" );
}
```

An equivalent specification can be given in terms of the OR operator:

```
potential (E | T L)
{
    model_nodes = ();
    model_data
        = ( if (or (T="yes", L="yes"), "yes", "no") );
}
```

If all three nodes are given a boolean subtype, the specification can be simplified to the following:

```
potential (E | T L)
{
    model_nodes = ();
    model_data = ( or (T, L) );
}
```

∎

In general, the *model_nodes* attribute is a list containing a subset of the discrete parents listed to the right of the vertical bar in the ⟨potential⟩ specification. The order of the nodes in the *model_nodes* list defines the interpretation of the *model_data* attribute: The *model_data* attribute is a comma-separated list of expressions, one for each configuration of the nodes in the *model_nodes* list. As usual, the ordering of these configurations is row-major.

A non-empty *model_nodes* list is a convenient way to specify a model with distinct expressions for distinct parent state configurations. An alternative is nested **if**-expressions. See Example 12.17.

The *model_nodes* attribute must appear before the *samples_per_interval* and *model_data* attributes.

The complete definition of the syntax of expressions is given in Section 5.3.

If both a specification using the model attributes and a specification using the *data* attribute are provided, then the specification in the *data* attribute is assumed to be correct (regardless of whether it was generated from the model). The functions that generate NET files (Section 12.9) will output both, if HUGIN “thinks” that the table is up-to-date with respect to the model (see the description of *h_node_generate_table*(83) for precise details). Since generating a table from its model can be a very expensive operation, having a (redundant) specification in the *data* attribute can be considered a “cache” for *h_node_generate_table*.

### Function nodes

Function nodes do not have potentials, but it is convenient to use expressions to specify the functional relationship between a function node and its parents.

This is achieved by using a ⟨potential⟩ specification containing only the *model_nodes* and *model_data* attributes (the *data* attribute is not allowed). In this case, all expressions must be of numeric type.

### 12.5.3 Parameter learning

Information for use by the adaptation facility (see Chapter 10) is specified through the *experience* and *fading* attributes of a ⟨potential⟩ specification. These attributes have the same syntax as the *data* attribute.

The experience data is also used to control the EM algoritm (Section 11.6).

The *experience* attribute is only allowed in ⟨potential⟩ specifications for single (that is, there must be exactly one "child" node in the specification) chance nodes. The *fading* attribute is only allowed in ⟨potential⟩ specifications for single discrete chance nodes.

For the adaptation algoritm, valid experience counts must be positive numbers, while the EM algoritm only requires nonnegative numbers. Specifying an invalid value for some parent state configuration turns off parameter learning for that configuration. If the ⟨potential⟩ specification doesn't contain an *experience* attribute, parameter learning is turned off completely for the child node.

A fading factor $\lambda$ is valid if $0 < \lambda \leq 1$. Specifying an invalid fading factor for some parent state configuration turns off adaptation for that configuration. If the ⟨potential⟩ specification doesn't contain a *fading* attribute, then all fading factors are considered to be equal to 1 (which implies no fading).

See Chapter 10 and Section 11.6 for further details.

**Example 12.18** The following shows a specification of experience and fading information for the node D ('Dyspnoea') from the "Chest Clinic" example in [21]. This node has two parents, E and B. We specify an experience count and a fading factor for each configuration of states of ⟨E, B⟩.

```
potential (D | E B)
{
   data = ((( 0.9 0.1 )     %  E=yes   B=yes
            ( 0.7 0.3 ))    %  E=yes   B=no
           (( 0.8 0.2 )     %  E=no    B=yes
            ( 0.1 0.9 ))); %  E=no    B=no
   experience = (( 10       %  E=yes   B=yes
                   12 )     %  E=yes   B=no
                 (  0       %  E=no    B=yes
                   14 ));   %  E=no    B=no
   fading = (( 1.0          %  E=yes   B=yes
               0.9 )        %  E=yes   B=no
             ( 1.0          %  E=no    B=yes
               1.0 ));      %  E=no    B=no
}
```

Note that the experience count for the $E = no/B = yes$ state configuration is 0. This value is an invalid experience count for the adaptation algoritm (but not for the EM algoritm), so adaptation is turned off for that particular state configuration. Also, note that only the experience count for the $E = yes/B = no$ state configuration will be faded during adaptation (since the other parent state configurations have fading factors equal to 1). ∎

## 12.6 Global information

Information pertaining to the belief network or LIMID model as a whole is specified as attributes within the ⟨domain header⟩ (for domains) or within the ⟨class definition⟩ (for classes).

**Example 12.19** The HUGIN GUI tool uses several parameters when displaying networks.

```
net
{
    node_size = (100 40);
}
```

This specifies that nodes should be displayed with width 100 and height 40. ∎

Currently, only the *node_size* attribute is recognized as a special global attribute. However, as with nodes, extra attributes can be specified. These extra attributes must take strings as values. The attributes are accessed using the HUGIN API functions *h_domain_get_attribute*(43), *h_domain_set_attribute*(43), *h_class_get_attribute*(57), and *h_class_set_attribute*(57).

The HUGIN GUI tool uses the UTF-8 encoding for storing arbitrary text in attributes. If you need to have your networks loaded within that tool, you should use the UTF-8 encoding for non-ASCII text.

**Example 12.20**

```
net
{
    node_size = (100 40);
    MY_APPL_my_attr = "1000";
}
```

This specification has an application specific attribute named *MY_APPL_my_attr*. ∎

**Example 12.21** Recent versions of the HUGIN GUI tool use several application specific attributes. Some of them are shown here:

```
net
{
    node_size = (80 40);
```

```
        HR_Grid_X = "10";
        HR_Grid_Y = "10";
        HR_Grid_GridSnap = "1";
        HR_Grid_GridShow = "0";
        HR_Font_Name = "Arial";
        HR_Font_Size = "-12";
        HR_Font_Weight = "400";
        HR_Font_Italic = "0";
        HR_Propagate_Auto = "0";
    }
```

HUGIN GUI uses the prefix *HR_* on all of its application specific attributes (a predecessor of the HUGIN GUI tool was named HUGIN Runtime). ∎

Global attributes are used in a ⟨class definition⟩ to specify the interface of the class. The *inputs* and *outputs* attributes are used to specify the input nodes and the output nodes of the class, respectively. The values of these attributes are node lists (with the same syntax as that of the *model_nodes* attribute). The nodes mentioned in those attributes must be defined within the class.

**Example 12.22** The following class specification defines a class C with two inputs, X and Y, and one output, Z.

```
    class C
    {
        inputs = (X Y);
        outputs = (Z);

        node X
          ...

        node Y
          ...

        node Z
          ...

        ...
    }
```

∎

## 12.7 Lexical matters

In general, a name has the same structure as an identifier in the C programming language. That is, a name is a non-empty sequence of letters and digits, beginning with a letter. In this context, the underscore character (_)

is considered a letter. The case of letters is significant. The sequence of letters and digits forming a name extends as far as possible; it is terminated by the first non-letter/digit character (for example, braces or whitespace).

As an exception, a node name in a domain specification (but not a class specification) can be a "dotted" name: A list of names (as specified above) with a single dot (a period) separating each pair of consecutive names in the list. (Note: Whitespace is not allowed in a "dotted" name.) This exception makes it possible to assign unique meaningful names to nodes in a runtime domain (see Section 3.10).

A string is a sequence of characters not containing a quote character (`"`) or a newline character; its start and ending are indicated by quote characters.

A number is comprised of an optional sign, followed by a sequence of digits, possibly containing a decimal point character, and an optional exponent field containing an `E` or `e` followed by an (optionally signed) integer.

Comments can be placed in a NET specification anywhere (except within a name, a number, or other multi-character lexical elements). It is considered equivalent to whitespace. A comment is introduced by a percent character (%) and extends to the end of the line.

## 12.8 Parsing NET files

The HUGIN API provides two functions for parsing models specified in the NET language: one for non-object-oriented models ("domains"), and one for object-oriented models ("classes").

This function parses non-object-oriented specifications (i.e., NET files starting with the **net** keyword) and creates a corresponding **h_domain_t** object.

▸ **h_domain_t** ***h_net_parse_domain***
(**h_string_t** *file_name*,
**void** (∗*error_handler*) (**h_location_t**, **h_string_t**, **void** ∗),
**void** ∗*data*)

Parse the NET specification in the file with name *file_name*. If an error is detected (or a warning is issued), the *error_handler* function is called with a line number (indicating the location of the error within the file), a string that describes the error, and *data*. The storage used to hold the string is reclaimed by *h_net_parse_domain* when *error_handler* returns (so if the error/warning message will be needed later, a copy must be made).

The user-specified *data* allows the error handler to access non-local data (and hence preserve state between calls) without having to use global variables.

The **h_location_t** type is an unsigned integer type (such as **unsigned long**).

If no error reports are desired (in this case, only the error indicator returned by *h_error_code*(18) will be available), then the *error_handler* argument may be NULL. (In this case, warnings will be completely ignored.)

If the NET specification is successfully parsed, an opaque reference to the created domain structure is returned; otherwise, NULL is returned. The domain is *not* compiled; use a compilation function to get a compiled version.

**Example 12.23** The error handler function could be written as follows.

```
void my_error_handler
   (h_location_t line_no, h_string_t message, void *data)
{
    fprintf (stderr, "Error at line %d: %s\n",
             line_no, message);
}
```

This error handler simply writes all messages to *stderr*. See Example 12.24 for a different error handler. ∎

The following function must be used when parsing NET files containing class specifications (i.e., NET files starting with the **class** keyword).

▸ **h_status_t** ***h_net_parse_classes***
  (**h_string_t** *file_name*, **h_class_collection_t** *cc*,
    **void** (∗*get_class*) (**h_string_t**, **h_class_collection_t**, **void** ∗),
    **void** (∗*error_handler*) (**h_location_t**, **h_string_t**, **void** ∗),
    **void** ∗*data*)

This function parses the contents of the file with name *file_name*. This file must contain a sequence of class definitions. The parsed classes are stored in class collection *cc*.

In order to create the instance nodes (which represent instances of other classes), it may be necessary to load these other classes: If an instance of a class not present in *cc* is specified in the NET file, *get_class* is called with the name of the class, the class collection *cc*, and the user-specified *data*. The *get_class* function is supposed to load the named class into class collection *cc* (if it doesn't, then parsing is terminated). If the named class contains instances of other classes not present in *cc*, these classes must be loaded (or constructed) as well. The *get_class* function should not perform any other kind of actions. For example, it should not delete or rename any of the existing classes in *cc* — such actions may cause the HUGIN API to crash.

If the specified NET file is self-contained (i.e., no instance declaration refers to a class not specified in the file), then the *get_class* argument can be NULL.

Note that instance nodes are created when the corresponding **instance** definition is seen in the NET file. At that point, the instantiated class must have been loaded (or *get_class* will be called). For this reason, if the NET file contains several class definitions, classes must be defined before they are instantiated.

If an error is detected, the *error_handler* function is called with a line number, indicating the location within the source file currently being parsed, and a string that describes the error. The storage used to hold this string is reclaimed by *h_net_parse_classes* when *error_handler* returns (so if the error message will be needed later, a copy must be made).

If parsing fails, then *h_net_parse_classes* will try to preserve the initial contents of *cc* by deleting the new (and possibly incomplete) classes before it returns. If *get_class* has modified any of the classes initially in *cc*, then this may not be possible. Also, if the changes are sufficiently vicious, then removing the new classes might not even be possible. However, if *get_class* only does things it is supposed to do, there will be no problems.

As described above, the *get_class* function must insert a class with the specified name into the given class collection. This can be done by whatever means are convenient, such as calling the parser recursively, or through explicit construction of the class.

**Example 12.24** Suppose we have classes stored in separate files in a common directory, and that the name of each file is the name of the class stored in the file with `.net` appended. Then the *get_class* function could be written as follows:

```
void get_class
   (h_string_t name, h_class_collection_t cc, void *data)
{
    h_string_t file_name = malloc (strlen (name) + 5);

    if (file_name == NULL)
        return;

    (void) strcat (strcpy (file_name, name), ".net");

    (void) h_net_parse_classes
               (file_name, cc, get_class, error_handler,
                file_name);

    free (file_name);
}

void error_handler
   (h_location_t line_no, h_string_t err_msg, void *data)
{
    fprintf (stderr, "Error in file %s at line %lu: %s\n",
             (h_string_t) data, (unsigned long) line_no,
             err_msg);
}
```

Note that we pass the *file_name* as the *data* argument to *h_net_parse_classes*. This means that the error handler receives the name of the file as its third argument.

If more data is needed by either *get_class* or the error handler, then the *data* argument can be specified as a pointer to a structure containing the needed data items. ∎

## 12.9 Saving class collections, classes, and domains as NET files

The following functions can be used to create NET files.

▶ **h_status_t** ***h_cc_save_as_net***
(**h_class_collection_t** *cc*, **h_string_t** *file_name*)

▶ **h_status_t** ***h_class_save_as_net*** (**h_class_t** *class*, **h_string_t** *file_name*)

▶ **h_status_t** ***h_domain_save_as_net***
(**h_domain_t** *domain*, **h_string_t** *file_name*)

Save the class collection, class, or domain as a text file with name *file_name*. The format of the file is as required by the NET language.

Saving a class collection as a NET file is convenient when you must send the object-oriented model via email, since the resulting NET specification must necessarily be self-contained.

Note that if a NET file is parsed and then saved again, any comments in the original file will be lost. Also note that if (some of) the nodes have not been assigned names, then names will automatically be assigned (through calls to the *h_node_get_name*(39) function). Likewise, if a class has not been named, the "save-as-NET" operation will assign a name (by calling *h_class_get_name*(49)).

▶ **h_string_t** ***h_class_get_file_name*** (**h_class_t** *class*)

Return the file name used for the most recent (successful) save-operation applied to *class*. If no such operation has been performed, or *class* is NULL, NULL is returned.

▶ **h_string_t** ***h_domain_get_file_name*** (**h_domain_t** *domain*)

Return the file name used for the most recent (successful) save-operation applied to *domain*. If no such operation has been performed, or *domain* is NULL, NULL is returned.

Note that domains may be saved both as a NET and as a HUGIN KB file.

# Chapter 13

# Display Information

The HUGIN API was developed partly to satisfy the needs of the HUGIN GUI application. This application can present an arbitrary belief network or LIMID model. To do this, it was necessary to associate a certain amount of "graphical" information with each node of the network. The functions to support this are hereby provided for the benefit of the general API user.

Please note that not all items of graphical information have a special interface (such as the one provided for the label of a node — see Section 13.1 below). Many more items of graphical information have been added using the attribute interface described in Section 2.9.2. To find the names of these extra attributes, take a look at the NET files generated by the HUGIN GUI application.

## 13.1 The label of a node

In addition to the *name* (the syntax of which is restricted), a node can be assigned an arbitrary string, called the *label*.

▸ **h_status_t** ***h_node_set_label*** (**h_node_t** *node*, **h_string_t** *label*)

Make a copy of *label* and assign it as the label of *node*. There are no restrictions on the contents of the label.

Note that a copy of *label* is stored inside the node structure, not *label* itself.

OOBN: If *node* is an output clone, then the label is not saved if the class is saved as a NET file (because the NET file format doesn't support that).

▸ **h_string_t** ***h_node_get_label*** (**h_node_t** *node*)

Returns the label of *node*. If no label has been associated with *node*, the empty string is returned. On error, NULL is returned.

Note that the string returned is the one stored in the node structure. Do not free it yourself.

## 13.2 The position of a node

In order to display a network graphically, the HUGIN GUI application associates with each node a *position* in a two-dimensional coordinate system.

The coordinates used by HUGIN are integral values; their type is **h_coordinate_t**.

▶ **h_status_t** ***h_node_set_position***
(**h_node_t** *node*, **h_coordinate_t** *x*, **h_coordinate_t** *y*)

Set the position of *node* to $(x, y)$.

▶ **h_status_t** ***h_node_get_position***
(**h_node_t** *node*, **h_coordinate_t** **x*, **h_coordinate_t** **y*)

Retrieve the position ($x$- and $y$-coordinates) of *node*. On error, the values of *x* and *y* are indeterminate.

## 13.3 The size of a node

As part of a belief network/LIMID specification, the dimensions (width and height) of a node in a graphical representation of the network can be given. These parameters apply to all nodes of a domain or a class and are needed in applications that display the layout of the network in a graphical manner. An application can modify and inspect these parameters using the functions described below.

▶ **h_status_t** ***h_domain_set_node_size***
(**h_domain_t** *domain*, **size_t** *width*, **size_t** *height*)

Set the width and height dimensions of nodes of *domain* to *width* and *height*, respectively.

▶ **h_status_t** ***h_domain_get_node_size***
(**h_domain_t** *domain*, **size_t** **width*, **size_t** **height*)

Retrieve the dimensions of nodes of *domain*. If an error occurs, the values of variables pointed to by *width* and *height* will be indeterminate.

**Example 13.1** In an application using a graphical display of a network, a node could be drawn using the following function.

```
void draw_node (h_node_t n)
{
    h_domain_t d = h_node_get_domain (n);
    size_t w, h;
    h_coordinate_t x, y;
```

```
        h_domain_get_node_size (d, &w, &h);
        h_node_get_position (n, &x, &y);

        draw_rectangle (x, y, w, h);
    }
```

Here, *draw_rectangle* is an application-defined function, or maybe a function defined in a graphics library, e.g., *XDrawRect* if you are using the X Window System. ∎

In a similar way, the width and height dimensions of nodes belonging to classes in object-oriented models can be accessed.

▸ **h_status_t** ***h_class_set_node_size***
(**h_class_t** *class*, **size_t** *width*, **size_t** *height*)

Set the width and height dimensions of nodes of *class* to *width* and *height*, respectively.

▸ **h_status_t** ***h_class_get_node_size***
(**h_class_t** *class*, **size_t** **width*, **size_t** **height*)

Retrieve the dimensions of nodes of *class*. If an error occurs, the values of variables pointed to by *width* and *height* will be indeterminate.

# Appendix A

# Belief networks with Conditional Gaussian variables

Beginning with Version 3, the HUGIN API can handle networks with both discrete and continuous random variables. The continuous random variables must have a *Gaussian* (also known as a *normal*) distribution conditional on the values of the parents.

The distribution for a continuous variable $Y$ with discrete parents $I$ and continuous parents $Z$ is a (one-dimensional) Gaussian distribution conditional on the values of the parents:

$$P(Y|I = i, Z = z) = \mathcal{N}(\alpha(i) + \beta(i)^{T} z, \gamma(i))$$

Note that the mean depends linearly on the continuous parent variables and that the variance does not depend on the continuous parent variables. However, both the linear function and the variance are allowed to depend on the discrete parent variables. These restrictions ensure that exact inference is possible.

Discrete variables cannot have continuous parents.

**Example A.1** Figure A.1 shows a belief network model for a waste incinerator:

> "The emissions [of dust and heavy metals] from a waste incinerator differ because of compositional differences in incoming waste [$W$]. Another important factor is the waste burning regimen [$B$], which can be monitored by measuring the concentration of $CO_2$ in the emissions [$C$]. The filter efficiency [$E$] depends on the technical state [$F$] of the electrofilter and on the amount and composition of waste [$W$]. The emission of heavy metals [$M_o$] depends on both the concentration of metals [$M_i$] in the incoming waste and the emission of dust particulates [$D$] in general. The emission of dust [$D$] is monitored through measuring the penetrability of light [$L$]." [16]

∎

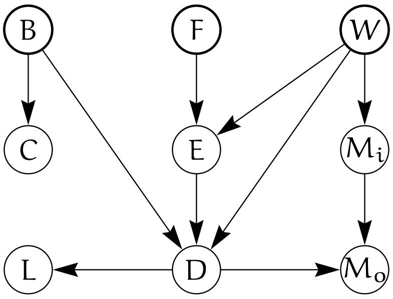

Figure A.1: The structural aspects of the waste incinerator model described in Example A.1: B, F, and $W$ are discrete variables, while the remaining variables are continuous.

The result of inference within a belief network model containing Conditional Gaussian variables is the beliefs (i.e., marginal distributions) of the individual variables given evidence. For a discrete variable this (as usual) amounts to a probability distribution over the states of the variable. For a Conditional Gaussian variable two measures are provided:

(1) the mean and variance of the distribution;

(2) since the distribution is in general not a simple Gaussian distribution, but a mixture (i.e., a weighted sum) of Gaussians, a list of the parameters (weight, mean, and variance) for each of the Gaussians is available.

The algorithms necessary for computing these results are described in [19].

**Example A.2** From the network shown in Figure A.1 (and given that the discrete variables B, F, and $W$ are all binary), we see that

- the distribution for C can be comprised of up to two Gaussians (one if B is instantiated);
- initially (i.e., with no evidence incorporated), the distribution for E is comprised of up to four Gaussians;
- if L is instantiated (and none of B, F, or $W$ is instantiated), then the distribution for E is comprised of up to eight Gaussians.

∎

# Appendix B

# HUGIN API Revision History

This appendix contains lists (in reverse chronological order) of changes and new features in all releases of the HUGIN API since version 2.

### Overview of changes and new features in HUGIN API version 7.3

- A new node type has been introduced: The *function* node type. This node type is used to express calculations to be performed using the results of inference or simulation as input. However, function nodes are not involved in the inference process itself, so evidence cannot be specified for function nodes. The exact rules for evaluation of function nodes are specified in Section 8.7.

- Here is a list of changes related to the introduction of function nodes:
    - The category of function nodes is *h_category_function*, and the kind is *h_kind_other* (this kind is also used for utility and instance nodes). See Section 2.1.
    - A real-valued function is associated with each function node. This function is specified using a model (Chapter 5).
    - The function associated with a function node must only depend on the parents of the node. The parents can be of any type of node (except instance nodes). See Section 2.4.
    - The value of a function node based on the results of inference is requested by *h_node_get_value*(112).
    - The value of a function node based on the results of simulation is requested by *h_node_get_sampled_value*(129).
    - The *h_domain_new_node*(30), *h_node_clone*(30), and *h_node_delete*(30) functions do not uncompile if a function node is involved.

  - The *h_node_add_parent*(31), *h_node_remove_parent*(32), and *h_node_switch_parent*(32) functions do not uncompile if the child node is a function node.
  - The HKB format has been updated to support networks with function nodes (but the old format is still used for networks without function nodes).
  - The NET language has been extended: The new **function** keyword denotes the definition of a function node, and ⟨potential⟩ specifications are now also used to specify links and models for function nodes. See Chapter 12.

- The results of simulation are now invalidated by (implicit as well as explicit) uncompile operations.
- The new *h_node_get_sampled_utility*(129) function requests the "sampled" utility of a utility node.

## Overview of changes and new features in HUGIN API version 7.2

- A .NET Compact Framework version of the HUGIN API is now available, providing support for PDAs.
- An algorithm for finding the "requisite" parents of a decision node in a LIMID has been implemented. See Section 2.4.1.
- A Monte Carlo algorithm for finding the most probable configurations of a set of nodes has been implemented. See Section 9.11.
- A new triangulation method has been implemented: This method triangulates each prime component using all of the elimination based triangulation heuristics and uses the best result. See Section 6.3.

  This triangulation method is now used by *h_domain_compile*(89) when compiling untriangulated domains.
- As the elimination based triangulation heuristics may produce non-minimal triangulations, an extra pass that removes "redundant fill-in edges" has been added to these heuristics.

## Overview of changes and new features in HUGIN API version 7.1

- The EM algorithm has been extended to learn CG distributions for continuous nodes.
- The EM algorithm now permits learning to be enabled/disabled for specific parent configurations: A nonnegative experience count enables learning, while a negative experience count disables learning.

- The EM algorithm now checks that the equilibrium is 'sum' with no evidence incorporated. As a "sanity" check, it also verifies that learning is enabled (that is, there must exist at least one node with a nonnegative experience count), and that case data has been specified.
- Because the EM algorithm is controlled by experience tables, continuous nodes may now also be given experience tables.
- The HKB format has been updated (in order to support parameter learning for continuous nodes).
- The model scoring functions (Section 11.2) now check that the junction tree potentials are up-to-date with respect to the node tables and their models (if any).

## Overview of changes and new features in HUGIN API version 7.0

- HUGIN API libraries for the Windows platforms are now provided for Visual Studio 2008 (in addition to Visual Studio 6.0, Visual Studio .NET 2003, and Visual Studio 2005).
- The *h_domain_compile*(89) function now initializes the junction trees using the available evidence (if any).
- The *h_class_create_domain*(54) function now assigns unique "meaningful" names to the nodes of the created domains. This is accomplished by allowing "dotted names" for nodes in domains (but not for nodes in classes).

  This change affects the lexical syntax of node names in files (NET, data, case, and node list files) and in strings containing expressions.
- Sensitivity analysis: It is now possible to compute sensitivity data for multiple output probabilities simultaneously. This allows for easy solution of constraints on the output probabilities.
- *Limited memory influence diagrams* (LIMIDs) [20] replace influence diagrams as the tool for modeling decision problems.

  This has several implications:

  - There is no "no-forgetting" assumption in a LIMID: All information links must be explicitly represented in the network.
  - A total order of the decisions is no longer required.
  - Evidence specified for a LIMID must satisfy the "free will" condition: A chance node can't be observed before all decisions in the ancestral set of the chance node have been made.

    Moreover, only simple instantiations are now permitted as evidence for decision nodes.

    See Section 8.1.3 and *h_error_invalid_evidence*(120).

- Computing optimal decisions is now a separate operation (that is, they are no longer computed as part of inference). See Section 9.3.
- Decision nodes now have tables. These tables represent decision policies and can be specified in the same way (including the use of models) as conditional probability and utility tables. However, they are usually computed by *h_domain_update_policies*$^{(122)}$.
- In LIMIDs, there are no constraints on elimination orders used for triangulation.
- The overall expected utility of the decision problem is provided by the new *h_domain_get_expected_utility*$^{(111)}$ function.
- *h_node_get_belief*$^{(108)}$ and *h_node_get_expected_utility*$^{(111)}$ now accept (discrete) chance as well as decision nodes as arguments, and *h_node_get_expected_utility* also accepts utility nodes.
- The usage conditions and the treatment of information links has changed for *d*-separation analyses in LIMIDs. See Section 8.4 for further details.
- The *nodes* argument of *h_domain_get_marginal*$^{(109)}$ may now contain decision as well as chance nodes. Also, decision nodes are no longer required to be instantiated and propagated.
- The *h_domain_simulate*$^{(128)}$ operation no longer requires decision nodes to be instantiated and (for compiled domains) propagated.
- The EM algorithm and the *h_domain_get_log_likelihood*$^{(150)}$, *h_domain_get_AIC*$^{(151)}$, and *h_domain_get_BIC*$^{(151)}$ functions no longer require decision nodes to be instantiated in all cases of the data set. Instead, the evidence in a given case is only required to be "valid."
- Sensitivity analysis: Sensitivity functions where the input parameter is associated with a decision node can now be computed. Also, decision nodes are no longer required to be instantiated and propagated.
- The arguments of *h_node_get_entropy*$^{(131)}$ and *h_node_get_mutual_information*$^{(131)}$ may now be decision as well as chance nodes. Also, decision nodes are no longer required to be instantiated and propagated.
- The *h_domain_save_to_memory*$^{(125)}$ operation is disabled for LIMIDs.
- Calculation of conflict of evidence is not supported for LIMIDs.
- The NET format has been extended in order to represent policies.
- The HKB format has changed.

## Overview of changes and new features in HUGIN API version 6.7

- Sensitivity analysis: Functions to aid in the process of identifying the most influential (conditional probability) parameters of a belief network model and analyzing their effects on the "output" probabilities of the model are now provided.

  See Section 9.10.
- New statistical distributions: LogNormal, Triangular, and PERT.

  See Section 5.7.1.
- An optional location parameter has been added to the Gamma, Exponential, and Weibull distributions.

  See Section 5.7.1.
- Truncated continuous distributions can be expressed using a new truncation operator: It is possible to specify double as well as single truncation (single truncation is either left or right truncation).

  See *h_operator_truncate*(70).
- It is now possible to combine expressions of different types in conditional expressions. This permits specification of relationships that are sometimes probabilistic and sometimes deterministic.

  See *h_operator_if*(71).
- The HKB format has changed (because of the new expression operators).
- The performance of *h_domain_get_marginal*(109) has been improved.
- The performance of inference has been improved for the case where a memory backup is not available.
- A memory backup can no longer be created when evidence has been propagated.

## Overview of changes and new features in HUGIN API version 6.6

- A new .NET API is now available for the Windows platforms. This API targets the .NET 2.0 framework.
- HUGIN API libraries for the Windows platforms are now provided for Visual Studio 2005 (in addition to Visual Studio 6.0 and Visual Studio .NET 2003).
- The HUGIN APIs (except the Visual Basic API) are now available as 64-bit versions on all platforms except Mac OS X.

  On the Windows platforms, the 64-bit libraries are only provided for Visual Studio 2005.

- A new naming scheme has been introduced for the HUGIN API libraries on the Windows platforms: The libraries are now uniquely named, making it possible to have all DLLs in the search path simultaneously.

- *d*-separation analysis is now used to improve the performance of inference. This is particularly useful for incremental propagation of evidence in large networks.

- The performance of the total-weight triangulation method has been greatly improved.

- The triangulation functions now construct junction trees, but do not allocate storage for the data arrays of the clique and separator tables. This permits the application to see the junction trees before attempting the final (and most expensive) part of the compilation process.

- It is now possible to query the size of a junction tree (even before storage is allocated for the junction tree tables). See *h_jt_get_total_size*(101) and *h_jt_get_total_cg_size*(101).

- A function — *h_domain_is_triangulated*(92) — for testing whether a domain is triangulated is now provided.

- The HUGIN KB file format has changed (in order to handle HKB files produced by 64-bit versions of the HUGIN API, among other things).

  There are a few user-visible changes: If a compiled (but not compressed) domain is saved as an HKB file, it will only be triangulated when loaded. A compilation is required before inference can be performed (see Section 2.10). Compressed domains are still loaded as compressed (which implies compiled), but a propagation is required before beliefs can be retrieved.

- Functions for converting between table indexes and state configurations are now provided: *h_table_get_index_from_configuration*(61) and *h_table_get_configuration_from_index*(61).

- A function to retrieve the CG size of a table is provided: *h_table_get_cg_size*(64).

## Overview of changes and new features in HUGIN API version 6.5

- Domains and nodes can now be constructed by cloning existing objects — see *h_domain_clone*(29) and *h_node_clone*(30).

- Default state labels for boolean nodes are now `false` and `true` — see *h_node_get_state_label*(77).[1]

---

[1] This was done in order to ensure compatibility between the different versions of the HUGIN API.

- A function for translating a state label to a state index is provided — see *h_node_get_state_index_from_label*(77). Also, a function for translating a state value to a state index is provided — see *h_node_get_state_index_from_value*(78).
- Functions for determining independence properties (also known as *d*-separation properties) are provided — see *h_domain_get_d_connected_nodes*(108) and *h_domain_get_d_separated_nodes*(108)
- The number of cases that can be handled using a given amount of memory has been doubled compared to previous releases. This has been made possible by limiting the number of different states that can be represented in case data to 32768 per node — see *h_node_set_case_state*(148).
- A function for entering a case as evidence is provided — see *h_domain_enter_case*(150).
- *h_domain_get_log_likelihood*(150) now computes the log-likelihood with respect to the current parameter values. It used to return the log-likelihood computed during the final iteration of the EM algorithm (implying that the log-likelihood was computed with respect to slightly outdated parameter values). The function no longer requires invoking the EM algorithm before use.
- Functions for computing the AIC and BIC scores are provided — see *h_domain_get_AIC*(151) and *h_domain_get_BIC*(151). AIC and BIC are scores of comparing model quality taking model complexity into account.
- The EM algorithm now reports the AIC and BIC scores (in addition to the log-likelihood score) to the log file after completion of parameter estimation.
- The formats of case files and data files have been extended: If the contents of a state label form a valid name, the quotes can be omitted. See Section 8.9 and Section 11.3.

## Overview of changes and new features in HUGIN API version 6.4

- The performance of inference (including the EM algorithm) has been improved.
- Functions for computing entropy and mutual information have been added: *h_node_get_entropy*(131) and *h_node_get_mutual_information*(131). These functions are useful for value of information analysis. See Section 9.9 for further details.
- New function: *h_domain_get_log_likelihood*(150).

- *h_domain_learn_tables*(157) and *h_domain_learn_class_tables*(161) now report the log-likelihood to the log-file (if it is non-NULL) after each iteration of the EM algorithm.
- Direct access to the pseudorandom number generator implemented in Hugin is now provided through the functions *h_domain_get_uniform_deviate*(129) and *h_domain_get_normal_deviate*(130).
- HUGIN API libraries for Windows platforms are now provided for Visual Studio .NET 2003 (in addition to Visual Studio 6.0).

### Overview of changes and new features in HUGIN API version 6.3

- A function for replacing the class of an instance node with another compatible class (i.e., the interface of the new class must be a superset of the interface of the class being replaced), while preserving all input and output relations associated with the instance node. This is useful when, for example, a new version of an instantiated class becomes available. See *h_node_substitute_class*(53).
- A function for replacing a parent of a node with another compatible node, while preserving the validity of existing tables and model associated with the child node. See *h_node_switch_parent*(32).
- The C++ HUGIN API is now available as both a single-precision and a double-precision library.

### Overview of changes and new features in HUGIN API version 6.2

- HUGIN KB files are now automatically compressed using the Zlib library (`www.zlib.net`). This change implies that the developer (i.e., the user of the HUGIN API) must explicitly link to the Zlib library, if the application makes use of HKB files. See Section 1.2.
- HUGIN KB files can now be protected by a password. The following new functions supersede old functions: *h_domain_save_as_kb*(45) and *h_kb_load_domain*(46).
- The EM learning algorithm can now be applied to object-oriented models. See the *h_domain_learn_class_tables*(161) function.
- Functions to save and load case data (as used by the learning algorithms — Section 11.1) have been added to the API. See Section 11.3.
- Functions to parse a list of node names stored in a text file are now provided. Such functions are useful for handling, e.g., collections of elimination orders for triangulations. See *h_domain_parse_nodes*(93) and *h_class_parse_nodes*(93).

- The HUGIN API Reference Manual is now provided as a hyperlinked PDF file.

## Overview of changes and new features in HUGIN API version 6.1

- The HUGIN API is now thread-safe. See Section 1.8 for further details.
- Functions to save and load cases have been added to the API. See Section 8.9.
- The heuristic used in the total-weight triangulation method for large networks has been improved.

## Overview of changes and new features in HUGIN API version 6

- Object-oriented models for belief networks and influence diagrams can now be constructed using HUGIN API functions (see Chapter 3). Also, NET language support for object-oriented models (including generation and parsing of NET specifications) is available (see Chapter 12).
- Support for API functions prior to Version 2 of the HUGIN API has been dropped.
- Loading of HKB files produced by API versions prior to version 5.0 has been dropped. If you have an old release, please save your domains using the NET format before upgrading.
- Some functions have been superseded by better ones: *h_domain_write_net* has been replaced by *h_domain_save_as_net*(181), *h_net_parse* has been replaced by *h_net_parse_domain*(178), and *h_string_to_expression* has been replaced by *h_string_parse_expression*(74). However, the old functions still exist in the libraries, but the functions should not be used in new applications.

## Overview of changes and new features in HUGIN API version 5.4

- A new triangulation method has been implemented. This method makes it possible to find a (minimal) triangulation with minimum sum of clique weights. For some large networks, this method has improved time and space complexity of inference by an order of magnitude (sometimes even more), compared to the heuristic methods provided by earlier versions of the HUGIN API.

  See Section 6.3 for more information.
- The computations used in the inference process have been reorganized to make better use of the caches in modern CPUs. The result is faster inference.

## Overview of changes and new features in HUGIN API version 5.3

- The structure learning algorithm now takes advantage of domain knowledge in order to constrain the set of possible networks. Such knowledge can be knowledge of the direction of an edge, the presence or absence of an edge, or both. See Section 11.5.

- A new operator ("Distribution") for specifying arbitrary finite discrete distributions has been introduced. This operator is only permitted for discrete variables (i.e., not interval variables).

- The discrete distributions (Binomial, Poisson, Negative Binomial, and Geometric) now also work for interval nodes.

- New functions for the table generator: the "floor" and "ceil" functions round real numbers to integers; the "abs" function computes the absolute value of a number, and the "mod" (modulo) function computes the remainder of a division.

- The HUGIN KB file format has changed (again), but version 5.3 of the HUGIN API will load HKB files produced by versions 3 or later (up to version 5.3). But note that support for older formats may be dropped in future versions of the HUGIN API.

## Overview of changes and new features in HUGIN API version 5.2

- An algorithm for learning the structure of a belief network given a set of cases has been implemented. See Section 11.4.

- Simulation in uncompiled domains (Section 9.8) is now permitted when the set of nodes with evidence form an *ancestral set* of instantiated nodes (i.e., no likelihood evidence is present, and if a chance node is instantiated, so are all of its parents). Decision nodes must, of course, be instantiated.

- If a domain is saved (as a HUGIN KB file) in compiled form, *h_kb_load_domain*(46) attempts to load it in that form as well. As the contents of the junction tree tables are not stored in the HKB file, the inference engine must be initialized from the user-specified tables and models. This can fail for various reasons (e.g., the tables and/or models contain invalid data). In this case, instead of refusing to load the domain, *h_kb_load_domain* instead returns the domain in uncompiled form.

- The $\log_2$, $\log_{10}$, sin, cos, tan, sinh, cosh, and tanh functions and the Negative Binomial distribution have been added to the table generation facility.

- The HUGIN KB file format has changed (again), but version 5.2 of the HUGIN API will load HKB files produced by versions 3 or later (up to version 5.2). But note that support for older formats may be dropped in future versions of the HUGIN API.

### Overview of changes and new features in HUGIN API version 5.1

- The simulation procedure has been extended to handle networks with continuous variables. Also, a method for simulation in uncompiled domains has been added. See Section 9.8.

- HUGIN will now only generate tables from a model when (the inference engine thinks) the generated table will differ from the most recently generated table. Such tests are now performed by the compilation, propagation, and reset-inference-engine operations (in previous versions of the HUGIN API, the compilation operation always generated all tables, and the propagation and reset-inference-engine operations never generated any tables).

  Also, tables can now be [re]generated individually on demand.

  See Section 5.8 for more information.

- The number of values to use per (bounded) interval of a (parent) interval node can now be specified on a per-model basis. This provides a way to trade accuracy for computation speed. See Section 5.9.

- Iterators for the attributes of nodes and domains are now provided. See Section 2.9.2.

- The HUGIN KB file format (the `.hkb` files) has changed. This was done in order to accommodate the above mentioned features.

  The HUGIN API version 5.1 will load HUGIN KB files produced by versions 3 or later (up to version 5.1). But note that support for older formats may be dropped in future versions of the HUGIN API.

- The NET language has been extended with a model attribute for specifying the number of values to use per (bounded) interval of a (parent) interval node.

  Also, if both a specification using the model attributes and a specification using the *data* attribute are provided, then the specification in the *data* attribute is used. Previous versions of the HUGIN API used the model in such cases.

  See Section 12.5.2 for more information.

## Overview of changes and new features in HUGIN API version 5

- A batch learning method (based on the EM algorithm) has been implemented. Given a set of cases and optional expert-supplied priors, it finds[2] the best unrestricted model matching the data and the priors.
- The sequential learning method (also known as *adaptation*) has been reimplemented and given a new API interface. (HUGIN API version 1.2 provided the first implementation of sequential learning.)
- The HUGIN KB file format (the `.hkb` files) has changed. This was done in order to accommodate adaptation information and evidence. Also, junction tree tables (for compiled domains) are not stored in the HUGIN KB file anymore.

  The HUGIN API version 5 will load HUGIN KB files produced by HUGIN API versions 3 or later (up to version 5).
- The NET language has been extended in order to accommodate adaptation information.
- A single-precision version of the HUGIN API is now able to load a HUGIN KB file created by a double-precision version of the HUGIN API — and vice versa.

## Overview of changes and new features in HUGIN API version 4.2

- The traditional function-oriented version of the HUGIN API has been supplemented by object-oriented versions for the Java and C++ language environments.
- The most time-consuming operations performed during inference have been made threaded. This makes it possible to speed up inference by having individual threads execute in parallel on multi-processor systems.
- The class of belief networks with CG nodes that can be handled by HUGIN has been extended. A limitation of the old algorithm has been removed by the introduction of the recursive combination operation (see [19] for details).
- Evidence entered to a domain is no longer removed when an (explicit or implicit) "uncompile" operation is performed. Also, evidence can be entered (and retracted) when the domain is not compiled. These

---

[2] The EM algorithm is an iterative method that searches for a maximum of a function. There is, however, no guarantee that the maximum is global. It might be a local maximum — or even a saddle point.

changes affect all functions that enter, retract, or query (entered) evidence, as well as *h_domain_uncompile*(94) and the functions that perform implicit "uncompile" operations — with the exception of *h_node_set_number_of_states*(35) which still removes the entered evidence.

## Overview of changes and new features in HUGIN API version 4.1

- An "arc-reversal" operation is provided: This permits the user to reverse the direction of an edge between two chance nodes of the same kind, while at the same time preserving the joint probability distribution of the belief network or influence diagram.
- A "Noisy OR" distribution has been added to the table generation facility (Chapter 5).
- Support for C compilers that don't conform to the ISO Standard C definition has been dropped.

## Overview of new features in HUGIN API version 4

- Version 4 of the HUGIN API makes it possible to generate conditional probability and utility potentials based on mathematical descriptions of the relationships between nodes and their parents. The language provided for such descriptions permits both deterministic and probabilistic relationships to be expressed.

  This facility is implemented as a front end to the HUGIN inference engine: The above mentioned descriptions only apply to discrete nodes, implying that continuous distributions (such as the gamma distribution) are discretized. Thus, inference with such distributions are only approximate. The only continuous distributions for which the HUGIN API provides exact inference are the CG distributions.

  See Chapter 5 for further details.
- The table for a node is no longer deleted when a parent is removed or the number of states is changed (either for the node itself or for some parent). Instead, the table is resized (and the contents updated).

  This change affects the following functions: *h_node_remove_parent*(32), *h_node_set_number_of_states*(35), and *h_node_delete*(30) (since deletion of a node implies removing it as a parent of its children).

## Overview of new features in HUGIN API version 3

- Version 3 of the HUGIN API introduces belief networks with *Conditional Gaussian* (CG) nodes. These represent variables with a Gaussian (also known as a 'normal') distribution conditional on the values of their parents. The inference is exact (i.e., no discretization is performed).

- It is no longer required to keep a copy of the initial distribution stored in a disk file or in memory in order to initialize the inference engine. Instead, the initial distribution can be computed (when needed) from the conditional probability and utility tables.
- It is now possible for the user to associate attributes (key/value pairs) with nodes and domains. The advantage over the traditional user data (as known from previous versions of the HUGIN API) is that these attributes are saved with the domain in both the NET and the HUGIN KB formats.
- It is no longer necessary to recompile a domain when some conditional probability or utility potential has changed. When HUGIN notices that some potential has changed, the updated potential will be taken into account in subsequent propagations.
- It is now possible to reorganize the layout of conditional probability and utility potentials (Section 4.5).
- The HUGIN API is now provided in two versions: a (standard) version using single-precision floating-point arithmetic and a version using double-precision floating-point arithmetic. The double-precision version may prove useful in computations with continuous random variables (at the cost of a larger space requirement).

## Overview of new features in HUGIN API version 2

- Version 2 of the HUGIN API introduces influence diagrams. An *influence diagram* is a belief network augmented with decisions and utilities. Edges in the influence diagram into a random variable represents probabilistic dependencies while edges into a decision variable represents availability of information at the time the decision is taken. Assuming a total order of the decisions, an optimal decision policy using maximization of expected utility for choosing between decision alternatives can be computed.

  Version 2 of the HUGIN API allows specification of and inference and decision making with influence diagrams. This version will also take advantage of the case where the overall utility is a sum of a set of local utilities.
- New propagation methods: (1) In addition to the well-known 'sum'-propagation method, a 'max'-propagation method that identifies the most probable configuration of all variables and computes its probability is introduced, and (2) a new way to incorporate evidence, known

as 'fast-retraction', permits the computation, for each variable, of the conditional probability of that variable given evidence on the remaining variables (useful for identifying suspicious findings). Thus, four different ways of propagating evidence are now available.

- Models with undirected edges, so-called 'chain graph' models, are now permitted. This extends the class of models so that automatically generated models are more easily used with HUGIN (in automatically generated models, the direction of an association is only rarely identified). Chain graph models are currently only available via NET specifications (Chapter 12).
- Extraction of the joint probability distribution for a group of variables, even when the group is not a subset of any clique, is now possible.
- Version 2 of the HUGIN API allows construction and editing of belief networks and influence diagrams.
- Analysis of data conflicts, previously only available within the HUGIN GUI application, is now also available via the API.
- Simulation: given evidence, a configuration for all variables can be sampled according to the distribution determined by the evidence.
- The interface of the API has undergone a major clean-up and redesign. The naming has been made more consistent: a common prefix *h_* is introduced, all functions operating on the same type of object has a common prefix (e.g., all functions with a node as 'primary' argument shares the common prefix *h_node_*)
- The concept of a 'current' or 'selected' domain has been removed. The domain to be operated upon is now an explicit argument.
- Backwards compatibility: Application programs built using the documented functions and types of previous versions of the HUGIN API can still be compiled and should work as expected, although use of these older functions and types in new applications is strongly discouraged.

# Bibliography

[1] S. K. Andersen, K. G. Olesen, F. V. Jensen, and F. Jensen. HUGIN — a shell for building Bayesian belief universes for expert systems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1080–1085, Detroit, Michigan, Aug. 20–25, 1989. Reprinted in [28].

[2] A. Berry, J.-P. Bordat, and O. Cogis. Generating all the minimal separators of a graph. *International Journal of Foundations of Computer Science*, 11(3):397–403, Sept. 2000.

[3] V. Bouchitté and I. Todinca. Treewidth and minimum fill-in: Grouping the minimal separators. *SIAM Journal on Computing*, 31(1):212–232, July 2001.

[4] H. Chan and A. Darwiche. When do numbers really matter? *Journal of Artificial Intelligence Research*, 17:265–287, 2002.

[5] V. M. H. Coupé and L. C. van der Gaag. Properties of sensitivity analysis of Bayesian belief networks. *Annals of Mathematics and Artificial Intelligence*, 36(4):323–356, Dec. 2002.

[6] R. G. Cowell and A. P. Dawid. Fast retraction of evidence in a probabilistic expert system. *Statistics and Computing*, 2(1):37–40, Mar. 1992.

[7] R. G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Statistics for Engineering and Information Science. Springer-Verlag, New York, 1999.

[8] A. P. Dawid. Applications of a general propagation algorithm for probabilistic expert systems. *Statistics and Computing*, 2(1):25–36, Mar. 1992.

[9] F. Jensen and S. K. Andersen. Approximations in Bayesian belief universes for knowledge-based systems. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence*, pages 162–169, Cambridge, Massachusetts, July 27–29, 1990.

[10] F. V. Jensen, B. Chamberlain, T. Nordahl, and F. Jensen. Analysis in HUGIN of data conflict. In P. P. Bonissone, M. Henrion, L. N. Kanal, and J. F. Lemmer, editors, *Uncertainty in Artificial Intelligence*, volume 6, pages 519–528. Elsevier Science Publishers, Amsterdam, The Netherlands, 1991.

[11] F. V. Jensen, S. L. Lauritzen, and K. G. Olesen. Bayesian updating in causal probabilistic networks by local computations. *Computational Statistics Quarterly*, 5(4):269–282, 1990.

[12] F. V. Jensen and T. D. Nielsen. *Bayesian Networks and Decision Graphs*. Information Science and Statistics. Springer-Verlag, New York, second edition, 2007.

[13] F. V. Jensen, K. G. Olesen, and S. K. Andersen. An algebra of Bayesian belief universes for knowledge-based systems. *Networks*, 20(5):637–659, Aug. 1990. Special Issue on Influence Diagrams.

[14] U. Kjærulff. Triangulation of graphs — algorithms giving small total state space. Research Report R-90-09, Department of Mathematics and Computer Science, Aalborg University, Denmark, Mar. 1990.

[15] U. B. Kjærulff and A. L. Madsen. *Bayesian Networks and Influence Diagrams: A Guide to Construction and Analysis*. Information Science and Statistics. Springer-Verlag, New York, 2008.

[16] S. L. Lauritzen. Propagation of probabilities, means, and variances in mixed graphical association models. *Journal of the American Statistical Association (Theory and Methods)*, 87(420):1098–1108, Dec. 1992.

[17] S. L. Lauritzen. The EM algorithm for graphical association models with missing data. *Computational Statistics & Data Analysis*, 19(2):191–201, Feb. 1995.

[18] S. L. Lauritzen, A. P. Dawid, B. N. Larsen, and H.-G. Leimer. Independence properties of directed Markov fields. *Networks*, 20(5):491–505, Aug. 1990. Special Issue on Influence Diagrams.

[19] S. L. Lauritzen and F. Jensen. Stable local computation with conditional Gaussian distributions. *Statistics and Computing*, 11(2):191–203, Apr. 2001.

[20] S. L. Lauritzen and D. Nilsson. Representing and solving decision problems with limited information. *Management Science*, 47(9):1235–1251, Sept. 2001.

[21] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society, Series B (Methodological)*, 50(2):157–224, 1988. Reprinted in [28].

[22] A. L. Madsen, F. Jensen, U. B. Kjærulff, and M. Lang. The HUGIN tool for probabilistic graphical models. *International Journal on Artificial Intelligence Tools*, 14(3):507–543, June 2005.

[23] A. L. Madsen, M. Lang, U. B. Kjærulff, and F. Jensen. The HUGIN tool for learning Bayesian networks. In T. D. Nielsen and N. L. Zhang, editors, *ECSQARU 2003*, volume 2711 of *LNAI*, pages 594–605, Aalborg, Denmark, July 2–5, 2003. Springer-Verlag.

[24] K. G. Olesen, S. L. Lauritzen, and F. V. Jensen. aHUGIN: A system creating adaptive causal probabilistic networks. In D. Dubois, M. P. Wellman, B. D'Ambrosio, and P. Smets, editors, *Proceedings of the Eighth Conference on Uncertainty in Artificial Intelligence*, pages 223–229, Stanford, California, July 17–19, 1992. Morgan Kaufmann, San Mateo, California.

[25] K. G. Olesen and A. L. Madsen. Maximal prime subgraph decomposition of Bayesian networks. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 32(1):21–31, Feb. 2002.

[26] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, California, 1988.

[27] J. Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, Cambridge, UK, 2000.

[28] G. Shafer and J. Pearl, editors. *Readings in Uncertain Reasoning*. Morgan Kaufmann, San Mateo, California, 1990.

[29] K. Shoikhet and D. Geiger. A practical algorithm for finding optimal triangulations. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 185–190, Providence, Rhode Island, July 27–31, 1997. AAAI Press, Menlo Park, California.

[30] D. J. Spiegelhalter and S. L. Lauritzen. Sequential updating of conditional probabilities on directed graphical structures. *Networks*, 20(5):579–605, Aug. 1990. Special Issue on Influence Diagrams.

[31] P. Spirtes, C. Glymour, and R. Scheines. *Causation, Prediction, and Search*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, Massachusetts, second edition, 2000.

[32] D. Vose. *Risk Analysis: A Quantitative Guide*. Wiley, Chichester, UK, second edition, 2000.

# Index