SMILE: Structural Modeling, Inference, and Learning Engine

Programmer's Manual

Version 2.2.4.R1, Built on 4/27/2024
BayesFusion, LLC

This page is intentionally left blank.

This page is intentionally left blank.

Introduction

1 Introduction

Welcome to SMILE Programmer's Manual, version 2.2.4.R1, built on 4/27/2024. For the most recent version of this manual, please visit https://support.bayesfusion.com/docs/.

To download the software described in this manual, please visit https://download.bayesfusion.com. The source code of SMILE is proprietary. If the operating system and/or compiler that you want to use SMILE with is not on the list of binaries available at our software download website, please contact us.

SMILE (Structural Modeling, Inference, and Learning Engine) is a software library for performing Bayesian inference, written in C++, available in compiled form for a variety of platforms, including multiple versions of Visual C++ for 32-bit and 64-bit Windows, macOS and iOS running on Intel and ARM, and Linux. We assume that the reader has a basic knowledge of the C++ programming language. We also provide wrappers exposing SMILE functionality to programs written in Java (jSMILE), Python (PySMILE), R (rSMILE), .NET (Smile.NET), or using COM (SMILE.COM, targeted for use with Microsoft Excel). However, this manual is for C++ programmers and does not cover the interoperability issues, except for MATLAB (see *Appendix M: MATLAB and SMILE* 218). Wrapper documentation is provided in a separate manual.

If you are new to SMILE and would like to start with an informal, tutorial-like introduction, please start with the *Hello SMILE!* 22 section. If you are an advanced user, please browse through the Table of Contents or search for the topic of your interest.

This manual refers to a good number of concepts that are assumed to be known to the reader, such as probability, utility, decision theory and decision analysis, Bayesian networks, influence diagrams, etc. Should you want to learn more about these, please refer to GeNIe manual. SMILE is GeNIe's Application Programmer's Interface (API) and practically every elementary operation performed with GeNIe translates to calls to SMILE methods. Being familiar with GeNIe may prove extremely useful in learning SMILE. Understanding some of SMILE’s functionality may be easier when performed interactively in GeNIe. GeNIe manual, along with all other BayesFusion documentation, is available at https://support.bayesfusion.com/docs. Other resources, including introduction to probabilistic graphical models, are available at https://www.bayesfusion.com/resources/.

Yet another useful resource is available at https://repo.bayesfusion.com. The site is powered by BayesBox, our interactive model viewer/repository software. The repository contains more than 100 example Bayesian networks, hybrid Bayesian networks, dynamic Bayesian networks, and influence diagrams. BayesBox runs SMILE on the server side and calls into its API to calculate the posterior probabilities after evidence is modified in the web browser.

Licensing

2 Licensing

SMILE library is a commercial product that requires a development license to use. There are two types of development licenses: Academic and Commercial. Academic license is free of charge for research and teaching use by those users affiliated with an academic institution. All other use requires a commercial license, available for purchase from BayesFusion, LLC.

Deployment of SMILE library, i.e., embedding it into user programs, requires a deployment license. There are two types of deployment licenses: Server license, which allows a program linked with SMILE to be deployed on a computer server, and end-user program license, which allows distribution of user programs that include SMILE. Please contact BayesFusion, LLC, for details of the licenses and pricing.

The licensing system is implemented as two variables (`DSL_LIC1` and `DSL_LIC2`), which must be declared and initialized in order to successfully link your program with SMILE. The definitions for these variables are included in your license key header file (usually `smile_license.h`). **The license key header file is not included in SMILE distribution**, it is personalized by BayesFusion, LLC, for you or your organization. Six-month academic and free 30-day evaluation license keys can be obtained directly at https://download.bayesfusion.com.

Six-month academic license should be sufficient for most coursework. If you need SMILE Academic for a research project and would like a longer license, please email us at support@bayesfusion.com from your university email account.

What's new in SMILE 2

3 What's new in SMILE 2

Major new features in SMILE 2 are:

- Support for metalog distributions in continuous nodes. There is no new API for metalog distributions, equation nodes can use `Metalog` and `MetalogA` functions in their equations. For details, see the metalog entry in the *Random Number Generators* 199 section.
- Discrete nodes with outcomes based on numeric intervals or point values. See the *Discrete nodes and numeric domains* 39 section for details.
- Qualitative De Morgan nodes, documented in the *Qualitative models* 41 section.
- Diagnosis API, documented in the *Diagnosis* 53 section.
- C++ 11 support. It is now possible to use range-based for loops with SMILE arrays and matrices. The arrays and matrices can be intialized with `std::initializer_list`. Some frequently used methods, like `DSL_nodeDef::SetDefinition` can also directly use `std::initializer_list` as input.
- Two new method overloads were added to enable quick prototyping: `DSL_network::GetNode(const char *nodeId)` and `DSL_nodeVal::SetEvidence(const char *outcomeId)`. It is now possible to set evidence by specifying a pair of string identifiers (node identifier and outcome identifier) in one line: `net.GetNode("nodeId")->Val()->SetEvidence("outcomeId")`.
- New template methods were added to DSL_node class in order to eliminate the need to explicitly `static_cast` the returned pointer to the correct node definition/node value class. For more details, see *Node definition* 32 and *Node value & evidence* 33 sections.
- Shorter and more consistent API names. One of the SMILE 2 development goals was to make the user code shorter and more consistent. In order to achieve that goal, we changed some class and method names. To keep the backward compatibility as close to 100% as possible, SMILE 2.x uses typedefs and inline methods to ensure that SMILE 1.x-based programs compile and run correctly. The backward compatibility is enabled by default. It can be suppressed by defining the `SMILE_NO_V1_COMPATIBILITY` pre-processor macro before `#include "smile.h"`. For new projects, we recommend blocking the backward compatibility.

The table below lists the changed names.

SMILE 1.x	SMILE 2
`DSL_node::Definition`	`DSL_node::Def`
`DSL_node::Value`	`DSL_node::Val`
`DSL_nodeDefinition`	`DSL_nodeDef`
`DSL_nodeDefinition::GetOutcomesNames`	`DSL_nodeDef::GetOutcomeIds`

SMILE 1.x	SMILE 2
DSL_nodeValue	DSL_nodeVal
DSL_list	DSL_decision
DSL_table	DSL_utility
DSL_valEqEvaluation	DSL_equationEvaluation
DSL_simpleCase	DSL_case

This page is intentionally left blank.

Compiler and linker options

4 Compiler and linker options

SMILE is distributed as a C++ library along with a set of header files. To compile your program, you need to include the `smile.h` header file. You also need to ensure that your binary is linked with SMILE.

BayesFusion, LLC can provide binaries built with settings different from these described below on request.

4.1 Visual C++

We support compiled SMILE library for multiple versions of Visual C++. Each zip file in our download repository contains libraries for both x86 (32-bit) and x64 (64-bit) architectures. In addition, for each architecture, we provide three libraries built to use with different versions of Visual C++ runtime environment. For example, SMILE for Visual Studio 2015 includes the following libraries:

- `smile_dbg_vc_140x64.lib`: debug build for dynamic debug CRT, 64-bit
- `smile_dbg_vc_140x86.lib`: debug build for dynamic debug CRT, 32-bit
- `smile_vc_140x64.lib`: release build for static CRT, 64-bit
- `smile_vc_140x86.lib`: release build for static CRT, 32-bit
- `smile_dyn_vc_140x64.lib`: release build for dynamic (DLL) CRT, 64-bit
- `smile_dyn_vc_140x86.lib`: release build for dynamic (DLL) CRT, 32-bit

The number '140' in the list of libraries above corresponds to Microsoft's internal toolkit version, which is 140 for Visual C++ included in the Visual Studio 2015 product.

Only one of these libraries needs to be linked with your executable. The `smile.h` header contains `#pragma` directives which automatically select the proper library depending on the selected architecture and current build configuration of your project.

NOTE: due to auto-linking implemented in smile.h, **you do not need to add `smile*.lib` files** to the list of libraries in your project's linker settings. Doing that is likely to cause linker errors.

However, you still need to tell the linker the location of the directory containing SMILE's .lib files. There are two independent ways to do that:

1. Go to Project Settings | Linker | General and add SMILE directory to 'Additional Library Directories', or
2. Go to Project Settings | VC++ Directories and add SMILE directory to 'Library Directories'.

There are also two independent ways of specifying the SMILE include directory:

1. Go to Project Settings | C++ | General and add SMILE directory to 'Additional Include Directories', or

2. Go to Project Settings | VC++ Directories and add SMILE directory to 'Include Directories'.

Starting with version 1.2.0 released in November 2017, SMILE no longer requires `_SECURE_SCL=0` to be defined in pre-processor settings.

4.2 gcc and clang

SMILE for Unix-based and Unix-like systems (Linux, FreeBSD) is compiled with the gcc toolchain. SMILE for Apple operating systems (iOS and macOS) is compiled with Apple clang. In both cases, we compile public binaries with `-O3` and `-DNDEBUG`.

To use SMILE you will need the following on your g++ or clang command line:

- Specify directory containing `libsmile.a` using -L option: `-L<smiledir>`
- Specify directory containing `smile.h` using -I (uppercase I) option: `-I<smiledir>`
- Add `libsmile.a` to libraries used by your program with -l (lowercase L) option: `-lsmile`. Note that the 'lib' prefix and '.a' suffix are implied.

Example:

We assume that SMILE's headers and libraries are located in the subdirectory 'smile' and your source code is in one or more files with the .cpp extension.

```
g++ -DNDEBUG -O3 *.cpp -I./smile -L./smile -lsmile
```

This command compiles your program and then links it with `libsmile.a`. The output executable binary has the default filename `a.out`. Note that `-lsmile` should be placed after `*.cpp` due to the g++ toolchain linker behavior (linker searches command line parameters from left to right).

To use C++ 11 support in SMILE, add the `-std=c++11` (or a C++ standard later than C++11) option. This is required for `std::initializer_list` arguments in methods like `DSL_nodeDef::SetDefinition`.

The archive (.tar.gz) with SMILE binaries includes the `build.txt` file, which contains the details of the build environment used to compile the library, including the version of the compiler. Please make sure you are using the library compatible with your compiler.

This page is intentionally left blank.

Hello, SMILE!

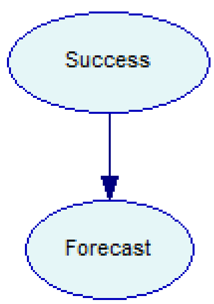
5 Hello, SMILE!

In this section, we will show how SMILE can load and use a model created in GeNIe to perform useful work. We will use the model developed in GeNIe manual (Section *Hello GeNIe!*). The model for this problem is available as one of the example networks (file `VentureBN.xdsl`). If you have GeNIe installed, you can copy the file into your working directory. The same file is also included in the zip file containing all source code for tutorials, available from http://support.bayesfusion.com/docs. Alternatively, create a file named `VentureBN.xdsl` with any text editor by copying the content of the *VentureBN.xdsl* [24] section below.

The complete source code of the program is also provided in this chapter in the *hello.cpp* [23] section.

5.1 Success/Forecast model

The model encodes information pertaining to a problem faced by a venture capitalist, who considers a risky investment in a startup company. A major source of uncertainty about her investment is the success of the company. She is aware of the fact that only around 20% of all start-up companies succeed. She can reduce this uncertainty somewhat by asking expert opinion. Her expert, however, is not perfect in his forecasts. Of all start-up companies that eventually succeed, he judges about 40% to be good prospects, 40% to be moderate prospects, and 20% to be poor prospects. Of all start-up companies that eventually fail, he judges about 10% to be good prospects, 30% to be moderate prospects, and 60% to be poor prospects.



Success	0.2
Failure	0.8

	Success	Failure
Good	0.4	0.1
Moderate	0.4	0.3
Poor	0.2	0.6

5.2 The program

We will show how to load this model using SMILE, how to enter observations (evidence), how to perform inference, and how to retrieve the results of SMILE's calculations. The complete source code is included below. Note that you will need to #include your SMILE license key. See the *Licensing* [12] section of this manual if you want to obtain your academic or trial license key.

The program starts with redirecting the error and warning messages to the standard output. We do not expect to see any messages. If VentureBN.xdsl is not in the current directory, you will get notified.

```
DSL_errorH().RedirectToFile(stdout);
```

Our network object is declared as local variable, then we read the file. We proceed only if the file loaded correctly.

```
DSL_network net;
```

```
int res = net.ReadFile("VentureBN.xdsl");
if (DSL_OKAY != res)
{
    return res;
}
```

For clarity, we will not be checking the return status codes in the remaining part of the program. We assume that VentureBN model contains a *Forecast* node with a *Moderate* outcome, and a *Success* node. While SMILE 1.x required a function call to convert a textual node/outcome identifier to an integer handle/index, SMILE 2 can use strings directly:

```
net.GetNode("Forecast")->Val()->SetEvidence("Moderate");
```

Note that `DSL_network::GetNode` will return `NULL` if there is no node with the specified identifier. Similarly, `DSL_nodeVal::SetEvidence` will return a nonzero error code if the node does not have the outcome with specified identifier.

`DSL_network::UpdateBeliefs` performs inference in the network.

```
net.UpdateBeliefs();
```

After network update, we can read the posterior probabilities of the *Success* node. We again convert node identifier to handle and iterate over its outcomes, displaying the probability of each outcome in the loop.

```
DSL_node* sn = net.GetNode("Success");
const DSL_Dmatrix& beliefs = *sn->Val()->GetMatrix();
const DSL_idArray& outcomes = *sn->Def()->GetOutcomeIds();
for (int i = 0; i < outcomes.GetSize(); i++)
{
    printf("%s=%g\n", outcomes[i], beliefs[i]);
}
```

If you compile and run this program, the output that you should see is the following:

```
Success=0.25
Failure=0.75
```

'Success' and 'Failure' are outcomes of the *Success* node.

We will build upon the simple network described in this chapter in the *Tutorials* 70 section of this manual.

5.3 hello.cpp

```
// hello.cpp

#include <cstdio>
#include <smile.h>
#include "smile_license.h" // your licensing key

int main()
{
    DSL_errorH().RedirectToFile(stdout);
    DSL_network net;
    int res = net.ReadFile("VentureBN.xdsl");
    if (DSL_OKAY != res)
    {
        return res;
```

```
    }

    net.GetNode("Forecast")->Val()->SetEvidence("Moderate");
    net.UpdateBeliefs();
    DSL_node* sn = net.GetNode("Success");
    const DSL_Dmatrix& beliefs = *sn->Val()->GetMatrix();
    const DSL_idArray& outcomes = *sn->Def()->GetOutcomeIds();
    for (int i = 0; i < outcomes.GetSize(); i++)
    {
        printf("%s=%g\n", outcomes[i], beliefs[i]);
    }

    return DSL_OKAY;
}
```

5.4 VentureBN.xdsl

```
<?xml version="1.0" encoding="UTF-8"?>
<smile version="1.0" id="VentureBN" numsamples="1000" discsamples="10000">
	<nodes>
		<cpt id="Success">
			<state id="Success" />
			<state id="Failure" />
			<probabilities>0.2 0.8</probabilities>
		</cpt>
		<cpt id="Forecast">
			<state id="Good" />
			<state id="Moderate" />
			<state id="Poor" />
			<parents>Success</parents>
			<probabilities>0.4 0.4 0.2 0.1 0.3 0.6</probabilities>
		</cpt>
	</nodes>
	<extensions>
		<genie version="1.0" app="GeNIe 4.0.2405" name="Venture">
			<node id="Success">
				<name>Success of the venture</name>
				<interior color="e5f6f7" />
				<outline color="000080" />
				<font color="000080" name="Arial" size="10" />
				<position>63 25 145 76</position>
				<barchart active="true" width="171" height="64" />
			</node>
			<node id="Forecast">
				<name>Expert forecast</name>
				<interior color="e5f6f7" />
				<outline color="000080" />
				<font color="000080" name="Arial" size="10" />
				<position>64 150 150 203</position>
				<barchart active="true" width="168" height="80" />
			</node>
		</genie>
	</extensions>
</smile>
```

Using SMILE

6 Using SMILE

6.1 Main header file

To use SMILE, you need to include single header file, namely `smile.h`. As described earlier, with Visual C++ this file also takes care of specifying the correct variant of the library (debug vs. release with static CRT vs. release with CRT in DLL).

For new SMILE-based projects, we recommend adding `#define SMILE_NO_V1_COMPATIBILITY` before `#include "smile.h"`. This will disable SMILE 1.x compatibility features and can make IDE features like autocomplete work better with SMILE. For more information about SMILE 2 backward compatibility refer to the *What's new in SMILE 2* 14 section.

The second required header, usually named `smile_license.h` contains your licensing information. You need to include this file in exactly one of your .cpp files. See the *Licensing* 12 section for more details, including information on how to obtain evaluation or academic license from our website.

6.2 Error handling

Most SMILE functions and methods return an integer status code. Negative numbers are used for specific error codes. For a complete list of codes, see the *Error codes* 115 section in the *Reference Manual* 114.

Sometimes SMILE emits a warning or an error message. By default, your program is not notified about these messages: you need to query the information stored in a global instance of `DSL_errorStringHandler`, accessible through the `DSL_errorH` function. It is also possible to redirect the error output to FILE* (including stdout or stderr):

```
DSL_errorH().RedirectToFile(stdout);
```

For the complete control over warnings and error messages, use `DSL_errorStringHandler::Redirect` method. Its only argument of this method is a pointer to the instance of the class derived from `DSL_errorStringRedirect`.

```
class MyRedirect : public DSL_errorStringRedirect
{
public:
    void LogError(int code, const char *message)
    {
            // do something with the code and message
    }
};
// …
MyRedirect myRedir;
DSL_errorH().Redirect(&myRedir);
```

Tutorials included in this manual redirect errors to `stdout`.

6.3 Multithreading

SMILE does not perform any locking on its objects. A multithreaded application can use SMILE from more than one thread, but should provide appropriate synchronization. In practice, this translates to "use each `DSL_network` from a single thread at a time". Random number generators used in SMILE do not share any global state.

6.4 Unicode support

SMILE treats all text as null-terminated byte strings encoded as UTF-8. The default I/O format (.xdsl) is saved as UTF-8 encoded XML.

If you are using SMILE on Windows and your project is configured with "Use Unicode character set", and the string values are not known at compile time, you should convert between native Windows WCHAR strings encoded as UTF-16 and SMILE's UTF-8 with `WideCharToMultiByte`. The conversion from strings in SMILE to WCHAR should be performed with `MultiByteToWideChar`. Note that this conversion is required also for filename arguments in SMILE's I/O functions like `DSL_network::ReadFile` (SMILE converts its UTF-8 filename back to native Windows WCHAR). On the other hand, if the string is known and does not include characters with codepoints above 127, you can directly use `char[]` literal: `net.WriteFile("network1.xdsl")`.

6.5 Naming conventions and identifiers

The names of all publicly accessible SMILE classes and functions start with the `DSL_` prefix and use camel case afterwards. For example:

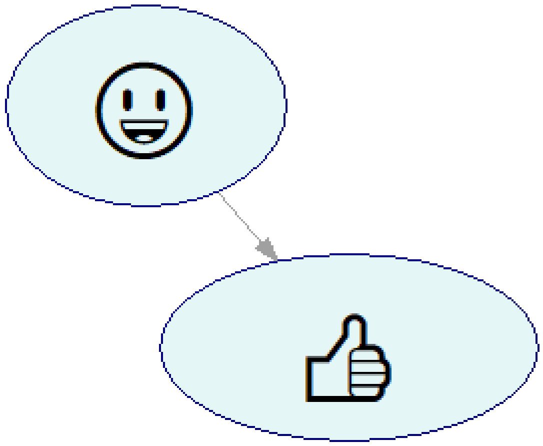
```
DSL_network
DSL_node
```

Constants and #defined symbols start with the same `DSL_` prefix followed by all-uppercase letters, for example:

```
DSL_OKAY
DSL_CPT
```

SMILE requires nodes in the network to have unique textual identifiers (in addition to potentially non-unique names and descriptions). A valid identifier starts with a letter and contains letters, decimal digits, and underscores. Rules for identifier syntax are also used in other parts of the SMILE API. For example, the names of the *custom functions* 215 and *user properties* 52 must be valid identifiers.

The letter in the identifier is defined as characters from a-z and A-Z range, but also any Unicode character with the codepoint greater than 127 (the Unicode character category is not checked). This allows for identifiers containing mathematical symbols, letters from alphabets other than Latin, or even emojis. Of course the program that renders the graphical view of your network must be able to use Unicode. Here is how a simple two-node network with emoji node ids may look in GeNIe:



6.6 Arrays

The predecessors of SMILE were developed in academia (Decision Systems Laboratory, University of Pittsburgh) before the STL became a part of the C++ standard. The library always had its own classes representing arrays of integers, doubles and strings. To keep the backward compatibility, we retained array classes in SMILE 2, making their public interfaces more consistent.

Numeric arrays are derived from the common `DSL_numArray` template, which implements a contiguous buffer for a specified numeric type. An array can be resized, and the storage for it can be reserved in advance to avoid large number of heap reallocations, just like `std::vector`. `DSL_numArray` also implements small object optimization by including an inline buffer for s specified (small) number of elements. Two widely used specializations of `DSL_numArray` are `DSL_intArray` and `DSL_doubleArray`. All three are documented in the reference section (start from `DSL_numArray` 119.) The method names are self-explanatory: `Contains` returns `true` when array contains a specified value, `SetSize` resizes the array, `Insert` inserts the element at the specific index, etc. Access to array elements is performed through `operator[]`.

SMILE 2.x adds interoperability with `std::vector`: `DSL_numArray::CopyFrom` and `CopyTo` provided to avoid manual loops copying between `DSL_numArray` and `std::vector`. The `DSL_numArray::begin` and `end` make the SMILE arrays compatible with the algorithms in C++ Standard Library, and with range-based foor loops if C++11 (or later) is enabled. `std::initializer_list` can be used as an input parameter to `DSL_numArray` constructor or `operator=`.

SMILE 2.x fixes the subtle API semantics issue related to the previous implementation of `GetSize` and `NumItems`. If the `SMILE_NO_V1_COMPATIBILITY` is not #defined, then `NumItems` is still available, but simply calls `GetSize`. Otherwise `NumItems` is no longer available.

In addition to numeric arrays, SMILE also defines the `DSL_stringArray` and `DSL_idArray`. These classes represent sequences of strings; their API is very similar to the DSL_numArray. Both are described in detail in the reference section.

6.7 Networks, nodes and arcs

The most important class defined by SMILE is `DSL_network`.

The objects of this class represent directed acyclic graphs (DAGs). They act as containers for nodes and are responsible for node creation and destruction. Nodes and arcs are always created and destroyed by invoking `DSL_network`'s methods.

Nodes (DAG vertices) are created by the `DSL_network::AddNode` method and destroyed by `DSL_network::DeleteNode`.

Arcs (DAG edges) are created by the `DSL_network::AddArc` method and destroyed by `DSL_network::RemoveArc`.

6.7.1 Network

To create an instance of the `DSL_network`, simply use the default constructor.

```
DSL_network myNetwork;
```

You can create `DSL_network` as a local variable on the stack, an object on the heap, or a member of another class, depending on your specific needs. Programs will typically call the `ReadFile` or `ReadString` methods to initialize the content of the network after creating it. The `UpdateBeliefs` method invokes the inference algorithm on the network and sets the node values. The `CalcProbEvidence` method computes the probability of evidence currently set in the network.

The objects of `DSL_network` class can be copied with the copy constructor and assigned with `operator=`.

6.7.2 Nodes

Nodes are represented by `DSL_node` objects. Your program should always use the `DSL_network::AddNode` method to create a node within the network and never use `DSL_node` constructor directly. To delete a node, use `DSL_network::DeleteNode`; never try to use `delete` on an `DSL_node` pointer.

The node type specified during node creation can be changed later with `DSL_node::ChangeType`. The type change modify neither node pointer nor its handle. However, node definition and value objects representing the old type are destroyed and recreated to reflect the new type.

Within a network, nodes are uniquely identified by their handles. A node handle is a non-negative integer, which is preserved when the network is copied using a copy constructor or `operator=`. Most of `DSL_network` methods dealing with nodes uses node handles as input arguments. In general, node handles may change when you write the network to file and read it later. This means that you should not rely on handles as persistent identifiers.

Values of handles are not guaranteed to be consecutive or start from any particular value. To iterate over nodes in a network, please use `DSL_network::GetFirstNode` and `GetNextNode`:

```
for (int h = net.GetFirstNode(); h >= 0; h = net.GetNextNode(h))
{
    DSL_node *node = net.GetNode(h);
    printf("Node handle: %d, node id: %s\n", h, node->GetId());
}
```

Note the loop exit condition - we stop when `GetNextNode` returns a negative value. It is also possible to get all node handles copied into a `DSL_intArray` object by means of `GetAllNodes`.

`DSL_network::PartialOrdering` returns a reference to a `DSL_intArray` containing all node handles in the network partially ordered (ancestor handles always precede their descendants).

In addition to a handle, each node has an unique (in the context of its containing network), persistent, textual identifier. This identifier is specified as an argument to `DSL_network::AddNode` method at node creation, and may be changed later with `DSL_node::SetId`. Identifiers in SMILE are case-sensitive, start with a letter and contain letters, digits, and underscores.

A node identifier can be translated to a node handle with a call to `DSL_network::FindNode` method.

```
int handle = net.FindNode("myNodeId");
if (handle >= 0)
{
    printf("Handle of myNodeId is: %d\n", handle);
}
else
{
    printf("There's no node with ID=myNodeId\n");
}
```

Note that `FindNode` has O(n) complexity, as it simply compares its input argument with all node identifiers in the loop. On the other hand, `GetNode` is O(1), as it performs an index lookup in the array. `GetNode` also has an overload, which accepts a node identifier as input, and performs O(n) linear search, returning `NULL` if there is no node with the specified identifier.

```
DSL_node *node = net.GetNode("myNodeId");
if (NULL == node)
{
    printf("Node 'myNodeId' doest not exist.\n");
}
```

To get all node identifiers into a `DSL_idArray` object, use `GetAllNodeIds`.

Nodes may be marked as targets with `DSL_network::SetTarget` method. Target nodes are always guaranteed to be updated by the inference algorithm. Other nodes, i.e., nodes that are not designated as targets, may be updated or not, depending on the internals of the algorithm used, but are not guaranteed to be updated. Focusing inference on the target nodes can reduce time and memory required to complete the calculation. When no targets are specified, SMILE assumes that all nodes are of interest to the user.

A node can be marginalized. The operation removes it from the network, and modifies the definitions of the remaining nodes, so that the joint probability distribution over the remaining variables in the network is unchanged.

6.7.3 Arcs

SMILE does not define a class representing an arc between nodes. When you call `DSL_network::AddArc` method, the internal data structures are updated to keep the relationship between the parent and child node. To remove an arc, call `DSL_network::RemoveArc`. Arcs are automatically removed when either the parent or the child node represented by the arc is deleted.

The graph defined by nodes and arcs in `DSL_network` is an acyclic directed graph (DAG) at all times. If your call to `AddArc` would result in a cycle in the graph, the method fails and returns `DSL_CYCLE_DETECTED`.

To inspect the graph structure, use `DSL_network::GetParents` and `GetChildren` methods, which return a const reference to the `DSL_intArray` object containing the handles of theparents/children:

```
int nodeHandle = …;
const DSL_intArray& parents = net.GetParents(nodeHandle);
for (int i = 0;i < parents.GetSize(); i ++)
{
    printf("Parent %d: %d %s\n", i, parents[i], net.GetNode(parents[i]->GetId());
}
const DSL_intArray& children = net.GetChildren(nodeHandle);
for (int i = 0; i < children.GetSize(); i ++)
{
    printf("Child %d: %d %s\n", i, children[i], net.GetNode(children[i]->GetId());
}
```

To check if an arc exists, use `DSL_intArray::Contains` with either parents or children array. For example, if an arc exists betwen h1 and h2 both `net.GetParents(h2).Contains(h1)` and `net.GetChildren(h1).Contains(h2)` will return `true`.

To check if two nodes are connected by an arc irrespective of its direction, use `DSL_network::Related` function. `DSL_network::GetAncestors` and `GetDescendants` return all ancestors and descendants of the node.

Arcs can be reversed with `DSL_network::ReverseArc`. This operation preserves the joint probability distribution represented by the network.

After adding an arc with `AddArc`, the child node's probability distribution is expanded to accommodate the parent, but the actual probabilities in the distribution are copied. Call `DSL_network::IsArcNecessary` to verify if the child node's conditional dependencies expressed by different probabilities in its probability table.

6.8 Anatomy of a node

The instance of `DSL_node` object managed by `DSL_network` aggregates other objects representing different aspects of the node. The most important are the definition and value objects. Additionally, the node contains the description attributes which do not affect inference, but determine node’s location, color, name, etc.

6.8.1 Multidimensional arrays

To represent conditional probability tables (CPTs), SMILE uses `DSL_Dmatrix` class. CPT describes the interaction between a node and its immediate predecessors. The number of dimensions and the total size of a conditional probability table are determined by the number of parents, the number of states of each of these parents, and the number of states of the child node. Essentially, there is a probability for every state of the child node for every combination of the states of the parents. Nodes that have no predecessors are specified by a prior probability distribution table, which is a vector specifying the prior probability of every state of the node.

Conditional probability tables are stored as vectors of doubles that are a flattened versions of multidimensional tables with as many dimensions as there are parents plus one for the node itself. The order of the coordinates reflects the order in which the arcs to the node were created. The most significant (leftmost) coordinate will represent the state of the first parent. The state of the node itself corresponds to the least significant (rightmost) coordinate.

The image below is an annotated screenshot of GeNIe's node properties window open for the *Forecast* node in the model created in *Tutorial 1* [71]. *Forecast* has three outcomes and two parents: *Success of the venture* and *State of the economy*, with two and three outcomes, respectively. Therefore, the total size of the CPT is 2 x 3 x 3 = 18. The arrows in the image (not part of the actual GeNIe window) show the ordering of the entries in the linear buffer that `DSL_Dmatrix` uses internally. The first (or rather, the zero-th, because all indices in SMILE are zero-based) element, the one with the value of 0.7 and yellow background, represents P(Forecast=Good | Success of the venture=Success & State of the economy=Up). It is followed by the probabilties for *Moderate* and *Poor* outcomes given the same parent configuration. The next parent configuration is *Success of the venture=Success* & *State of the economy=Flat*, etc.

Success of the venture	Success			Failure		
State of the economy	Up	Flat	Down	Up	Flat	Down
Good	0.7	0.65	0.6	0.15	0.1	0.05
Moderate	0.29	0.3	0.3	0.3	0.3	0.25
Poor	0.01	0.05	0.1	0.55	0.6	0.7

In addition to linear indexing, `DSL_Dmatrix` allows access to its elements through coordinate system by overloading `operator[]` and `Subscript` methods. The coordinates are specified by `DSL_intArray` object containing values for each parent and node for which CPT is defined. For example, the element for

- *Success of the venture=Failure*
- *State of the economy=Up*
- *Forecast=Poor*

would have the coordinates [1, 0, 2]. It's linear index is 11. `DSL_Dmatrix` provides `CoordinatesToIndex` and `IndexToCoordinates` methods for conversion between coordinates and linear indices and vice versa. The `NextCoordinates` and `PrevCoordinates` methods can be used to shift the coordinates forward or backward in odometer-like fashion (with the rightmost/least significant entry representing node for which the CPT is defined changing every time).

Note that `DSL_Dmatrix` is not used exclusively for CPTs. Other uses of this class in SMILE include (but are not limited to) expected utility tables and marginal probability distributions, both of which can be indexed.

6.8.2 Node definition

The definition of a node specifies how this node interacts with other nodes in the network. Node definition are saved in the network file (by `DSL_network::WriteFile` and `WriteString`) and are retrieved back when reading the file. For a general chance node, the definition consists of a conditional probability table (CPT) and a list of state names. To access a node definition, please use `DSL_node::Def` method, which returns a pointer to the instance of the class derived from the `DSL_nodeDef`.

```
int nodeHandle = …;
DSL_node *node = net.GetNode(nodeHandle);
DSL_nodeDef *def = node->Def();
```

The definition object is managed by the network containing the node. SMILE provides a number of classes derived from `DSL_nodeDef`, specialized to represent different node types (CPT, Noisy-MAX, decision, etc.). The choice of the definition object class associated with a given node is based on the node type parameter that is passed to `DSL_network::AddNode`. As SMILE is compiled with RTTI disabled, you cannot use `dynamic_cast` to check for the actual type of the object returned by `DSL_node::Def`. However, you can use `DSL_nodeDef::GetType` and `GetTypeName` methods:

```
int nodeHandle = net.AddNode(DSL_CPT, "myNodeId");
DSL_nodeDef *def = net.GetNode(nodeHandle)->Def();
printf("Type of the definition: %d %s\n", def->GetType(), def->GetTypeName());
```

In the example above, the `def` variable points to the object of the `DSL_cpt` class derived from `DSL_nodeDefinition`. While it is possible to `static_cast` or use `DSL_node::Def<T>` to obtain access to type-specific functionality of the object, SMILE provides general purpose virtual methods defined in `DSL_nodeDef`, overridden in derived classes, which makes casting unnecessary most of the time. For example, two of these methods are `DSL_nodeDef::GetMatrix` and `GetOutcomeIds`, which give access to node's parameters and state names. Some of the node types do not support all of the operations: the decision nodes do not have any numeric parameters, and, therefore, their definition object of `DSL_decision` class returns `NULL` from its `GetMatrix` method.

While the virtual methods defined in `DSL_nodeDef` support most of the APIs required for discrete Bayesian networks and influence diagrams, there is plenty of extended functionality in SMILE's node definition classes (continuous equation nodes, canonical nodes, interval-based discrete nodes, etc). SMILE provides `DSL_nodeDef::Def<T>` template method, which returns a pointer to the node definition type specified as its template parameter. The example below assumes that `eqNodeHandle` is the handle to the `DSL_EQUATION` node, which uses `DSL_equation` class derived from `DSL_nodeDef` to represent its definition:

```
DSL_node *node = net.GetNode(eqHandle);
auto eq = node->Def<DSL_equation>(); // eq variable type is DSL_equation*
eq->SetEquation("a=Normal(0,1)");
```

The templated `Def<T>` method does a simple inline `static_cast` and is provided as a 'syntactic sugar'. Note that there is no runtime type checking here. If there is only one call through the pointer cast to the derived type, the code can be shortened to a single line:

```
net.GetNode(eqHandle)->Def<DSL_equation>()->SetEquation("a=Normal(0,1)");
```

The choice of style is, of course, left to the SMILE user.

6.8.3 Node value & evidence

The value of a node contains values (typically, marginal probability distribution or expected utilities) calculated for the node by the inference algorithm. Unlike the definition, the value is not written as part of the network by `DSL_network::WriteFile` and `WriteString`. Like the definition, the value object is managed by the network. The actual value object, which you can access through `DSL_node::Val` method, is an instance of the class derived from `DSL_nodeVal`. SMILE chooses the class appropriate for the node type during node creation.

In addition to the numeric output of the inference algorithm, the node value object may contain the evidence for the node, which is (along with the node definition) part of the input to the inference algorithm. To set and remove the evidence, use `DSL_nodeVal::SetEvidence` and `ClearEvidence` methods. As with the definition part of node, most of the time there is no need to cast the pointer returned by `DSL_node::Val` to a specific class, because the base class provides the set of general purpose virtual functions overridden by derived value classes.

```
int evidenceNodeHandle = …
DSL_nodeVal *evVal = net.GetNode(evidenceNodeHandle)->Val();
evVal.SetEvidence(1); 1 is the 0-based outcome index
net.UpdateBeliefs();
int beliefNodeHandle = …;
DSL_nodeVal *beVal = net.GetNode(beliefNodeHandle)->Val();
if (beVal->IsValueValid())
```

```
{
    const DSL_Dmatrix *m = beVal->GetMatrix();
    // use the matrix
}
```

Note that before accessing the actual numeric value of the node with GetMatrix, we need to check if the value is valid by calling IsValueValid. The value will not be valid, for example, if inference algorithm was not called yet, or some definition or evidence has changed after the last inference call.

It is also possible to specify virtual evidence using DSL_nodeVal::SetVirtualEvidence method. Virtual evidence allows for entering uncertain observation (in form of probability distribution over the possible states of the observation) directly into a normally unobservable variable. SetVirtualEvidence requires an array, a vector, or an initializer list with size equal to the number of node outcomes. The following snippet shows how to set virtual evidence for a node with three outcomes:

```
int evidenceNodeHandle = …
DSL_nodeVal *evVal = net.GetNode(evidenceNodeHandle)->Val();
evVal->SetVirtualEvidence({ 0.2, 0.7, 0.1});
```

Similarly to the node definition class hierarchy, some functionality (like equation node values) is available only when the DSL_nodeVal pointer is cast to an appropriate type. The DSL_node::Val<T> template member function performs the inline static_cast. The example below assumes that eqNodeHandle is the handle to the DSL_EQUATION node, which uses DSL_equationEvaluation class derived from DSL_nodeVal to represent its value:

```
DSL_node *node = net.GetNode(eqNodeHandle);
auto veq = node->Val<DSL_equationEvaluation>();
veq->GetHistogram(-1, 1, 100, histogram);
```

6.8.4 Other node attributes

Node attributes that not related to inference are grouped in the DSL_nodeInfo object, accessible through DSL_node::Info method. In turn, the DSL_nodeInfo provides access to the following objects:

- header: textual attributes, like node identifier, name, and description, through DSL_nodeInfo::Header method returning a reference to DSL_header object. For convenience, these attributes are also available directly through DSL_node member functions, like DSL_node::GetName.

- screen information: position, colors, border thickness, etc., through DSL_nodeInfo::Screen method returning a reference to DSL_screenInfo object.

- user properties: a list of key/value pairs used for application-specific purposes, through DSL_nodeInfo::UserProperties method returning a reference to DSL_userProperties object.

The example below displays a node's screen position and size:

```
int nodeHandle = …;
DSL_node *node = net.GetNode(nodeHandle);
DSL_nodeInfo &info = node->Info();
const DSL_rectangle &pos = info.Screen().position;
printf("Node center: (%d, %d), size: (%d,%d)\n",
    pos.center_X, pos.center_Y,
    pos.width, pos.height);
```

6.9 Discrete Bayesian networks

Discrete Bayesian networks contain variables (nodes). A node describes a finite set of conditions and takes values from a finite, usually small, set of states (outcomes). The `DSL_nodeDef` 141 and `DSL_nodeVal` 161 APIs were designed to cover all functionality required to work with discrete variables with outcomes described by string labels (identifiers). In SMILE 2, we introduced an option to use *numeric domains* 39 with discrete nodes.

Regardless of the domain of a node, its outcomes can be managed with `DSL_nodeDef::SetNumberOfOutcomes`, `AddOutcome`, `InsertOutcome` and `DeleteOutcome`. Access to outcomes is available through `DSL_nodeDef::GetNumberOfOutcomes` and `GetOutcomeIds`.

Evidence for discrete nodes can be set and read with `DSL_nodeVal::SetEvidence(int)` and `DSL_nodeVal::GetEvidence`. After successful inference `DSL_nodeVal::GetMatrix` will return a pointer to an one-dimensional matrix containing the calculated posterior marginal probabilities for the node.

6.9.1 CPT nodes

CPT nodes are general chance variables, represented by conditional probability tables. To create a CPT node in SMILE, pass `DSL_CPT` as node type to `DSL_network::AddNode`.

The new node will have an instance of `DSL_cpt` class as its definition. In practice, there is no need to cast the `DSL_nodeDef` pointer returned by `DSL_node::Def` to `DSL_cpt`, because all required functionality (outcome management and access to the conditional probability table) is accessible through the virtual methods introduced in the base class `DSL_nodeDef`. The node value object will be an instance of `DSL_beliefVector` class. As with the definition, there is no need for explicit casting the `DSL_nodeVal` returned from `DSL_node::Val`. Access to evidence and calculated posterior probabilities can be performed through virtual methods defined in the base class `DSL_nodeVal`.

The *Hello, SMILE!* 22 program and tutorials *1* 71, *2* 75, and *3* 79 are using CPT nodes through base classes `DSL_nodeDef` and `DSL_nodeVal`.

Please note that as you increase the number of parents of a node, the node's CPT grows exponentially. With 10 binary parents, the CPT contains 1,024 columns, with 20 binary parents, this number is 1,048,576. As the number of parents grows, at some point, this will exhaust all available computer memory. The canonical nodes described in the next section offer a solution to the exponential CPT growth.

6.9.2 Canonical nodes

Canonical probabilistic nodes, such as Noisy-MAX/OR, Noisy-MIN/AND, and Noisy-Adder gates, implemented by SMILE, are convenient knowledge engineering tools widely used in practical applications. In case of a general CPT binary node with n binary parents, the user has to specify 2^n parameters, a number that is exponential in the number of parents. This number can quickly become prohibitive: when the number of parents n is equal to 10, we need 1,024 parameters, when it is equal to 20, the number of parameters is equal to 1,048,576, with each additional parent doubling it. A Noisy-OR model allow for specifying this interaction with only $n+1$ parameters, one for each parent plus one more number. This comes down to 11 and 21 for n equal to 10 and 20 respectively.

Canonical models are not only great tools for knowledge engineering - they also lead to significant reduction in computation through the independences that they model implicitly. Using canonical gates makes thus model construction easier but also leads to models that are easier to solve.

To create canonical nodes, pass DSL_NOISY_MAX or DSL_NOISY_ADDER to DSL_network::AddNode. Each node type exposes its own specific attributes through its definition object, but otherwise works in exactly the same way as a CPT node (has at least two outcomes, can be used as a parent or a child wherever the CPT node can, etc).

6.9.2.1 Noisy-MAX

Noisy-MAX is a generalization of the popular canonical gate Noisy-OR that is capable of modeling interactions among variables with multiple states. If all the nodes in question are binary, a Noisy-MAX node reduces to a Noisy-OR node. The Noisy-MAX, as implemented in SMILE, includes an equivalent of negation (through re-ordering of states). By DeMorgan's laws, the OR function (or its generalization, the MAX function) along with a negation, is capable of expressing any logical relationship, including the AND (and its generalization, MIN). This means that SMILE's Noisy-MAX can be used to model the Noisy-AND/MIN functions, as well as other logical relationships.

SMILE's clustering algorithm contains special code path for networks with Noisy-MAX nodes, which can speed up computations significantly. See the *Inference* 52 section of this manual for details.

To create a Noisy-MAX node, use the DSL_NOISY_MAX type with DSL_network::AddNode:

```
int h = net.AddNode(DSL_NOISY_MAX, "node1");
```

AddNode will in this case create a node with a definition object of DSL_noisyMAX class. In addition to information about node outcomes, the definition contains the following Noisy-MAX specific data:

- a set of conditional probabilities, also called outcome strengths
- for each of parent node, the vector of parent outcome strengths.

The conditional probabilities are stored in a two-dimensional DSL_Dmatrix object, the Noisy-MAX table, accessible through DSL_noisyMAX::GetCiWeights. The size of the first dimension of the matrix is the sum of the number of each parent outcomes plus one (for the leak column). The second dimension is the number of the child node's own outcomes. The last column for each of the parents is constrained to contain zero probabilities in all but its last element.

Each set of parent outcomes can be ordered specifically for the current interaction with the child node. The Noisy-MAX table always follows the specified order of parent outcomes.

As an example, consider a binary Noisy-MAX node with two parents, each with three outcomes. The following snippet modifies the probabilities and outcome strengths for the second parent (with zero-based index of 1).

```
auto maxDef = net.GetNode(h)->Def<DSL_noisyMAX>());
DSL_Dmatrix weights = maxDef->GetCiWeights();
int BASE = 2 * 3;
weights[BASE]     = 0.1;
weights[BASE + 1] = 0.9;
weights[BASE + 2] = 0.3;
weights[BASE + 3] = 0.7;
maxDef->SetCiWeights(weights);
DSL_intArray strengths({2, 0, 1});
maxDef->SetParentOutcomeStrengths(1, strengths);
```

The value of BASE is calculated as a product of node outcome count and preceding parents' outcome counts (in this case, there is just one preceding parent with three outcomes). The probabilities for the parent are written into a `DSL_Dmatrix` object initialized with `GetCiWeights`, and written back with `SetCiWeights`. The next step changes the order of parents' outcomes in relationship to the Noisy-MAX (child) node. The `DSL_intArray` object is created and initialized with parent outcome indices. After the `DSL_noisyMAX::SetParentOutcomeStrengths` call, the first column of probabilities for the parent with index 1 (the one with 0.1 and 0.9) represents the probabilities for the parent outcome with index 2 (because 2 is the first element in the `strengths` array). Note that this does not modify the parent node in any way and the ordering is valid only in the context of this particular parent-child relationship. Assuming that we started with the default uniform probabilities in the table, our modifications yields the following Noisy-MAX definition, as viewed in GeNIe:

Parent	p1			p2			LEAK
State	State0	State1	State2	State2	State0	State1	
State0	0.5	0.5	0	0.1	0.3	0	0.5
State1	0.5	0.5	1	0.9	0.7	1	0.5

The outcomes of both parents are *{State0, State1 and State2}*. However, by using `DSL_noisyMAX::SetParentOutcomeStrenghts` for parent *p2*, its outcomes are seen by the Noisy-MAX child node as *{State2, State0, State1}*. Other Noisy-MAX nodes in the same network can set up their own parent outcome ordering if *p2* becomes their parent.

See the `DSL_noisyMAX` [154] reference for details.

6.9.2.2 Noisy-Adder

The Noisy-Adder model is described in the doctoral dissertation of Adam Zagorecki (2010), Section 5.3.1 *Non-decomposable Noisy-average*. Essentially, it is a non-decomposable model that derives the probability of the effect by taking the average of probabilities of the effect given each of the causes in separation.

To create a Noisy-Adder node, use the `DSL_NOISY_ADDER` type with `DSL_network::AddNode`:

```
int h = net.AddNode(DSL_NOISY_MAX, "node1");
```

The class of the definition object associated with the new node is `DSL_noisyAdder`. In addition to information about node outcomes, the definition contains the following Noisy-Adder specific data:

- index of the node's own distinguished state
- for each of the node's parents, the index of the parent's distinguished state
- weights for each of the node's parents and for the leak

Consider a binary Noisy-Adder node with two parents, each having three outcomes. The following code snippet modifies the probabilities, the weight, and the distinguished state for the second parent (with zero-based index 1). Node's own distinguished state is set to 0.

```
auto adderDef = net.GetNode(h)->Def<DSL_noisyAdder>());
adderDef->SetDistinguishedState(0);
DSL_Dmatrix weights = addDef->GetCiWeights();
int BASE = 2 * 3;
p[BASE + 2] = 0.2;
p[BASE + 3] = 0.8;
p[BASE + 4] = 0.4;
```

```
p[BASE + 5] = 0.6;
adderDef->SetCiWeights(weights);
adderDef->SetParentDistinguishedState(1, 0);
adderDef->SetParentWeight(1, 2.5);
```

The image below shows the Noisy-Adder definition table (as viewed in GeNIe) after the modifications applied by the code above (assuming that the table contained initially the default probabilities of 0.5). The modified parent's weight is shown in parenthesis after its identifier (*p2*). Distinguished states are marked by bold font.

Parent (Weight)	p1 (1)			p2 (2.5)			LEAK (1)
State	State0	State1	**State2**	**State0**	State1	State2	
State0	0.5	0.5	1	1	0.2	0.4	0
State1	0.5	0.5	0	0	0.8	0.6	1

Parent distinguished states and weights are part of the child Noisy-Adder definition. If *p2* has other Noisy-Adder children, each of these nodes can specify its own distinguished state and weight for *p2*.

See the `DSL_noisyAdder` [155] reference for details.

6.9.3 Discrete deterministic nodes

Discrete deterministic nodes can be created by using the `DSL_TRUTHTABLE` node type identifier when calling `DSL_network::AddNode`. They behave like CPT nodes, with the exception that their probability tables contain only zeros and ones. Deterministic nodes are defined as having no noise in their definition, so there is no uncertainty about the outcome of a deterministic node once all its parents are known. Please note that deterministic nodes are not free of the problem of exponential probability table growth described in the *CPT nodes* [35] section.

While it is possible to model a deterministic variable using a chance node with its CPT filled with zeros and ones, the deterministic nodes are displayed as double ellipses in GeNIe, making the modeler's intent explicit and, hence, reduces the probability of modeling errors.

The definition of a deterministic node is represented by the object of the `DSL_truthTable` class. The code snippet below assumes a deterministic node with three outcomes and two binary parents. Its definition is set with a call to `DSL_truthTable::SetResultingStates`. For each column of the truth table, we specify one resulting state:

```
DSL_node *node = net.GetNode(detHandle);
auto tt = node->Def<DSL_truthTable>();
DSL_intArray resStates = { 2, 0, 1, 1 };
tt->SetResultingStates(resStates);
```

The following one-liner peforms the same task: .

```
net.GetNode(detHandle)->Def<DSL_truthTable>()->SetResultingStates({2,0,1,1});
```

The truth table created by the above code would look as follows in GeNIe:

p1	State0		State1	
p2	State0	State1	State0	State1
State0	○	●	○	○
State1	○	○	●	●
State2	●	○	○	○

There is also SetResultingStates method which takes an array of strings as input. An equivalent to the example above would initialize the DSL_stringArray with {"State2", "State0", "State1", "State1"}. See the DSL_truthTable[150] reference for details.

6.9.4 Discrete nodes and numeric domains

By default, discrete nodes represent discrete random variables that have categorical outcomes described by outcome identifiers. In SMILE 2, it is possible to model discrete random variables that are numerical in nature. The outcomes of the node can be associated with adjacent numeric intervals with explicitly specified borders, or with a set of discrete numeric values (one for each outcome). This new functionality is defined in the DSL_discDef class, which is a base class for all discrete node types.

There are four possible outcome types for discrete nodes:

- identifiers (the default)
- identifiers and numeric intervals
- numeric intervals with no identifiers
- identifiers and numeric point values

Nodes with intervals and point values are described in the next two subsections. See DSL_discDef[145] reference for API details.

6.9.4.1 Outcome intervals

Discrete nodes by default do not have any numeric information associated with their outcomes. To add outcome intervals, call DSL_discDef::SetIntervals.

```
int handle = …
auto discDef = net.GetNode(handle)->Def<DSL_discDef>();
discDef->SetIntervals({ 0, 2.78, 3.14, 77, DSL_inf() }, true);
```

The example above passes five numeric interval boundaries in std::initializer_list (overloads with DSL_doubleArray and std::vector are also available) as its first parameter, and defines four intervals: 0 to 2.78, 2.78 to 3.14, 3.14 to 77, and 77 to infinity. We used the DSL_inf() function to return a floating point value representing the positive infinity. DSL_inf() is just a convenient way to call std::numeric_limits<double>::infinity(). Using -DSL_inf() as first element in the intervals definition creates open first interval. The list of interval boundaries has to be sorted from lowest to highest. It is possible to specify a point interval by means of two identical numbers as its boundaries.

The number of outcomes will change to reflect the number of intervals passed to SetIntervals. It is not necessary to call SetNumberOfOutcomes before SetIntervals.

The second parameter in the SetIntervals call is set to true to indicate that existing outcome identifiers should be discarded. The use of outcome identifiers with outcome intervals is optional. If the identifiers were discarded, the outcome identifier array will have empty string for each outcome id.

It is possible to call the outcome modifying methods when node has intervals. After `AddOutcome` or `InsertOutcome` new interval will be created by splitting the existing interval at the appropriate index. `DeleteOutcome` will remove the interval at the specified index. At any time it is possible to call `SetIntervals` again with a new list of interval boundaries. In such case, the list of new interval boundaries may have different size than the existing list.

Other methods in the `DSL_discDef` for working with intervals are `HasIntervals`, `GetIntervals`, and `RemoveIntervals`. The last function removes the interval boundaries from the node definition, and in case there were no outcome identifiers, creates the default set of identifiers. The number of outcomes does not change.

With intervals defined, it is possible to calculate the mean and standard deviation for the node given the marginal probability distribution over its outcomes. In calculating the moments of the distribution, SMILE treats it as a continuous distribution, assuming that it is uniform within each of the intervals. Open intervals are an exception; the distribution over these is assumed to be half-normal scaled to ensure the normal PDF height is equal to the neighboring closed interval's uniform distribution. To retrieve the mean or the standard deviation, call `GetMean` or `GetStdDev` respectively on the node value object.

Intervals allow for continuous evidence in discrete nodes. To set continuous evidence, call `DSL_nodeValue::SetEvidence(double)`. The evidence will be stored and will be internally converted into an outcome index during inference in a discrete Bayesian network or an influence diagram. However, hybrid models 50 can take advantage of continuous evidence in a discrete node.

6.9.4.2 Outcome point values

To define a discrete distribution over numeric node outcomes, use `DSL_discDef::SetPointValues`. The example below assumes that node has three outcomes:

```
int handle = …
auto discDef = net.GetNode(handle)->Def<DSL_discDef>();
discDef->SetPointValues({ 3.14, -2.78, 77 });
```

The three numeric values in `std::initializer_list` (overloads with `DSL_doubleArray` and `std::vector` are also available) will be associated with existing node outcomes. In contrast with `SetIntervals`, `SetPointValues` requires its first parameter to have the number of elements equal to the number of node outcomes (it will not add or remove outcomes). There is also no requirement for ordering point values. The identifiers of the outcomes will not be modified, as nodes with point values always require outcome identifiers.

Outcome modifying methods like `AddOutcome, InsertOutcome` or `DeleteOutcome` work normally when node has point values. New outcomes will have their point values initialized to zero. The point values can only be changed all at once with another `SetPointValues` call.

Other methods in the `DSL_discDef` for working with point values are `HasPointValues`, `GetPointValues` and `RemovePointValues`.

With point values defined, it is straightforward to calculate the mean and standard deviation for the marginal probability distribution over the outcomes of the node. To retrieve the mean or the standard deviation, call `GetMean` or `GetStdDev` respectively on the node value object.

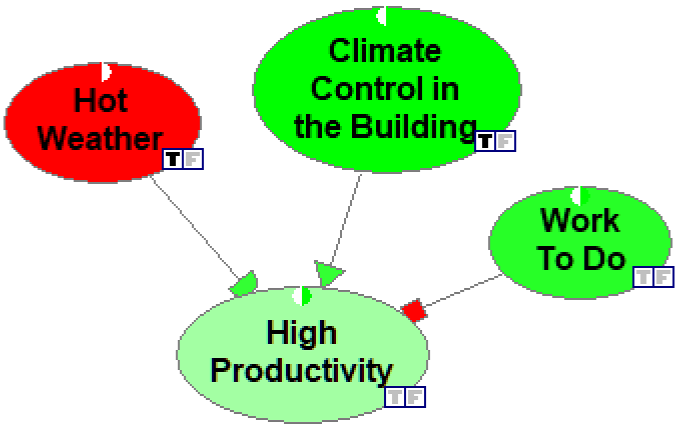
Point values do not allow for continuous evidence in the discrete nodes. Calling `SetEvidence(double)` will fail, even if the parameter is equal to one of the defined point values.

6.10 Qualitative models

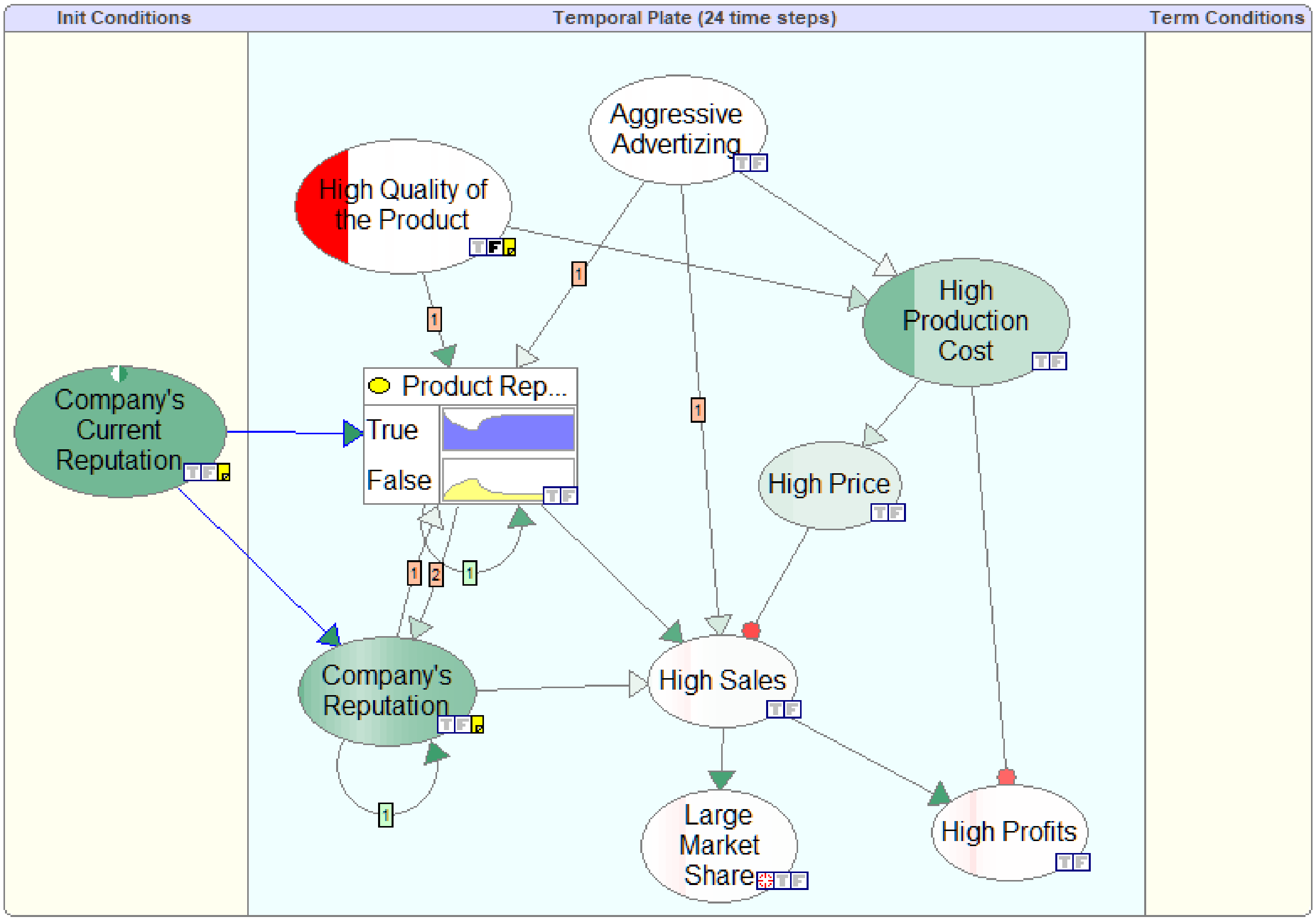
Qualitative models in SMILE consist of the DeMorgan nodes. These nodes have `DSL_DEMORGAN` type and their definition is implemented by `DSL_demorgan` class. These nodes have always two states (*True* and *False*) and can only be linked with other nodes of `DSL_DEMORGAN` type. While the underlying methodology (and, therefore, `DSL_demorgan` API) is quantitative and grounded in probability theory, the models can be created interactively in QGeNIe (a simplified version of GeNIe designed to work exclusively with qualitative models) by means of qualitative sliders, colors, and gradients rather than numbers. The color gradients are calculated in QGeNIe based on user preferences, but are based on the numerical output from SMILE, which for DeMorgan nodes are stored in the `DSL_beliefVector` node value classes (just like `DSL_CPT` or `DSL_NOISY_MAX` nodes).

QGeNIe's file format extension is `.qdsl`. The `.qdsl` files are supported by `DSL_network::ReadFile` and `WriteFile`. Note that `DSL_network::WriteFile` does not check for the node types in the network and your code should explicitly pass the file name with the `.qdsl` extension when the network to be saved is qualitative.

Here is a simple qualitative model displayed in QGeNIe colored with the default green/red palette. The different types of arc heads indicate connection types, described later in this section. `T` and `F` icons are used to indicate the node evidence.



Qualitative models can also be dynamic. The horizontal shading on the plate nodes represents their probabilities changing over time.



Those readers who were exposed to some logic in high school or college know that any logical function can be expressed in one of the Augustus De Morgan's canonical forms, an alternative (OR) of conjunctions (AND) or a conjunction of alternatives. It is also a fact stemming from so called De Morgan laws that the OR function and a negation can express the AND function. A combination of OR functions and negations can express any logical function. QGeNIe's DeMorgan gate offers essentially an intuitive way of expressing any logical function and, in particular, a combination of ORs, ANDs, and negation.

The DeMorgan gate allows for modeling four basic types of influences that one variable (parent in the directed graph) can have on another (a child in the directed graph): (1) a positive influence (a *Cause*) and (2) a negative influence (a *Barrier*), both combining with other causes using a noisy OR interaction, (3) a required condition (a *Requirement*) and (4) a condition that prevents the effect from happening (an *Inhibitor*), both combining with other causes through an AND interaction.

We will now explain the meaning of the four types of causal influences and how they form any logical function when combined.

- Cause: A cause is a parent that has a positive influence on the child. Please note that this influence does not need to be perfect. For example, smoking is generally believed to be a causal factor in lung cancer. Yet, incidence of lung cancer among smokers, while much larger than incidence of lung cancer among non-smokers, is still within a few percent. Hence, the conditional probability of lung cancer given that a person is a smoker is still fairly low. The cause increases the probability of the effect but does not need to be perfect in its ability to cause it.

- Barrier: A barrier is a parent that decreases the probability of a child. For example, regular exercise decreases the probability of heart disease. While it is a well established factor with a negative influence on heart disease, it is unable by itself to prevent heart disease. One way of looking at a barrier is that it is dual to a cause: Absence of the barrier event is a causal factor for the child. One might go around the very existence of barriers by using negated versions of the variables that represent them. In the example above, one might define a variable *Lack of regular exercise*, which would behave as a cause of the variable *Heart disease*. This, however, might become cumbersome if *Regular exercise* participated in other interactions in a model. It might happen that it is a parent of both *Heart disease* and *Good physical shape*. Because *Regular exercise* decreases the probability of one and increases the probability of the other, Barrier, which is a negated Cause, is a useful modeling construct.

- Requirement: A requirement is a parent that is required for the child to be present. There are perfect requirements, such as being a biological female is a requirement for being pregnant but there are also requirements that are in practice not completely necessary. For example, a sexual intercourse is generally believed to be a requirement for pregnancy, but it is not a strict requirement, as pregnancy may be also caused by artificial insemination.

- Inhibitor: An inhibitor is a parent that prevents the child from happening. For example, rain may inhibit wild land fire. Like in the other types of interactions, the parent may be imperfect in inhibiting the occurrence of the child. Fire may start even if there is rain. Similarly to the relationship between causes and barriers, inhibitors are dual to requirements: Absence of an inhibitor event is a requirement for the child. One might go around the very existence of inhibitors by using negated versions of the variables that represent them. In the example above, one might define a variable *No rain*, which would behave as a requirement for the variable *Wild land fire*. This, however, might become cumbersome if *Rain* participated in other interactions in a model. It might happen that it is a parent of both *Wild land fire* and *Good crop*. Because *Rain* is an inhibitor for the former and a requirement for the latter, Inhibitor, which is a negated Requirement, is a useful modeling construct.

The four types of causes interact with their effect through the following logical formula:

$$e = (c1 \lor c2 \lor \ldots \lor \neg b1 \lor \neg b2 \lor \ldots) \land r1 \land r2 \land \ldots \land \neg i1 \land \neg i2 \land \ldots ,$$

where c_is are Causes, b_is are Barriers, r_is are Requirements and i_is are Inhibitors.

For the DeMorgan node API details, see the `DSL_demorgan` 151 reference. QGeNIe manual is available at BayesFusion's documentation website https://support.bayesfusion.com/docs and contains a detailed description of qualitative models, including references to research papers.

6.11 Influence diagrams

Influence diagrams (IDs) introduced by Howard and Matheson (1984), are acyclic directed graphs modeling decision problems under uncertainty. An ID encodes three basic elements of a decision: (1) available decision options, (2) factors that are relevant to the decision, including how they interact among each other and how the decisions will impact them, and finally, (3) the decision maker’s preferences over the possible outcomes of the decision making process.

Influence diagrams use three additional node types next to chance (CPT and canonical) and deterministic nodes:

- Decision nodes represent variables that are under control of the decision maker and model available decision alternatives, modeled explicitly as possible states of the decision node. They have no numerical parameters, only a discrete set of outcomes. Decision nodes can be children of decision and chance nodes. The node type identifier passed to `DSL_network::AddNode` is `DSL_LIST` (short for "List of decisions").

- Value nodes, i.e., a measure of desirability of the outcomes of the decision process. They are quantified by the utility of each of the possible combinations of outcomes of the parent nodes, specified as an `DSL_Dmatrix` object. Value nodes can be children of decision and chance nodes. Pass `DSL_TABLE` to `DSL_network::AddNode` to create a value node.

- Multi-attribute utility (MAU) nodes, which combine value nodes to form a multi-attribute utility function. The function can be specified as a set of weights of a linear function (in such case, the node becomes an additive linear utility, ALU) or any expression that refers to identifiers of the value node parents. See the *Equations* 197 section of the reference manual for a list of available functions. MAU nodes can be children of decision, value, and other MAU nodes. If decision parents exist, the definition of the MAU node contains a separate set of weights or expressions for each combination of decision parents. Use `DSL_MAU` with `DSL_network::AddNode` to create a MAU node. Note that APIs for expression-based MAU nodes are available in the `DSL_mau` class derived from `DSL_nodeDef` (so casting is required).

As is the case with Bayesian networks, the values calculated by influence diagram inference algorithms are stored in node values and can be accessed through the same APIs. However, the interpretation of the numbers stored in `DSL_Dmatrix` objects retrieved with `DSL_nodeVal::GetMatrix` is extended. The matrices are indexed by the set of nodes called indexing parents. Indexing parents are unobserved decision nodes that precede the current node or unobserved chance nodes that are predecessors of decision nodes and should have been observed before the decisions can be made. Call `DSL_nodeVal::GetIndexingParents` to retrieve indexing parents. The set of outcomes of indexing parents is called a policy. After a successful inference in an influence diagram, node values are:

- for chance and deterministic nodes: posterior probabilities for each policy

- for decision nodes: expected utilities for all outcomes and for each policy

- for value and MAU nodes: expected utility for each policy

See *Tutorial 4* 83 for a simple influence diagram demo program.

6.12 Dynamic Bayesian networks

A Bayesian network is a snapshot of the world at a given time and is used to model systems that are in some kind of equilibrium state. Unfortunately, most systems in the world change over time and sometimes we would like to know how these systems evolve over time.

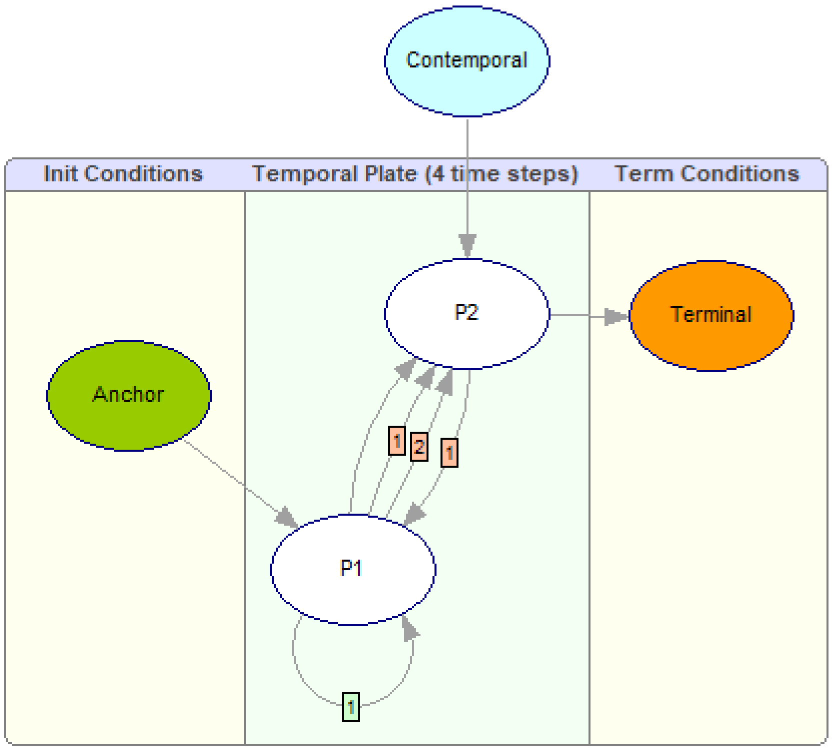
A dynamic Bayesian network (DBN) is a Bayesian network extended with additional mechanisms that are capable of modeling influences over time. The temporal extension of BNs does not mean that the network structure or parameters changes dynamically, but that a dynamic system is modeled. In other words, the underlying process, modeled by a DBN, is stationary. A DBN is a model of a stochastic process.

The implementation of DBNs in SMILE allows for use both chance (CPT and canonical) and deterministic node types in dynamic models.

Tutorial 6 89 contains a complete program demonstrating the use of DBNs.

6.12.1 Unrolling

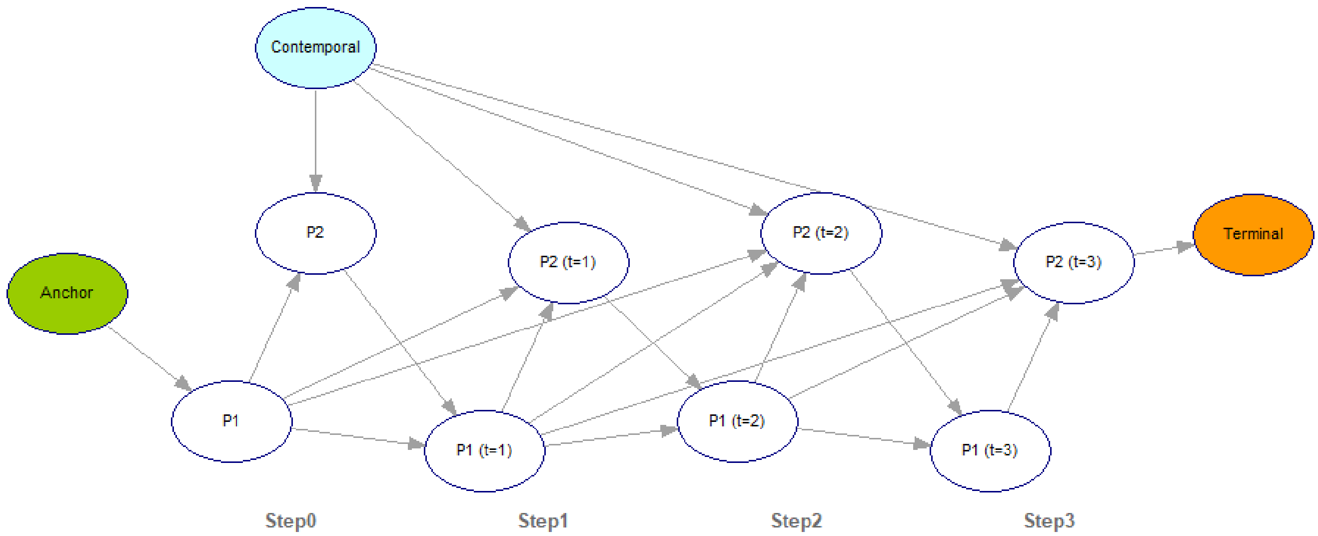
SMILE's inference algorithm for dynamic Bayesian networks (DBNs) converts them to temporary static networks by unrolling them over the time steps and then solves these networks. Results of this inference are copied back into the node values in the original DBNs. This gives SMILE the flexibility of allowing to specify time influences of any order and makes its implementation of DBN uniquely powerful. The structure of the following DBN (in GeNIe format) does not represent any real system, it is created just for demonstration purposes.



Different colors of the nodes represent their location relative to the abstract "temporal plate." There are four temporal types of nodes, defined by the following `enum`:

```
enum dsl_temporalType { dsl_normalNode, dsl_anchorNode, dsl_terminalNode, dsl_plateNode };
```

By default, the nodes in the network are set to be `dsl_normalNode`. By using `DSL_network::SetTemporalType` you can change their temporal type assignment. In the example model above, the plate nodes are white, the anchor node is green, the terminal node is orange and the blue node is normal. For simplicity, we use only single anchor, terminal, and normal nodes. Note the presence of multiple arcs between two plate nodes and the arc linking *P1* with itself. Some of the arcs between plate nodes have a tag with a number; these are *temporal arcs*, which are created by `DSL_network::AddTemporalArc`. The number on the tag is a *temporal order* of an arc. The arcs without the tag were added by `DSL_network::AddArc`. The result of unrolling the above DBN (this is performed automatically by the DBN inference algorithm; it is possible to create such network with `DSL_network::UnrollNetwork`) over the four time steps specified in the model is the following:



To reduce the size of the unrolled network, we changed the number of time steps (slices) from the default value of 10 to 4 with a call to `DSL_network::SetNumberOfSlices`. The colors of the original nodes were carried over to the unrolled model, which is extended by creating new copies of the plate nodes. The structure of this network shows how the temporal arcs are used to express the conditional dependency of plate nodes in time step *i* on the plate nodes in time step *j*, where *i* > *j*.

Consider the temporal arc between *P1* and *P2* with temporal order 2. It was copied twice to link *P1* with *P2 (t=2)*, and *P1 (t=1)* with *P2 (t=3)*. Note that there is no copy of this arc starting from *P1 (t=2)*, because its child would be in *t=2+2=4*, and (zero-based) time stops at 3 in our example. On the other hand, all three temporal arcs with temporal order 1 are copied three times. The relationship represented by an arc linking *P1* with itself in the DBN is clearly visible between *P1* and *P1 (t=1)*, *P1 (t=1)* and *P1 (t=2)* and *P1 (t=2)* and *P1 (t=3)*.

Note the difference between the arcs originating in *Contemporal* and *Anchor* nodes. The *Anchor* node has children only in the initial slice, while the *Contemporal* node is linked to all time slices. The *Terminal* node has parents only in the last slice.

To ensure that the unrolled network remains a directed acyclic graph, the following arcs are forbidden in DBNs:

- from plate nodes to normal and anchor nodes
- from terminal nodes to non-terminal nodes

Unrolling is performed automatically during inference. For debug/explanatory purposes, it is also possible to obtain an unrolled network by calling `DSL_network::UnrollNetwork`. The unrolled network created this way is an independent model, which means that changes made to the DBN after `UnrollNetwork` call are not propagated into the unrolled network. Any possible modifications of the unrolled network are not copied to the original DBN either.

6.12.2 Temporal definitions

Consider node *P2* from the example in the previous section. It has four incoming arcs:

- normal arc from *Contemporal*

- normal arc from *P1*
- two temporal arcs from *P1* with temporal orders 1 and 2

However, in the unrolled network's slice for *t=0*, *P2* has only two incoming arcs. This is because there are no slices representing time points before *t=0*. *P2* in the slice for *t=1* has three incoming arcs, because it is now possible to link *P1* at *t=0* with *P2* at *t=1*. Finally, starting with slice for *t=2*, *P2* has four incoming arcs. The structure of the unrolled network requires *P2* to have separate CPTs for *t=0*, *t=1* and *t>=2*. Generally speaking, if a plate node has an incoming arc of order *x*, it will require *x+1* separate definitions. To get and set the temporal definitions for CPT nodes, use `DSL_nodeDef::GetTemporalDefinition` and `SetTemporalDefinition`. The node definition classes for canonical and deterministic nodes have other methods for accessing temporal definitions. For example, the `DSL_noisyMAX` defines `Get/SetTemporalCiWeights` and `Get/SetTemporalParentOutcomeStrengths`. See the reference section for details; the method names always start with `GetTemporal` or `SetTemporal`.

All parents for the temporal definition with a specified temporal order can be retrieved by calling `DSL_network::GetUnrolledParents`. Note that `DSL_network::GetTemporalParents` called for the same temporal order returns only a subset of parents indexing this temporal definition. This is caused by unrolling: in the unrolled network node *P2* has four incoming arcs in slices for *t>=2*, but only two of these are actually arcs with temporal order 2.

6.12.3 Temporal evidence

To specify evidence for the plate nodes in the DBN, use `DSL_nodeVal::SetTemporalEvidence.` The code snippet below sets the temporal evidence in slices 5 and 7. The latter is virtual evidence.

```
int evidenceNodeHandle = …
DSL_nodeVal *val = net.GetNode(evidenceNodeHandle)->Val();
val.SetTemporalEvidence(5, 1); 1 is the 0-based outcome index
std::vector<double> virtEv(nodeOutcomeCount);
// fill in virtEv here
val.SetTemporalEvidence(7, virtEv);
```

To retrieve the temporal evidence, use `DSL_nodeVal::GetTemporalEvidence`. The method has two overloads, one for normal and another for virtual evidence.

Other useful methods are `DSL_nodeVal::HasTemporalEvidence` and `IsTemporalEvidence`, which check whether a node has any temporal evidence or temporal evidence in specified temporal order, respectively.

6.12.4 Temporal beliefs

For plate nodes in the DBN, the inference algorithm calculates temporal beliefs, which are marginal posterior probability distributions indexed by time. To access temporal beliefs, use the `DSL_nodeVal::GetMatrix` method. We described the same method earlier in *Node value & evidence* 33 section, where it was used to retrieve static marginal probabilities. The dependency of marginal probabilities on time makes the temporal beliefs matrix large. If a node has *X* outcomes and the slice count was set to *Y,* the matrix will have *X * Y* elements. The elements representing a single time slice are adjacent. Therefore, elements with indices *[0..X-1]* in the matrix are the beliefs for *t=0*, elements *[X..2*X-1]* are the beliefs for *t=1* and so on. The code snippet below iterates over the temporal beliefs and displays a single line of numbers for each time step. The numbers are marginal probabilities of node outcomes.

```
DSL_node *node = …;
```

```
int outcomeCount = node->Def()->GetNumberOfOutcomes();
int sliceCount = node->Network()->GetNumberOfSlices();
const DSL_Dmatrix *mtx = node->Val()->GetMatrix();
for (int sliceIdx = 0; sliceIdx < sliceCount; sliceIdx++)
{
    printf("\tt=%d:", sliceIdx);
    for (int i = 0; i < outcomeCount; i++)
    {
        printf(" %f", (*mtx)[sliceIdx * outcomeCount + i]);
    }
    printf("\n");
}
```

Only the plate nodes have the temporal beliefs. Other temporal node types have normal beliefs (the number of elements in the belief array is equal to their outcome count).

If a plate node has defined intervals or point values, the temporal mean and standard deviation are available. `DSL_nodeValue::GetTemporalMeanStdDev` returns two vectors or two arrays containing means and standard deviations for all time slices.

6.13 Continuous models

Graphical models, such as Bayesian networks, are not necessarily consisting of only discrete variables. They are, in fact, close relatives of systems of simultaneous structural equations. SMILE allows for constructing models consisting of equation nodes that are alternative, graphical representations of systems of simultaneous structural equations.

Tutorial 7 [94] contains a complete program demonstrating the use of continuous models.

6.13.1 Equation-based nodes

To create an equation node, use `DSL_EQUATION` node type when creating a node with `DSL_network::AddNode`. The node equation will be initialized to *id=0*, where *id* is the node identifier passed to `AddNode`. The definition of the equation node is represented by an object of the `DSL_equation` class. The code snippet below changes the default equation of the freshly created node (*x=0*) to an equation representing the standard Gaussian distribution: *x=Normal(0, 1)*.

```
DSL_network net;
int hx = net.AddNode(DSL_EQUATION, "x");
auto eqDefX = net.GetNode(hx)->Def<DSL_equation>();
eqDefX->SetEquation("x=Normal(0,1)");
```

SMILE defines many functions for use in node equations. The complete list of functions is available in the Equations [197] section in the reference chapter of this manual. Among these functions, there are random generators, each of which generates a single sample based on the passed parameters. In the example above, the `Normal` is the name of SMILE's random generator function which draws samples from normal distribution.

Node equations can reference other nodes by using their identifiers. Continuing with our code snippet:

```
int hy = net.AddNode(DSL_EQUATION, "y");
auto eqDefY = net.GetNode(hy)->Def<DSL_equation>();
eqDefY->SetEquation("y=2*x");
```

Second node is added and its equation is set to *y=2*x*, where *x* is a reference to the previously defined node. SMILE **adds an arc** between x and y automatically and it is not necessary to call `DSL_network::AddArc` before setting the equation referencing other nodes. Calling `AddArc` will change the child node equation by adding a term representing the parent node as a last term on the right hand side of the child equation. Assuming network with equation nodes a, b, c and d and d's equation set to *d=Normal(a, 1)+Normal(b, 2)*, calling `AddArc` to with node handles of c and d would rewrite d's equation to *d=Normal(a, 1)+Normal(b, 2)+c*.

If an arc is removed, either by calling `DSL_network::RemoveArc` or `DSL_network::DeleteNode` on one of the parents, the node equation will be rewritten as sum of the remaining parents. This ensures that equations and arcs are always in sync. If node b would be removed with `DeleteNode`, or an arc from b to d would be removed by `RemoveArc`, d's equation would become *d=a+c*.

Node identifier changes are propagated to all equations referring that node. If the identifier of the first node from the code snippet above were changed from *x* to *x_prime*, the equation of node *y* would change to *y=2*x_prime*.

The equation nodes can be unbounded, semi-bounded or bounded. The bounds are defined with `DSL_equation::SetBounds`.

See the `DSL_equation` 159 reference for details.

6.13.2 Continuous inference

To run inference in a continuous model, use `DSL_network::UpdateBeliefs`, the same method that is used in discrete networks.

Inference in continuous networks is based on stochastic sampling when there is no evidence in the network, or when the evidence is specified only for nodes without parents. Otherwise, inference is performed on a temporary discrete network, derived from the original continuous model. The definitions for the temporary discrete nodes are derived from the discretization intervals defined in each continuous node.

Equation nodes have their values represented by `DSL_equationEvaluation` class derived from `DSL_nodeVal`. Since the representation of the equation node value uses either stochastic samples or discretized beliefs, the `DSL_equationEvaluation` defines multiple methods not present in `DSL_nodeVal`. However, for setting and getting the evidence we can use (overridden) methods `SetDefintion` and `GetDefinition` without casting the `DSL_nodeVal` pointer to DSL_equationEvaluation.

To set evidence in an equation node, use `SetEvidence(double)` overload. In the snippet below we are assuming that `evNodeHandle` is the handle of the equation node.

```
int evNodeHandle = …
DSL_nodeVal *evVal = net.GetNode(evNodeHandle)->Val();
evVal.SetEvidence(1.5); // 1.5 is the evidence value
```

To retrieve the evidence, use `GetEvidence(double &)`.

Both types of inference algorithms use the lower and upper bounds defined for each equation node. To set the bounds, use `DSL_equation::SetBounds` method. Stochastic sampling can reject a sample when its value falls outside of the bounds defined for the node. You can control this behavior by `DSL_network::EnableRejectOutlierSamples` method. By default, outlier rejection is disabled.

Stochastic sampling can be controlled by setting the sample count with `DSL_network::SetNumberOfDiscretizationSamples`. Large number of samples provides better approximation of the joint probability distribution represented by the set of equations embedded in the continuous network but requires more time to complete. Output samples can be accessed directly with `DSL_equationEvaluation::GetSamples`. They can be also returned as histogram bins from `DSL_equationEvaluation::GetHistogram`. Statistics for the collected samples are available from `DSL_equationEvaluation::GetStats`, or `GetMean` and `GetStdDev`. If there were samples out of bounds, `DSL_equationEvaluation::HasSamplesOutOfBounds` returns `true`.

When discretization was used to perform inference, samples are not available. SMILE creates a temporary discrete model, where nodes are derived from the original model's equations and discretization intervals. The discretization intervals, just like equation bounds, are stored in equation node definitions (`DSL_equation` objects).

To set the discretization intervals, use `DSL_equation::SetDiscIntervals`. The method accepts a vector of intervals as its argument. The vector consists of string/double pairs. The first element of the pair is the interval identifier, which is not used during inference. The second element of the pair is the upper bound of the discretization interval. The lower bound of the interval with index j is defined by upper interval of the interval with index j-1. The lower bound of the first interval is defined by the lower bound defined for the node with a `DSL_equation::SetBounds` call.

Discretization intervals are used to obtain CPTs for the temporary discrete network. The CPTs are calculated by drawing a number of samples specified at the network level for each CPT column. The number of discretization samples is set to 10,000 by default but can be changed using the `DSL_network::SetNumberOfDiscretizationSamples` method. The size of the discretized CPT is a product of the number of node intervals and parent node intervals. Note that this may lead to excessive memory use when the node has many parents.

When discretization algorithm is used, the equation node value stores discretized beliefs (not samples). To check which inference algorithm was executed, use `DSL_equationEvaluation::IsDiscretized`. When `IsDiscretized` returns `true`, use `DSL_equationEvaluation::GetDiscBeliefs` to get access to the discretized beliefs. Note that mean and standard deviation are still available through `DSL_equationEvaluation::GetMean` and `GetStdDev` (but not `GetSampleStats`, because there are no samples).

See the `DSL_equationEvaluation` 171 reference for more information.

6.14 Hybrid models

Hybrid models are networks with both discrete and continuous nodes. Arcs in a hybrid network can link all combinations of node types, so it is possible to add an arc from a continuous to a discrete node, and from a discrete to a continuous node. Inference in hybrid models follows the rules defined for continuous nodes (sampling when there is no evidence in nodes with parents, discretization otherwise). Basically, hybrid models can be treated as continuous models with discrete nodes representing a specialized function, namely a conditional probability table specified by an array of numbers.

Adding arcs from a discrete to a continuous node is performed by including the discrete node identifier in the continuous node's equation (as is the case for the continuous to continuous arcs). The following discussion assumes that the reader is familiar with functions from which node equations are built (functions implemented in SMILE are described in detail in the *Equations* 197 section of the reference part of this manual).

For example, assuming that a node *c* is continuous and its only parent *d* is a discrete node with three outcomes *High*, *Medium* and *Low*, the equation for *c* may look as follows: `c=Normal(If(d="Medium"), 1, -1), 5)`. When in the process of sampling, we calculate the value of node *c*, we need the value of *d*. The equation says that c should be drawn from a normal distribution with standard deviation equal to 5, but with mean depending on value of *d*. If *d* is in state *Medium*, the mean will be 1 and -1 otherwise.

Reference to a discrete node can appear anywhere in an equation, for example, `c=log(1+d) or c=2^d`, where *d* is discrete. The value of *d* amounts simply to its state number. The equation in the example above could be thus rewritten as `c=Normal(If(d=1, 1, -1), 5)`. Index 1 refers to the (zero-based) second state of *d*, which is *Medium*. The original form, using a text literal, may be more readable.

Caution: SMILE will not modify the text literals representing the outcomes of discrete parent nodes if the outcome identifiers change. If the text literal cannot be associated with any parent node outcome, its value is evaluated as -1 (minus one).

In addition to the function `If()` or its equivalent, the ternary operator `?:`, the common functions to use with discrete nodes are *Switch* and *Choose*. For example, the equation for node *c* with three possible means of its normal distributions can look like this: `c=Normal(Switch(d, "High", 3.2, "Medium", 2.5, "Low", 1.4), 5)`. An alternative notation would be `c=Normal(Choose(d, 3.2, 2.5, 1.4), 5)`.

To add an arc from a continuous to a discrete node, use `DSL_network::AddArc` (the method used in discrete models). In such case, the discretization intervals of the parent node are considered to be the equivalent of the outcomes of discrete parent.

Tutorial 8 100 contains a complete program demonstrating the use of hybrid models.

6.15 Input and output

SMILE supports two types of network I/O:

- string-based, with `DSL_network::WriteString` and `ReadString`
- file-based, with `DSL_network::WriteFile` and `ReadFile`

The native format for SMILE networks is XDSL. The format is XML-based and the definition schema is available at BayesFusion documentation website (http://support.bayesfusion.com/docs/). When writing a network in this format to file, the `.xdsl` extension should be used.

XDSL is the only format supported by string I/O methods. File-based methods can read and write other formats. Depending on the feature parity between SMILE and the 3rd party software using other file types, some of the information may be lost. For complete list of supported file types, see the reference section for `DSL_network::ReadFile`.

By default, the file I/O methods infer the file format from the filename extension (the format can be also explicitly specified in the call).

```
DSL_errorH().RedirectToFile(stdout);
DSL_network net;
int res = net.ReadFile("my_network.xdsl");
if (DSL_OKAY != res)
```

```
{
    printf("ReadFile failed.\n");
}
```

In case of a read failure, the above program will send a specific error message to standard output (due to the earlier `RedirectToFile(stdout)` call).

6.16 Inference

SMILE includes functions for several popular Bayesian network inference algorithms, including the clustering algorithm, and several approximate stochastic sampling algorithms. To run the inference, use `DSL_network::UpdateBeliefs` method.

The default algorithm for discrete Bayesian networks is clustering over network pre-processed with relevance. To change the default Bayesian network algorithm, call `DSL_network::SetDefaultBNAlgorithm` and pass the chosen algorithm identifier as its parameter. For influence diagrams, the method is `DSL_network::SetDefaultIDAlgorithm`. Available algorithms are listed in the reference section for these methods.

`DSL_network::UpdateBeliefs` returns `DSL_OUT_OF_MEMORY` error code if the temporary data structures required to complete the inference require more memory than that available. In such case, or if the inference takes too long, consider taking advantage of SMILE's relevance reasoning layer. Relevance reasoning runs as a pre-processing step, which can lessen the complexity of later stages of inference algorithms. Relevance reasoning takes the target node set into account, therefore, to reduce the workload you should reduce the number of nodes set as targets if possible. Note that by default all nodes are targets (this is the case when no nodes were marked as such). If your network has 1,000 nodes and you only need the probabilities of 20 nodes, by all means call `DSL_network::SetTarget` on these nodes.

If changing the model to use *Noisy-MAX* 36 nodes is possible, then it is definitely worth trying. The inference can be performed efficiently on networks with Noisy-MAX nodes when Noisy-MAX decomposition is enabled. To enable it, call `DSL_network::EnableNoisyDecomp`. If enabled, the Noisy-MAX decomposition runs in the relevance layer and reduces the complexity of the subsequent phases of the inference algorithm. To further control the decomposition, you can call `DSL_network::SetNoisyDecompLimit`, which controls the maximal number of parents in the temporary structures managed by SMILE during inference.

Stochastic inference algorithms can be controlled by setting the number of generated samples with the `DSL_network::SetNumberOfSamples` method. Obviously, the more samples are generated, the more time it takes to complete the inference. The accuracy of posterior marginal probabilities changes with a square root of the number of samples.

To obtain the probability of evidence currently set in the network, call `DSL_network::CallProbEvidence`.

Confidence intervals for specific nodes can be calculated with `DSL_network::CalcConfidenceIntervals`.

6.17 User properties

To integrate data that are specific to your application into SMILE models, you can use SMILE’s user properties. User properties are lists of key/value pairs available at the `DSL_network` and `DSL_node` level. While it’s possible to use data structure not managed by SMILE (like `std::map<DSL_node *, YourObject>`) to extend the set of attributes associated with the nodes, that external data is not written by `DSL_network::WriteFile` and `WriteString`. On the other hand, user properties are stored in the XDSL files.

```
DSL_node *node = net.GetNode(handle);
// if that node has user properties,
// the loop below will print them out
DSL_userProperties &props = node->Info().UserProperties();
for (int i = 0; i < props.GetNumberOfProperties(); i ++)
{
    printf("%s=%s\n",
           props.GetPropertyName(i),
           props.GetPropertyValue(i));
}
```

Note that the property name is unique for the set of properties defined in given node. Property name follows the convention of SMILE identifiers: it is case-sensitive, starts with a letter, and contains letters, digits, and underscores.

To get access to network-level user properties, call `DSL_network::UserProperties` method.

6.18 Cases

SMILE includes management of cases, which allows users to save a partial or a complete evidence set as a case and retrieve this case at a later time. Cases are saved alongside the model (XDSL file format only), so when the model is loaded at a later time, all cases are available.

Internally, case information is stored in the `DSL_case` objects, which are managed by `DSL_network` (so your program does not create instances of this class directly). To add a case to the network, use `DSL_network::AddCase`, which returns a pointer to the newly created instance of the `DSL_case` object. To retrieve the case information, use `DSL_network::GetCase`.

Each case has a name (required), a category and a description (both optional). Once the case is created, your program can add evidence to the case with `DSL_case::AddEvidence`. Alternatively, you can copy current network evidence into the case with `DSL_case::NetworkToCase`.

To apply the evidence defined in the case, use `DSL_case::CaseToNetwork`.

```
DSL_case *c = net.GetCase("my_case");
for (int i = 0; i < c->GetNumberOfEvidence(); i ++)
{
    int handle, outcome;
    c->GetEvidence(i, handle, outcome);
    printf("%d %d %d\n", i, handle, outcome);
}
c->CaseToNetwork();
net.UpdateBeliefs();
```

6.19 Diagnosis

Diagnosis is one of the most successful applications of Bayesian networks. The ability of probabilistic knowledge representation techniques to perform a mixture of both predictive and diagnostic inference makes it very suitable for diagnosis. Bayesian networks can perform fusion of observations such as patient (or equipment) history and risk factors with symptoms and test results.

A Bayesian network built with SMILE represents various components of a system, possible faulty behaviors produced by the system (symptoms), along with results of possible diagnostic tests. The model essentially captures how possible defects of the system (whether it is a natural system, such as human body, or a human-made device, such as a car, an airplane, or a copier) can manifest themselves by error messages, symptoms, and test results.

To work with diagnosis, some nodes in the network should be designated as diagnostic faults, and some other nodes should become diagnostic observations. It is also possible to provide observation costs. The output of SMILE diagnostic algorithms is two-fold: (1) the posterior marginal probability of diagnostic faults, and (2) ranking of possible observations from the most to the least informative from the point of view of the current diagnostic focus. Depending on the selected diagnostic algorithm, the ranking is based on one of the following measures:

- an information-theoretic measure known as cross-entropy and expresses, for each observation node X individually, the expected reduction in entropy of the probability distribution over the specified subset of fault states after observing X. Cross-entropy is a utility-free measure of value of information and it gives a good idea about the value of the observations for diagnosing the disorder in question.

- a distance between two vectors representing the probabilities of pursued faults. SMILE can use multiple distance definitions, including Euclidean, cosine or cityblock.

6.19.1 Diagnostic roles

`DSL_node::SetDiagType` allows for setting for each node one of three possible diagnostic roles, listed in the following enumeration:

```
enum dsl_diagType { dsl_diagFault, dsl_diagObservation, dsl_diagAux };
```

- `dsl_diagFault` is used for diagnostic faults. Diagnostic fault nodes require at least one fault state. During a diagnostic session, SMILE calculates the probabilities of the fault states after a change in the set of observations. To set a state of a diagnostic fault node to be a fault, use `DSL_discDef::SetFaultOutcome`.

- `dsl_diagObservation` indicates diagnostic observation. Diagnostic observation nodes can be instantiated (set to specified evidence) during a diagnostic session. Observations can have default outcomes, which are instantiated automatically when a diagnostic session begins. To set the default outcome, call `DSL_discDef::SetDefaultOutcome`. To mark that the observation node should be included in the ranked list of observations, use `DSL_node::SetDiagRanked`. To make the observation mandatory, use `DSL_node::SetDiagMandatory`. The diagnostic session will not rank observations until all mandatory observations have been instantiated.

- `dsl_diagAux` is the default value, and is neither an observation nor a fault.

When `DSL_network` object is saved to a `.xdsl` file, the diagnostic roles of nodes are preserved.

For full API details, see the reference for `DSL_node` 138 and `DSL_discDef` 145.

6.19.2 Observation cost

Observing the value of a variable is often associated with a cost. For example, in order to observe the platelet count, one has to draw a blood sample and subject it to examination by professionals. Measuring the temperature of an air conditioner exhaust unit requires a technician's time. In order to perform an optimal diagnosis, one has to take into account the value of information along with the cost of obtaining it. For any diagnostic observation node, we can specify the cost of observing its value by calling the `DSL_node::SetCosts` method and read the cost by calling `DSL_node::GetCosts`.

The cost of performing a test can be expressed on some scale, like currency or time in minutes. It is also possible to specify negative number as cost. Negative cost indicate that tests so inexpensive that they should always be performed, for example checking the car model when diagnosing a car.

When observing the state of a node is independent of whether other nodes are observed or not, the cost is just a single number. Sometimes, however, the cost of observing a variable is not independent of observing other variables. For example, once a blood sample is drawn, performing additional tests on the sample is cheaper than performing these tests when no blood sample is available. The cost of measuring some parameter of a locomotive engine depends on whether the locomotive is in the shop or in the field. It may be much lower when the locomotive is in the shop. Taking off a locomotive cover may take a few hours but once it is removed, many tests become inexpensive. To represent such cases, use `DSL_network::AddArc` with its 3rd argument set to `dsl_costObserve`. This causes an observation cost arc to be added to the network. The observation cost arcs are not constrained by 'normal' arcs. When diagnostic observation has incoming cost arcs, its observation cost is a multi-dimensional matrix, just like conditional probability table, but with the last dimension size equal to one.

To remove a cost arc, call `DSL_network::RemoveArc` with `dsl_costObserve` as its last argument. `DSL_network::GetCostParents` and `GetCostChildren` return references to arrays containing handles of nodes that are cost parents and children of the specified node.

The observation costs are saved with the network when .xdsl format is used.

6.19.3 Diagnostic session

With diagnostic attributes, such as node roles and (optionally) observation costs defined, we can start a diagnostic session in order to obtain a ranked observation list. The diagnostic session is represented by `DSL_diagSession` object. Its constructor requires a reference to the `DSL_network`, where diagnostic information is defined. The session object provides an API to pursue faults, instantiate observations, and retrieve the statistics for observations. During the diagnostic session, your program should use the `DSL_diagSession` instance to read information from the network.

```
DSL_diagSession diag(net); // net is our DSL_network
```

The diagnostic session instantiates the mandatory observation during its initialization. To obtain a ranking of un-instantiated observations, we need to calculate their cross-entropy, which is a dynamic measure, depending on the already instantiated observations and the fault or faults selected as the focus of reasoning (these faults are called "pursued faults"). There is no default selection of pursued faults when `DSL_diagSession` is initialized and we need to specify a fault or faults to pursue. From the diagnostic session point of view, the fault is a node/state pair (because one fault node can have more than one faulty state). The node/state pairs are referenced by their indices in the session's data. Do not use fault node handles directly when setting the pursued fault. If needed, convert the node handle/state pair to fault index with `DSL_diagSession::FindFault`.

If we work with a network containing only one fault, we can pass zero to `DSL_diagSession::SetPursuedFault`. For general purpose diagnostic application, the initial selection of the pursued fault can be performed with `DSL_diagSession::FindMostLikelyFault`:

```
diag.UpdateFaultBeliefs();
int faultIndex = diag.FindMostLikelyFault();
diag.SetPursuedFault(faultIndex);
```

Please note that we need to calculate the fault probabilities with `DSL_diagSession::UpdateFaultBeliefs`, otherwise `FindMostLikelyFault` will not work. With pursued fault initialized, we can execute the main diagnostic algorithm, which is computationally complex and involves a series of runs of a belief updating algorithm.

```
int res = diag.UpdateTestStrengths();
```

If the status returned from `DSL_diagSession::UpdateTestStrengths` is `DSL_OKAY`, we can retrieve the vector of observations with their diagnostic values. Note that only ranked observations are included in this list. The vector is not sorted, but `std::sort` algorithm can be trivially applied.

```
vector<DSL_diagTestInfo> stats = diag.GetTestStatistics();
sort(stats.begin(), stats.end(),
    [](auto lhs, auto rhs) { return lhs.strength > rhs.strength; });
```

Each `DSL_diagTestInfo` structure contains an observation node handle, which can be used to create a user interface with a list of unperformed tests and their diagnostic values. Typical diagnostic application shows the lists of faults, and performed/unperformed observations. Users can change pursued fault(s) and instantiate observations. To instantiate an observation, call `DSL_diagSession::InstantiateObservation` with the observation node handle and its selected outcome index. This sets the evidence in the corresponding observation node and clears the internal data structures in the diagnostic session. `DSL_diagSession::ReleaseObservation` removes the evidence from an observation node.

Do not call `DSL_nodeVal::SetEvidence` and `ClearEvidence` directly on the observation nodes during the diagnostic session.

See `DSL_diagSession` 183 reference for API details.

6.19.4 Distance and entropy-based measures

The diagnostic measures calculated for observations depend on the pursued fault set. The pursued faults define the focus of reasoning, and the changes in their probabilities are the inputs to the measure algorithms. To select the algorithm, use `DSL_diagSession::SetSingleFaultAlgorithm` and `SetMultiFaultAlgorithm`. The output of the algorithm is a single number for each uninstantiated observation.

The available algorithms for single fault diagnosis are:

- Max probability change (default): for each observation, the maximum change is taken over the absolute values of the change in probability of the pursued fault. Note that this is a signed measure, meaning that some values may be negative (when the largest magnitude of fault probability change represents the probability decrease).

- Cross-entropy: the insight from the Max probability change is limited in the sense of not telling us a key piece of information how likely each of the changes happens. For example, a positive test result for cancer will make a huge change it the probability of cancer. However, the probability of seeing a positive test may be very small in a generally healthy person. So, effectively the expected amount of diagnostic information from performing this test is rather small. Cross-entropy is an information-theoretic measure that takes into account both the amount of information flowing from observing individual states of an observation variable and the probabilities of observing this states. A high cross-entropy indicates a high expected contribution of observing a variable to the probability of the pursued fault. Cross-entropy is unsigned.

- Normalized cross-entropy: the cross entropy divided by the current value of the entropy of the pursued fault node

For multi-fault diagnosis, the following algorithms are available:

- Max probability change (default): the value of the measure is the maximum change of probability over all pursued faults and outcomes of the observation. This is a signed measure.

- Euclidean distance, or L2 Norm: calculates the distance between two vectors in Euclidean space, where the vector coordinates are the probabilities of the pursued faults before and after the observation. The distance is normalized to ensure that a change from impossible (all coordinates are zero) to certain (all coordinates are ones) is equal to 1.0. For each observation, the selected value is the greatest distance over all observation outcomes. The larger the distance, the larger the impact of the observation.

- Cityblock distance: as above, but using cityblock metric

- Averaged L2 and cityblock distance

- Cosine distance, or cosine similarity: calculated between two fault probability vectors. Since vector coordinates are probabilities, and therefore non-negative, this measure will always produce non-negative values (despite general cosine similarity range between -1 and 1).

- A family of six entropy-based measures, which require calculation of the joint probability distribution over all pursued faults, which is computationally prohibitive, it is necessary to use approximations of the joint probability distribution. The approximations are based on two strong assumptions about dependencies among them: (1) complete independence (this is taken by the first group of approaches) and (2) complete dependence (this is taken by the second group of approaches). Each of the two extremes is divided into three groups: (1) At Least One, (2) Only One, and (3) All. These refer to different partitioning of the combinations of diseases in cross-entropy calculation.

- Two marginal probability-based measures, which are much faster than the independence/dependence-based joint probability distribution approaches but it is not as accurate because they make a stronger assumption about the joint probability distribution. Entropy calculations in this approach are based purely on the marginal probabilities of the pursued faults. The two algorithms that use the Marginal Probability Approach differ essentially in the function that they use to select the tests to perform. Both functions are scaled so that they return values between 0 and 1. Entropy/Marginal 1 uses a function without the support for maximum distance and its minimum is reached when all probabilities of the faults are equal to 0.5. Entropy/Marginal 2 uses a function that has support for maximum distance and is continuous in the domain [0,1].

6.20 Sensitivity analysis

`DSL_sensitivity` class contains an implementation of a sensitivity analysis algorithm proposed by Kjaerulff and van der Gaag (2000).

Given a set of target nodes, the algorithm calculates a complete set of derivatives of the posterior probability distributions over the target nodes over each numerical parameter of the Bayesian network. Note that these derivatives are dependent on the evidence currently set in the network and they give an indication of importance of precision of network numerical parameters for calculating the posterior probabilities of the targets. If the derivative is large for a parameter x, then a small change in x may lead to a large difference in the posteriors of the targets. If the derivative is small, then even a large change in the parameter will make little difference in the posteriors.

For each node, the sensitivity can be obtained as an actual value of the derivative, and also as a set of coefficients, which define the dependency between the target node posterior and the specific CPT parameter. The general form of the function is linear rational:

$$y = (Ax + B) / (Cx + D)$$

where y is the target posterior (calculated by inference algorithm) and x is the value of the specific CPT parameter. SMILE calculates the coefficients A, B, C, D and the derivative y' at the current value of x. The formula for the derivative is:

$$y' = (AD - BC) / (Cx + D)^2$$

Because the denominator of the above equation is positive, the sign of the derivative is constant for all x, and hence the function is monotonic (or constant). By substituting 0 and 1 for x in the first formula (because x is a probability) we can calculate how much the posterior will change if x is modified in its CPT. The range is defined by:

$$y_1 = B / D, \; y_2 = (A + B) / (C+D)$$

The sign of AD-BC determines which of the two values is minimum and which is maximum.

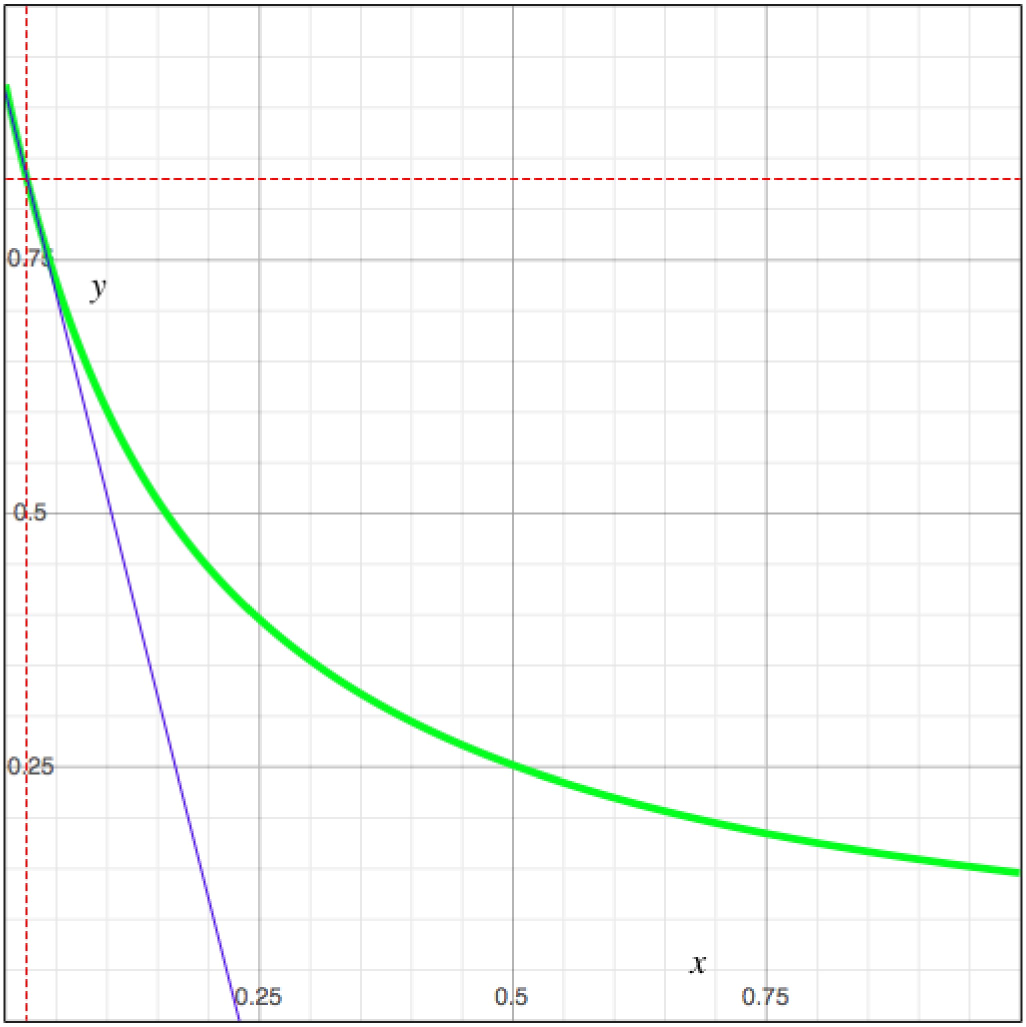
As an illustration, let us use the *HeparII.xdsl* model, which is distributed with GeNIe and is also available from BayesFusion's online model repository at https://repo.bayesfusion.com/. The choice of the target node, evidence and CPT parameter is motivated only by the large value of the derivative:

x = P(bilirubin=a19_7 | Hyperbilirubinemia=absent, PBC=absent, Cirrhosis=absent, gallstones=absent, ChHepatitis=absent)
y = P(PBC=present | ast=149_40, irregular_liver=present, bilirubin=a19_7, proteins=a5_2)

Note that the right-hand side of the vertical bar in the definition of x determines the position of the parameter within the CPT of the *bilirubin* node, while the right-hand side of the vertical bar in the definition of y represents the evidence set in the network. The sensitivity output for x=0.02126152 (the current value of the parameter in the CPT) and y=0.82906518 (the calculated posterior probability) is:

y'=-3.96207, A=0, B=6.18863e-5, C=0.00035673, D=6.70612e-5

With known coefficients, we can now visualize the relationship between x and y:



The green curve in the image represents the function *y(x)*, the blue line represents the tangent of the green curve at the current value of *x* (its slope being equal to *y'*) and the red dashed lines make it easier to locate the current *(x,y)* on the curve.

To perform sensitivity analysis, start with setting the target(s) in the network, instantiate the `DSL_sensitivity` object, and invoke its `Calculate` method. Next, read the values of derivatives and/or the function coefficients. The code snippet below uses the *HeparII.xdsl* network and calculates the sensitivity for the same CPT parameter and target node. To save space, it uses the `SetEvidenceById` helper function from *Tutorial 2* 75.

```
DSL_network net;
int res = net.ReadFile("HeparII.xdsl");
if (DSL_OKAY != res) return res;

SetEvidenceById(net, "ast", "a149_40");
SetEvidenceById(net, "irregular_liver", "present");
SetEvidenceById(net, "bilirubin", "a19_7");
SetEvidenceById(net, "proteins", "a5_2");

DSL_sensitivity sens;
res = sens.Calculate(net);
```

```
if (DSL_OKAY != res) return res;

int target = net.FindNode("PBC");
if (target < 0) return target;
DSL_sensitivity::Target targetNodeAndOutcome(target, 0);

int nodeUnderStudy = net.FindNode("bilirubin");
if (nodeUnderStudy < 0) return nodeUnderStudy;

// the CPT parameter index is specified as linear, but it is
// of course possible to obtain it by passing the outcomes
// of nodeUnderStudy and its parents to DSL_Dmatrix::CoordinatesToIndex
const int paramIndex = 285;

// just to ensure we have right paramIndex
const DSL_Dmatrix *cpt =
    net.GetNode(nodeUnderStudy)->Def()->GetMatrix();
printf("x=%g\n", (*cpt)[paramIndex]);

const DSL_Dmatrix *yPrim =
    sens.GetSensitivity(nodeUnderStudy, targetNodeAndOutcome);
double slope = (*yPrim)[paramIndex];

std::vector<const DSL_Dmatrix *> coeffs;
sens.GetCoefficients(nodeUnderStudy, targetNodeAndOutcome, coeffs);

printf("y'=%g\nA=%g B=%g C=%g D=%g\n", slope,
    (*coeffs[0])[paramIndex], (*coeffs[1])[paramIndex],
    (*coeffs[2])[paramIndex], (*coeffs[3])[paramIndex]);
```

The program outputs is:

```
x=0.0212615
y'=-3.96207
A=0 B=6.18863e-05 C=0.00035673 D=6.70612e-05
```

The `DSL_sensitivity` object owns the matrices containing the derivatives and the parameters - do not call `delete` on the pointers returned from its getter methods.

For performance reasons, the sensitivity algorithm works on a network processed by SMILE's relevance algorithms. Those nodes that do not affect target node's posteriors are dropped early by the relevance algorithms. The matrix objects returned for theses nodes by `GetSensitivity` and `GetCoefficients` will be empty (`DSL_Dmatrix::GetSize` will return zero). Unlike the example above, your code should include a check for the matrix size and assume that the target is not sensitive to changes in the node under study if the sensitivity matrix is empty.

Programs calculating sensitivity do not normally focus on a specific parameter. Instead, they iterate over the coefficients and derivatives over the entire CPT for one or more nodes. If your program only needs the maximum sensitivity value, use the overloaded `DSL_sensitivity::GetMaxSensitivity` methods. The returned numbers represent the maxima for the entire network, a single target node, a single node under study, or a combination of the two.

Sensitivity analysis can be performed on influence diagrams. In such case, the terminal utility node becomes the target for sensitivity calculations. The calculated coefficients depend on the outcomes for the utility indexing parents. Use the `DSL_sensitivity::GetConfigCount` and `SetConfig` to switch between different combinations of indexing parent outcomes.

See `DSL_sensitivity` 186 reference for API details.

6.21 Datasets

Data used by SMILE for learning and network validation are stored in objects of the `DSL_dataset` class. A data set is a table-like structure. Its columns are called *variables* and rows are called *records*.

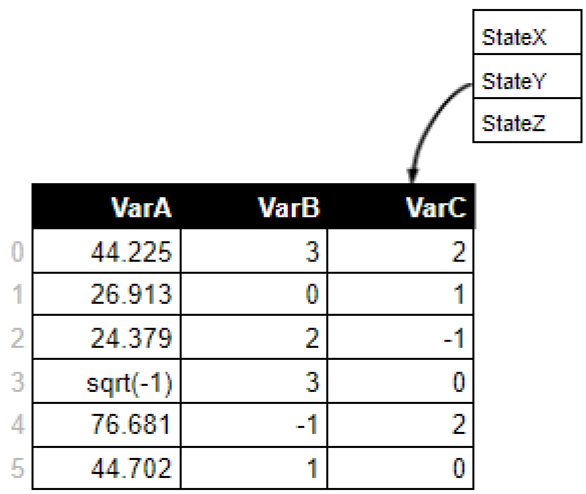
The contents of a data set can be loaded from a text file. Alternatively, your program can initialize columns in a data set and then add records containing actual data.

6.21.1 Text file I/O

You can load the contents of a data set from a text file by calling `DSL_dataset::ReadFile`. For the illustration purposes, let us assume that we have a comma separated text file with the following data:

```
VarA,VarB,VarC
44.225,3,StateZ
26.913,0,StateY
24.379,2,*
*,3,StateX
76.681,*,StateZ
44.702,1,StateX
```

After the `DSL_dataset::ReadFile` call, the data set will be structured as below:



	VarA	VarB	VarC
0	44.225	3	2
1	26.913	0	1
2	24.379	2	-1
3	sqrt(-1)	3	0
4	76.681	-1	2
5	44.702	1	0

- variable `VarA` is continuous. The value missing in the fourth record was replaced by `sqrt(-1)`.
- variable `VarB` is discrete. The value missing in the fifth record was replaced by `-1`.

- variable `VarC` is also discrete, and has the string labels associated with its integer values. The value missing in the third record was replaced by `-1`.

You can fine-tune the parsing by passing a `DSL_datasetParseParams` structure to `ReadFile`. The following code should be used if the data file has no header line with the names of the columns, missing values are marked by a "N/A" string and missing values in discrete columns should be replaced by 999:

```
DSL_dataset ds;
DSL_datasetParseParams params;
params.columdIdsPresent = false;
params.missingValueToken = "N/A";
params.missingInt = 999;
int res = ds.ReadFile("datafile.txt", &params);
```

To write the contents of the data set to a text file, use `DSL_dataset::WriteFile`. You can customize the field separator, the missing value marker, etc. by passing a `DSL_datasetWriteParams` structure to `WriteFile`. The code snippet below writes a comma-separated file, which includes a header line with column names and uses "(none)" as a marker for missing values:

```
DSL_dataset ds;
// create or load dataset here
DSL_datasetWriteParams params;
params.columdIdsPresent = true;
params.missingValueToken = "(none)";
params.separator = ',';
int res = ds.WriteFile("datafile.csv", &params);
```

6.21.2 Discrete and continuous variables

If the data for learning or validation comes from a source other than a text file, you will need to programmatically initialize the structure and the contents of the data set. Consider the example data set from the previous chapter - it had three variables, of which the first was continuous and other two were discrete. The code to create the structure of this data set looks as follows:

```
DSL_dataset ds;
ds.AddFloatVar("Var1");
ds.AddIntVar("Var2");
ds.AddIntVar("Var3");
vector<string> stateNames({"StateX","StateY",""StateZ""});
ds.SetStateNames(ds.FindVariable("Var3"), stateNames);
```

`DSL_dataset::FindVariable` was used to get the index of the variable with a known identifier. An alternative approach in the example above would use a hard coded index, as we know which variables were added to the data set just prior to `DSL_dataset::SetStateNames`.

You can set the number of records in the data set upfront with a call to `DSL_dataset::SetNumberOfRecords`, or call `AddEmptyRecord` for each record that you plan to append to the data set at a later time.

```
ds.AddEmptyRecord();
int recIdx = ds.GetNumberOfRecords() - 1;
ds.SetFloat(0, recIdx, 44.225);
ds.SetInt(1, recIdx, 3);
ds.SetInt(2, recIdx, 2);
```

Note that depending on the type of the variable, you need to use either `DSL_dataset::SetFloat` or `SetInt`. In the example above, the last variable has associated state names, but they are not used for data entry.

To mark an element of the variable as missing, use `DSL_dataset::SetMissing`. After a call to `AddEmptyRecord`, all elements of the last record in the data set are missing.

The following code snippet displays contents of a data set. It uses `DSL_dataset::IsDiscrete` to determine the type of the variable. For discrete variables, the state names vector returned by `DSL_dataset::GetStateNames` is also checked. If the vector is empty, there are no strings associated with the integer variable values.

```
int varCount = ds.GetNumberOfVariables();
int recCount = ds.GetNumberOfRecords();
for (int r = 0; r < recCount; r ++)
{
    for (int v = 0; v < varCount; v++)
    {
        if (v > 0) printf(",");
        if (ds.IsMissing(v, r))
        {
            printf("N/A");
        }
        else if (ds.IsDiscrete(v))
        {
            int x = ds.GetInt(v, r);
            const vector<string> &states = ds.GetStateNames(v);
            if (states.empty())
            {
                printf("%d", x);
            }
            else
            {
                printf("%s", states[x].c_str());
            }
        }
        else
        {
            printf("%f", ds.GetFloat(v, r));
        }
    }
    printf("\n");
}
```

6.21.3 Generating data from a network

Given that a Bayesian network is a representation of the joint probability distribution over its variables, we can generate a data set based on this distribution using the `DSL_dataGenerator` class. The `DSL_dataGenerator::GenerateData` method has three overloads, which write the data to the following outputs:

- `DSL_dataset` object
- a text file specified by a file name
- any object derived from the pure abstract class `DSL_dataGeneratorOutput`

The following code snippet generates 200 records with 1/4 of the values marked as missing and writes the output to a text file "out.txt":

```
DSL_network net;
// create or load network here
DSL_dataGenerator gen(net);
gen.SetNumberOfRecords(200);
gen.SetMissingValuePercent(25);
gen.GenerateData("out.txt");
```

6.21.4 Discretization

To discretize a variable in a data set, use the `DSL_dataset::Discretize` method. To convert a variable to a discrete variable with 10 states, use the following code:

```
vector<double> edges;
int res = ds.Discretize(varIdx, DSL_dataset::Hierarchical, 10, "discState", edges);
```

Note that after the discretization the variable will have state names starting with the specified prefix ("discState" in this case). The names will be suffixed with numeric values derived from calculated discretization intervals. These intervals are also returned in the (optional) `edges` parameter.

The supported discretization methods are defined in the `DSL_dataset::DiscretizeAlgorithm` enum:

- `Hierarchical`
- `UniformWidth`
- `UniformCount`

Discretization works on both continuous and discrete variables.

6.22 Learning

Learning in SMILE can perform two tasks:

- structure learning: create a new network from a data set
- parameter learning: refine parameters (CPTs) in an existing network

SMILE also supports network validation, which is frequently used after learning to evaluate the results.

6.22.1 Learning network structure

The following classes can be used to learn a `DSL_network` from a `DSL_dataset`:

- `DSL_bs`: Bayesian Search, a hill climbing procedure guided by scoring heuristic with random restarts
- `DSL_nb`: Naive Bayes
- `DSL_tan`: Tree Augmented Naive Bayes, semi-naive method based on the Bayesian Search approach

- `DSL_abn`: Augmented Naive Bayes, another semi-naive method based on the Bayesian Search approach

In the simplest scenario, just declare the object representing a learning algorithm and call its `Learn` method:

```
DSL_dataset ds;
ds.ReadFile("myfile.txt");
DSL_network net;
DSL_bs bayesianSearch;
int res = bayesianSearch.Learn(ds, net);
```

If the algorithm succeeds, the learning algorithm returns `DSL_OKAY` and the `DSL_network` passed as an argument to `Learn` is the output of the learning. Note that every variable in the data set takes part in the learning process. If your data comes from the text file and you want to exclude some variables, use `DSL_dataset::RemoveVar`.

After learning the structure, each of the algorithms listed above performs parameter learning by counting cases. Counting cases can be used instead of a more sophisticated EM parameter learning algorithm, because the learning data set does not contain missing data entries.

The code example above used the default settings for Bayesian Search. To tweak the learning process, you can set some public data members in the learning object before calling its `Learn` method. The example below sets the number of iterations and maximum number of parents:

```
DSL_bs bayesianSearch;
bayesianSearch.nrIteration = 10;
bayesianSearch.maxParents = 4;
int res = bayesianSearch.Learn(ds, net);
```

All settings for the learning algorithms are described in detail in the *Reference* 114 section.

SMILE also contains the `DSL_pc` class, which implements the PC structure learning algorithm (the algorithm name is an acronym derived from its inventors' first names, Peter and Clark). This algorithm also uses `DSL_dataset` as data source, but instead of `DSL_network` it learns the `DSL_pattern` object, which is a graph with directed and undirected edges, which is not guaranteed to be acyclic.

`DSL_bs` and `DSL_pc` objects contain a public data member of `DSL_bkgndKnowledge` type. It can be used to pass the background knowledge to the learning algorithm. The background knowledge influences the learned structure by:

- forcing arcs between specified variables
- forbidding arcs between specified variables
- ordering specified groups of variables by temporal tiers: in the resulting structure, there will be no arcs from nodes in higher tiers to nodes in lower tiers

The example below forces an arc from X to Y and forbids an arc from Z to Y. It is assumed that the data set contains variables with the identifers used in the calls to `DSL_dataset::FindVariable`.

```
DSL_network net;
DSL_bs baySearch;
int varX = ds.FindVariable("X");
int varY = ds.FindVariable("Y");
int varZ = ds.FindVariable("Z");
```

```
baySearch.bkk.forcedArcs.push_back(make_pair(varX, varY));
baySearch.bkk.forbiddenArcs.push_back(make_pair(varZ, varY));
res = baySearch.Learn(ds, net);
```

Tutorial 9 [104] contains a program, which performs structure learning using Bayesian Search, Tree Augmented Naive Bayes and PC.

6.22.2 Learning network parameters

To learn or refine parameters in an existing `DSL_network` object, you can use the EM algorithm implemented in `DSL_em` class. As with structure learning, the data comes in `DSL_dataset` object. However, the network and the data must be matched to ensure that learning algorithm knows the relationship between the data set variables and network nodes. If the variables and nodes have identical identifiers, you can use the `DSL_dataset::MatchNetwork` method:

```
DSL_dataset ds;
DSL_network net;
// load network and data here
vector<DSL_datasetMatch> matching;
string errMsg;
res = ds.MatchNetwork(net, matching, errMsg);
if (DSL_OKAY == res)
{
    DSL_em em;
    res = em.Learn(ds, net, matching);
}
```

If your network and data cannot be automatically matched with `MatchNetwork`, you can build a vector of `DSL_datasetMatch` structures in your own code. `DSL_datasetMatch` has `node` and `column` members representing node handle and variable index, respectively. For each node/variable pair you need one element of the matching vector.

`DSL_em::Learn` can be fine-tuned with various algorithm settings - see the *DSL_em* [188] reference section for details. The method can also return log likelihood, ranging from minus infinity to zero, which is a measure of fit of the model to the data:

```
DSL_em em;
double logLik
res = em.Learn(ds, net, matching, &logLik);
```

It is possible to exclude some nodes from learning. These fixed nodes do not change their CPTs during parameter learning.

```
DSL_em em;
double loglik;
vector<int> fixedNodes;
// add node handles to fixedNodes here
res = em.Learn(ds, net, matching, fixedNodes, &loglik);
```

6.22.3 Validation

To evaluate the predictive quality of a network you can use the `DSL_validator` class. The `DSL_validator` constructor requires references to `DSL_dataset` and `DSL_network` objects to be specified. To properly match the network and data, the constructor also requires a vector of `DSL_datasetMatch` objects (as did `DSL_em::Learn` method). After the validator object is constructed, you need to specify which nodes in the network are considered class nodes by calling `DSL_validator::AddClassNode` method. Validation requires at least one class node.

During the validation, for each record in the data set, the variables matched to non-class nodes are used to set the evidence. The posterior probabilities are then calculated and for each class node the outcome with the highest probability is selected as the predicted outcome. The prediction is compared to the outcome variable (in the data set) that is associated with the class node. The number of matches and calculated posteriors are used to obtain the accuracy, confusion matrix, ROC (including the AUC) and calibration curves.

Validation can be performed without parameter learning, using `DSL_validator::Test` method, or with parameter learning using `DSL_validator::KFold` and `LeaveOneOut` methods. *K*-fold cross-validation divides the data set into *K* parts of equal size, trains the network on *K-1* parts, and tests it on the last, *K*th part. The process is repeated *K* times, with a different part of the data being selected for testing. Leave-one-out is an extreme case of *K*-fold, in which *K* is equal to the number of records in the data set.

The example below performs *K*-fold crossvalidation with five folds using one class node.

```
DSL_dataset ds;
DSL_network net;
vector<DSL_datasetMatch> matching;
// load network and dataset, create the matching here
DSL_validator validator(ds, net, matching);
int classNodeHandle = net.FindNode("someNodeIdentifier");
validator.AddClassNode(classNodeHandle);
DSL_em em;
// optionally tweak the EM options here
int res = validator.KFold(em, 5);
if (DSL_OKAY == res)
{
    double acc;
    validator.GetAccuracy(classNodeHandle, 0, acc);
    vector<pair<double, double> roc;
    vector<double> thresholds;
    double auc;
    validator.CreateROC(classNodeHandle, 0, roc, thresholds, auc);
    printf("Accuracy=%f Area under the curve=%f\n", acc, auc);
}
```

See the `DSL_validator` 180 reference for more details.

This page is intentionally left blank.

Tutorials

7 Tutorials

Each tutorial in this section is contained in a single `.cpp` file. The file defines a function named `TutorialN` (with N being an ordinal number of the tutorial) and one or more helper functions. We show how the tutorials work by interleaving actual code with textual explanation. The complete code ready to be copied and pasted is located at the end of each subsection.

To ensure that tutorials are self-contained, there is some duplicated code defined in functions declared as static. For example, the function `CreateCptNode` is present in tutorials 1, 6, and 8.

If you want to create a program containing a single tutorial only, add a simple main function at the bottom of the `tutorialN.cpp` file, for example:

```
int main()
{
   return Tutorial1();
}
```

To run all tutorials you can use the `main.cpp` file included in the next sub-section of this manual. This approach requires compiling and linking `main.cpp` AND all of the `tutorialN.cpp` files.

You can also download tutorial sources from our documentation site at https://support.bayesfusion.com/docs.

C++11 standard is used (required for range-based for loops and `std::initializer_list`), compiling tutorial code with gcc or clang requires the -std=c++11, or -std option specifying a later standard. To compile all tutorials with gcc, use the following command:

```
g++ -std=c++11 main.cpp tutorial?.cpp -I./smile -L./smile -lsmile
```

The example command assumes that SMILE files are located in the `./smile` subdirectory. Replace `./smile` used with `-I` and `-L` with the path to directory containing SMILE's .h and .a files if the library can be found in other location in your file system.

Visual C++ users can load `SmileTutorial.sln` directly (included with the tutorial source files). Project properties are set to use the compiler from Visual Studio 2015 (toolkit v140). The toolkit version should be changed to the version appropriate for the version of the Visual Studio used, and correct SMILE for Visual Studio distribution used with tutorials.

7.1 main.cpp

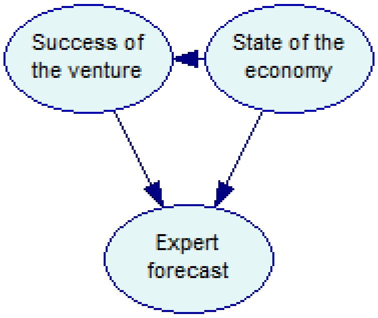
```
// main.cpp
// Run all SMILE tutorials.

// smile_license.h contains your personal license key
#include "smile_license.h"

int Tutorial1();
int Tutorial2();
int Tutorial3();
int Tutorial4();
int Tutorial5();
int Tutorial6();
```

```
int Tutorial7();
int Tutorial8();
int Tutorial9();

int main()
{
    static int (* const f[])() =
    {
        Tutorial1, Tutorial2, Tutorial3,
        Tutorial4, Tutorial5, Tutorial6,
        Tutorial7, Tutorial8, Tutorial9
    };
    for (int i = 0; i < sizeof(f) / sizeof(f[0]); i++)
    {
        int r = f[i]();
        if (r)
        {
            return r;
        }
    }
    return 0;
}
```

7.2 Tutorial 1: Creating a Bayesian Network

Consider a slight twist on the problem described in the *Hello SMILE!* 22 section of this manual.

The twist will include adding an additional variable *State of the economy* (with the identifier *Economy*) with three outcomes (*Up*, *Flat*, and *Down*) modeling the developments in the economy. These developments are relevant to the decision, as they are impacting expert forecast. When the economy is heading *Up*, our expert makes more optimistic predictions, when it is heading *Down*, the expert makes more pessimistic predictions for the same venture. This is reflected by a directed arc from the node *State of the economy* to the node *Expert forecast*. *State of the economy* also impacts the probability of venture being successful, which we model by adding another arc.



We will show how to create this model using SMILE and how to save it to disk. In subsequent tutorials, we will show how to enter observations (evidence), how to perform inference, and how to retrieve the results of SMILE's calculations.

We start by redirecting error and warning messages to the console and declaring our network variable. The three nodes in the network are subsequently created by calling a helper function `CreateCptNode` declared beforehand and defined below the `Tutorial1` function.

```
DSL_errorH().RedirectToFile(stdout);
DSL_network net;
int e = CreateCptNode(net, "Economy", "State of the economy",
    { "Up", "Flat", "Down" }, 160, 40);
int s = CreateCptNode(net, "Success", "Success of the venture",
    { "Success", "Failure" }, 60, 40);
int f = CreateCptNode(net, "Forecast", "Expert forecast",
    { "Good", "Moderate", "Poor" }, 110, 140);
```

Before connecting the nodes with arcs, let us have a look at the `CreateCptNode` function.

```
static int CreateCptNode(
    DSL_network &net, const char *id, const char *name,
    std::initializer_list<const char *> outcomes, int xPos, int yPos)
{
    int handle = net.AddNode(DSL_CPT, id);
    DSL_node *node = net.GetNode(handle);
    node->SetName(name);
    node->Def()->SetNumberOfOutcomes(outcomes);
    DSL_rectangle &position = node->Info().Screen().position;
    position.center_X = xPos;
    position.center_Y = yPos;
    position.width = 85;
    position.height = 55;
    return handle;
}
```

The function creates a CPT node with a specified identifier, name, outcomes, and position on the screen. The handle returned by `DSL_network::AddNode` is converted to a `DSL_node*` pointer with a call to `DSL_network::GetNode`. Next, we set the name of the node with `DSL_node::SetName`.

The default outcomes of the CPT nodes are named *State0* and *State1*. To change the number of outcomes and their identifiers in one step, we call `DSL_nodeDef::SetNumberOfOutcomes`. To obtain a pointer to a node definition object, we use `DSL_node::Def`. The outcome identifiers are specified in the `std::initializer_list` passed as `outcomes` parameter to `CreateCptNode`.

Finally, we set the location and size of the node by direcly writing to the `DSL_rectangle` object in node's screen info block.

Back in the `Tutorial1` function, we add arcs between nodes.

```
net.AddArc(e, s);
net.AddArc(s, f);
net.AddArc(e, f);
```

Next step is to initialize the conditional probability tables of the nodes. See the *Multidimensional arrays* 31 section in this manual for the description of the CPT memory layout. For each of the three nodes in our network, we obtain a pointer to the node definition object. To change the numbers (probabilities) in the CPT, we will call `DSL_nodeDef::SetDefinition` method and pass a `std::initializer_list<double>` with the probabilities. We show the code snippet for the *Success* node below, two other nodes are initialized in the same way.

```
res = net.GetNode(s)->Def()->SetDefinition({
```

```
        0.3, // P(Success=S|Economy=U)
        0.7, // P(Success=F|Economy=U)
        0.2, // P(Success=S|Economy=F)
        0.8, // P(Success=F|Economy=F)
        0.1, // P(Success=S|Economy=D)
        0.9  // P(Success=F|Economy=D)
    });
```

With CPTs initialized, our network is complete. We write its contents to the `tutorial1.xdsl` file.

```
    res = net.WriteFile("tutorial1.xdsl");
```

Tutorial 2 75 will load this file and perform inference. The split between tutorials is artificial, your program can use networks right after its creation without the need to write/read to/from the file system.

7.2.1 tutorial1.cpp

```
// tutorial1.cpp
// Tutorial1 creates a simple network with three nodes,
// then saves it as XDSL file to disk.

#include "smile.h"
#include <cstdio>

static int CreateCptNode(
    DSL_network& net, const char* id, const char* name,
    std::initializer_list<const char*> outcomes, int xPos, int yPos);


int Tutorial1()
{
    printf("Starting Tutorial1...\n");

    // show errors and warnings in the console
    DSL_errorH().RedirectToFile(stdout);

    DSL_network net;

    int e = CreateCptNode(net, "Economy", "State of the economy",
        { "Up", "Flat", "Down" }, 160, 40);

    int s = CreateCptNode(net, "Success", "Success of the venture",
        { "Success", "Failure" }, 60, 40);

    int f = CreateCptNode(net, "Forecast", "Expert forecast",
        { "Good", "Moderate", "Poor" }, 110, 140);

    net.AddArc(e, s);
    net.AddArc(s, f);
    net.AddArc(e, f);

    int res = net.GetNode(e)->Def()->SetDefinition({
        0.2, // P(Economy=U)
        0.7, // P(Economy=F)
        0.1  // P(Economy=D)
    });
```

```
    if (DSL_OKAY != res)
    {
        return res;
    }

    res = net.GetNode(s)->Def()->SetDefinition({
        0.3, // P(Success=S|Economy=U)
        0.7, // P(Success=F|Economy=U)
        0.2, // P(Success=S|Economy=F)
        0.8, // P(Success=F|Economy=F)
        0.1, // P(Success=S|Economy=D)
        0.9  // P(Success=F|Economy=D)
    });
    if (DSL_OKAY != res)
    {
        return res;
    }

    res = net.GetNode(f)->Def()->SetDefinition({
        0.70, // P(Forecast=G|Success=S,Economy=U)
        0.29, // P(Forecast=M|Success=S,Economy=U)
        0.01, // P(Forecast=P|Success=S,Economy=U)
        0.65, // P(Forecast=G|Success=S,Economy=F)
        0.30, // P(Forecast=M|Success=S,Economy=F)
        0.05, // P(Forecast=P|Success=S,Economy=F)
        0.60, // P(Forecast=G|Success=S,Economy=D)
        0.30, // P(Forecast=M|Success=S,Economy=D)
        0.10, // P(Forecast=P|Success=S,Economy=D)
        0.15, // P(Forecast=G|Success=F,Economy=U)
        0.30, // P(Forecast=M|Success=F,Economy=U)
        0.55, // P(Forecast=P|Success=F,Economy=U)
        0.10, // P(Forecast=G|Success=F,Economy=F)
        0.30, // P(Forecast=M|Success=F,Economy=F)
        0.60, // P(Forecast=P|Success=F,Economy=F)
        0.05, // P(Forecast=G|Success=F,Economy=D)
        0.25, // P(Forecast=G|Success=F,Economy=D)
        0.70  // P(Forecast=G|Success=F,Economy=D)
    });
    if (DSL_OKAY != res)
    {
        return res;
    }

    res = net.WriteFile("tutorial1.xdsl");
    if (DSL_OKAY != res)
    {
        return res;
    }

    printf("Tutorial1 complete: Network written to tutorial1.xdsl\n");
    return DSL_OKAY;
}


static int CreateCptNode(
```

```
      DSL_network &net, const char *id, const char *name,
    std::initializer_list<const char *> outcomes, int xPos, int yPos)
{
    int handle = net.AddNode(DSL_CPT, id);
    DSL_node *node = net.GetNode(handle);
    node->SetName(name);
    node->Def()->SetNumberOfOutcomes(outcomes);
    DSL_rectangle &position = node->Info().Screen().position;
    position.center_X = xPos;
    position.center_Y = yPos;
    position.width = 85;
    position.height = 55;
    return handle;
}
```

7.3 Tutorial 2: Inference with a Bayesian Network

Tutorial 2 starts with the model that we have previously created. We will perform multiple calls to a Bayesian inference algorithm through the `DSL_network::UpdateBeliefs`, starting with network without any evidence. After each `UpdateBeliefs` call, we retrieve and print the posterior probabilities of all outcomes of the nodes.

The `Tutorial2` function itself is very simple and delegates work to helper functions declared at the top of the file and defined below `Tutorial2`.

The model is loaded with the `DSL_network::ReadFile`. We exit the program if the status code `ReadFile` returned is not `DSL_OKAY`.

```
DSL_network net;
int res = net.ReadFile("tutorial1.xdsl");
if (DSL_OKAY != res)
{
   printf("Load failed, did you run Tutorial1 before Tutorial2?\n");
   return res;
}
```

UpdateBeliefs is followed by the call to PrintAllPosteriors helper.

```
printf("Posteriors with no evidence set:\n");
net.UpdateBeliefs();
PrintAllPosteriors(net);
```

`PrintAllPosteriors()` displays posterior probabilities calculated by `UpdateBeliefs()` and stored in node values. To iterate over all nodes, we use `DSL_network::GetFirstNode` and `GetNextNode`. In the body of the loop, we call another locally defined helper function, `PrintPosteriors()`.

```
for (int h = net.GetFirstNode(); h >= 0; h = net.GetNextNode(h))
{
   PrintPosteriors(net, h);
}
```

`PrintPosteriors()` converts node handle to node pointer. Then it checks if the node has evidence set with `DSL_nodeVal::IsEvidence`; if this is the case, the name of the evidence state is displayed. Otherwise, the function iterates over all states and displays the posterior probability of each state. We retrieve the state identifiers with `DSL_nodeDef::GetOutcomeIds`. The posterior probabilities are stored in a `DSL_Dmatrix` object, which has just one dimension in this case. The size of the dimension is equal to the number of states. `DSL_nodeVal::GetMatrix`. returns a pointer to the matrix.

```
static void PrintPosteriors(DSL_network &net, int handle)
{
    DSL_node *node = net.GetNode(handle);
    const char* nodeId = node->GetId();
    const DSL_nodeVal* val = node->Val();
    if (val->IsEvidence())
    {
        printf("%s has evidence set (%s)\n",
            nodeId, val->GetEvidenceId());
    }
    else
    {
        const DSL_idArray& outcomeIds = *node->Def()->GetOutcomeIds();
        const DSL_Dmatrix& posteriors = *val->GetMatrix();
        for (int i = 0; i < posteriors.GetSize(); i++)
        {
            printf("P(%s=%s)=%g\n", nodeId, outcomeIds[i], posteriors[i]);
        }
    }
}
```

Going back to `Tutorial2` function, we repeatedly call another locally defined helper function to change evidence, update network, and display posteriors:

```
printf("\nSetting Forecast=Good.\n");
ChangeEvidenceAndUpdate(net, "Forecast", "Good");
printf("\nAdding Economy=Up.\n");
ChangeEvidenceAndUpdate(net, "Economy", "Up");
printf("\nChanging Forecast to Poor, keeping Economy=Up.\n");
ChangeEvidenceAndUpdate(net, "Forecast", "Poor");
printf("\nRemoving evidence from Economy, keeping Forecast=Poor.\n");
ChangeEvidenceAndUpdate(net, "Economy", NULL);
```

Let us examine the `ChangeEvidenceAndUpdate` helper. The function is just a series of calls to various SMILE methods. The node identifier passed as the 2nd argument is validated and converted to `DSL_node` pointer. If the 3rd argument is NULL, the evidence is cleared with `DSL_nodeVal::ClearEvidence`. Otherwise, new evidence is set with `DSL_nodeVal::SetEvidence`, which has an overload accepting the outcome id directly. Finally, `DSL_network::UpdateBeliefs` performs inference and the posteriors are displayed by our `PrintAllPosteriors` function.

```
static int ChangeEvidenceAndUpdate(
    DSL_network &net, const char *nodeId, const char *outcomeId)
{
    DSL_node* node = net.GetNode(nodeId);
    if (NULL == node)
    {
        return DSL_OUT_OF_RANGE;
    }
    int res;
    if (NULL != outcomeId)
    {
        res = node->Val()->SetEvidence(outcomeId);
    }
```

```
        else
        {
            res = node->Val()->ClearEvidence();
        }
        if (DSL_OKAY != res)
        {
            return res;
        }
        res = net.UpdateBeliefs();
        if (DSL_OKAY != res)
        {
            return res;
        }
        PrintAllPosteriors(net);
        return DSL_OKAY;
    }
```

Tutorial 2 ends here.

7.3.1 tutorial2.cpp

```
// tutorial2.cpp
// Tutorial2 loads the XDSL file created by Tutorial1,
// then performs the series of inference calls,
// changing evidence each time.

#include "smile.h"
#include <cstdio>

static int ChangeEvidenceAndUpdate(
    DSL_network &net, const char *nodeId, const char *outcomeId);

static void PrintAllPosteriors(DSL_network &net);


int Tutorial2()
{
    printf("Starting Tutorial2...\n");

    DSL_errorH().RedirectToFile(stdout);

    // load the network created by Tutorial1
    DSL_network net;
    int res = net.ReadFile("tutorial1.xdsl");
    if (DSL_OKAY != res)
    {
        printf(
            "Network load failed, did you run Tutorial1 before Tutorial2?\n");
        return res;
    }

    printf("Posteriors with no evidence set:\n");
    net.UpdateBeliefs();
    PrintAllPosteriors(net);
```

```
    printf("\nSetting Forecast=Good.\n");
    ChangeEvidenceAndUpdate(net, "Forecast", "Good");

    printf("\nAdding Economy=Up.\n");
    ChangeEvidenceAndUpdate(net, "Economy", "Up");

    printf("\nChanging Forecast to Poor, keeping Economy=Up.\n");
    ChangeEvidenceAndUpdate(net, "Forecast", "Poor");

    printf("\nRemoving evidence from Economy, keeping Forecast=Poor.\n");
    ChangeEvidenceAndUpdate(net, "Economy", NULL);

    printf("\nTutorial2 complete.\n");
    return DSL_OKAY;
}


static void PrintPosteriors(DSL_network &net, int handle)
{
    DSL_node *node = net.GetNode(handle);
    const char* nodeId = node->GetId();
    const DSL_nodeVal* val = node->Val();
    if (val->IsEvidence())
    {
        printf("%s has evidence set (%s)\n",
            nodeId, val->GetEvidenceId());
    }
    else
    {
        const DSL_idArray& outcomeIds = *node->Def()->GetOutcomeIds();
        const DSL_Dmatrix& posteriors = *val->GetMatrix();
        for (int i = 0; i < posteriors.GetSize(); i++)
        {
            printf("P(%s=%s)=%g\n", nodeId, outcomeIds[i], posteriors[i]);
        }
    }
}

static void PrintAllPosteriors(DSL_network &net)
{
    for (int h = net.GetFirstNode(); h >= 0; h = net.GetNextNode(h))
    {
        PrintPosteriors(net, h);
    }
}

static int ChangeEvidenceAndUpdate(
    DSL_network &net, const char *nodeId, const char *outcomeId)
{
    DSL_node* node = net.GetNode(nodeId);
    if (NULL == node)
    {
        return DSL_OUT_OF_RANGE;
    }
```

```
    int res;
    if (NULL != outcomeId)
    {
        res = node->Val()->SetEvidence(outcomeId);
    }
    else
    {
        res = node->Val()->ClearEvidence();
    }
    if (DSL_OKAY != res)
    {
        return res;
    }

    res = net.UpdateBeliefs();
    if (DSL_OKAY != res)
    {
        return res;
    }
    PrintAllPosteriors(net);
    return DSL_OKAY;
}
```

7.4 Tutorial 3: Exploring the contents of a model

This tutorial will not perform any calculations. Instead, we will display the information about the network structure (nodes and arcs) and parameters (in this case, conditional probability tables). The `Tutorial3` function itself is again very simple. We load the network created by `Tutorial1` and for each node invoke a locally defined helper function `PrintNodeInfo()`. This is where real work is done. The first node attribute displayed is its name, followed by the outcome identifiers:

```
DSL_node *node = net.GetNode(nodeHandle);
printf("Node: %s\n", node->GetName());
printf("  Outcomes:");
const DSL_idArray &outcomes = *node->Def()->GetOutcomeIds();
for (const char* oid : outcomes)
{
    printf(" %s", oid);
}
printf("\n");
```

The parents of the node follow.

```
const DSL_intArray &parents = net.GetParents(nodeHandle);
if (!parents.IsEmpty())
{
    printf("  Parents:");
    for (int p: parents)
    {
        printf(" %s", net.GetNode(p)->GetId());
    }
    printf("\n");
}
```

Node's children are next. The code fragment is virtually identical to the iteration over parents above.

Finally, the node definition is checked. If its type is DSL_CPT (general chance node) or DSL_TRUTHTABLE (deterministic discrete node), we retrieve the probabilities from the node's definition with the DSL_nodeDef::GetMatrix. Note that all the nodes in the network created in *Tutorial 1* 71 are DSL_CPT. The if statement is there to keep the function general enough to be copied and pasted into other program using SMILE.

```
const DSL_nodeDef *def = node->Def();
int defType = def->GetType();
printf("  Definition type: %s\n", def->GetTypeName());
if (DSL_CPT == defType || DSL_TRUTHTABLE == defType)
{
    const DSL_Dmatrix &cpt = *def->GetMatrix();
    PrintMatrix(net, cpt, outcomes, parents);
}
```

PrintMatrix is another locally defined helper function. It iterates over all entries in the CPT. For each entry, the outcome of the node and its parents' outcomes are displayed along with the probability value. Let us inspect the main loop of the function:

```
for (int elemIdx = 0; elemIdx < mtx.GetSize(); elemIdx++)
{
    const char *outcome = outcomes[coords[dimCount - 1]];
    printf("    P(%s", outcome);
    if (dimCount > 1)
    {
        printf(" | ");
        for (int parentIdx = 0; parentIdx < dimCount - 1; parentIdx++)
        {
            if (parentIdx > 0) printf(",");
            DSL_node *parentNode = net.GetNode(parents[parentIdx]);
            const DSL_idArray &parentOutcomes =
                *parentNode->Def()->GetOutcomeIds();
            printf("%s=%s",
                parentNode->GetId(), parentOutcomes[coords[parentIdx]]);
        }
    }
    double prob = mtx[elemIdx];
    printf(")=%g\n", prob);
    mtx.NextCoordinates(coords);
}
```

The loop uses both linear index (elemIdx variable) and multidimensional coordinates (coords variable of DSL_intArray type, initialized to zeros before the loop). Both are kept in sync, the equivalent of increasing linear index by one (elemIdx++) is a call to DSL_Dmatrix::NextCoordinates (mtx.NextCoordinates(coords)). Note that we could use elemIdx to convert into coordinates during each iteration with IndexToCoordinates. Conversely, it is also possible to convert coords into linear index with DSL_Dmatrix::CoordinatesToIndex. While it is not important for this tutorial, the coordinate-based loop performance over large CPTs is computationally more efficient when NextCoordinates approach is used (NextCoordinates , on the other hand, is more efficient than IndexToCoordinates). Of course, the iteration with plain linear index only will be fastest.

The node outcome is the rightmost entry in the coordinates. The parents' outcome indices start from the left at index 0 in the coords. Part of the Tutorial3 output for the node *Forecast* is show below. All lines starting with "P"(were printed by PrintMatrix.

```
Node: Expert forecast
 Outcomes: Good Moderate Poor
```

```
    Parents: Success Economy
    Definition type: CPT
      P(Good | Success=Success,Economy=Up)=0.7
      P(Moderate | Success=Success,Economy=Up)=0.29
      P(Poor | Success=Success,Economy=Up)=0.01
      P(Good | Success=Success,Economy=Flat)=0.65
      P(Moderate | Success=Success,Economy=Flat)=0.3
      P(Poor | Success=Success,Economy=Flat)=0.05
      P(Good | Success=Success,Economy=Down)=0.6
      P(Moderate | Success=Success,Economy=Down)=0.3
      P(Poor | Success=Success,Economy=Down)=0.1
      P(Good | Success=Failure,Economy=Up)=0.15
      P(Moderate | Success=Failure,Economy=Up)=0.3
      P(Poor | Success=Failure,Economy=Up)=0.55
      P(Good | Success=Failure,Economy=Flat)=0.1
      P(Moderate | Success=Failure,Economy=Flat)=0.3
      P(Poor | Success=Failure,Economy=Flat)=0.6
      P(Good | Success=Failure,Economy=Down)=0.05
      P(Moderate | Success=Failure,Economy=Down)=0.25
      P(Poor | Success=Failure,Economy=Down)=0.7
```

Tutorial 3 ends here.

7.4.1 tutorial3.cpp

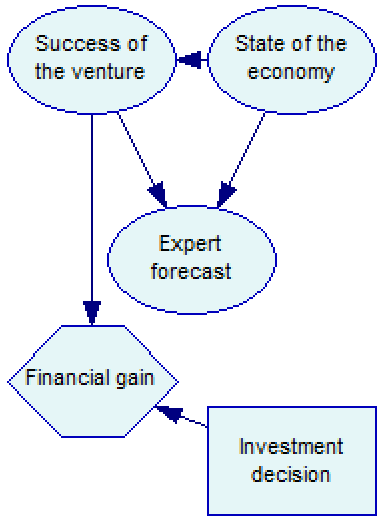
```
// tutorial3.cpp
// Tutorial3 loads the XDSL file and prints the information
// about the structure (nodes and arcs) and the parameters
// (conditional probabilities of the nodes) of the network.

#include "smile.h"
#include <cstdio>

static void PrintNodeInfo(DSL_network &net, int nodeHandle);


int Tutorial3()
{
    printf("Starting Tutorial3...\n");

    DSL_errorH().RedirectToFile(stdout);

    // load the network created by Tutorial1
    DSL_network net;
    int res = net.ReadFile("tutorial1.xdsl");
    if (DSL_OKAY != res)
    {
        printf(
            "Network load failed, did you run Tutorial1 before Tutorial3?\n");
        return res;
    }

    for (int h = net.GetFirstNode(); h >= 0; h = net.GetNextNode(h))
    {
        PrintNodeInfo(net, h);
```

```
    }

    printf("\nTutorial3 complete.\n");
    return DSL_OKAY;
}


// PrintMatrix displays each probability entry in the matrix in the separate
// line, preceeded by the information about node and parent outcomes the entry
// relates to.
// The coordinates of the matrix are ordered as P1,...,Pn,S
// where Pi is the outcome index of i-th parent and S is the outcome of the node
// for which this matrix is the CPT.
static void PrintMatrix(
    DSL_network &net, const DSL_Dmatrix &mtx,
    const DSL_idArray &outcomes, const DSL_intArray &parents)
{
    int dimCount = mtx.GetNumberOfDimensions();
    DSL_intArray coords(dimCount);
    coords.FillWith(0);

    // elemIdx and coords will be moving in sync
    for (int elemIdx = 0; elemIdx < mtx.GetSize(); elemIdx++)
    {
        const char *outcome = outcomes[coords[dimCount - 1]];
        printf("    P(%s", outcome);

        if (dimCount > 1)
        {
            printf(" | ");
            for (int parentIdx = 0; parentIdx < dimCount - 1; parentIdx++)
            {
                if (parentIdx > 0) printf(",");
                DSL_node *parentNode = net.GetNode(parents[parentIdx]);
                const DSL_idArray &parentOutcomes =
                    *parentNode->Def()->GetOutcomeIds();
                printf("%s=%s",
                    parentNode->GetId(), parentOutcomes[coords[parentIdx]]);
            }
        }

        double prob = mtx[elemIdx];
        printf(")=%g\n", prob);

        mtx.NextCoordinates(coords);
    }
}


// PrintNodeInfo displays node attributes:
//   name, outcome ids, parent ids, children ids, CPT probabilities
static void PrintNodeInfo(DSL_network &net, int nodeHandle)
{
    DSL_node *node = net.GetNode(nodeHandle);
    printf("Node: %s\n", node->GetName());
```

```
    printf("  Outcomes:");
    const DSL_idArray &outcomes = *node->Def()->GetOutcomeIds();
    for (const char* oid : outcomes)
    {
        printf(" %s", oid);
    }
    printf("\n");

    const DSL_intArray &parents = net.GetParents(nodeHandle);
    if (!parents.IsEmpty())
    {
        printf("  Parents:");
        for (int p: parents)
        {
            printf(" %s", net.GetNode(p)->GetId());
        }
        printf("\n");
    }

    const DSL_intArray &children = net.GetChildren(nodeHandle);
    if (!children.IsEmpty())
    {
        printf("  Children:");
        for (int c: children)
        {
            printf(" %s", net.GetNode(c)->GetId());
        }
        printf("\n");
    }

    const DSL_nodeDef *def = node->Def();
    int defType = def->GetType();
    printf("  Definition type: %s\n", def->GetTypeName());
    if (DSL_CPT == defType || DSL_TRUTHTABLE == defType)
    {
        const DSL_Dmatrix &cpt = *def->GetMatrix();
        PrintMatrix(net, cpt, outcomes, parents);
    }
}
```

7.5 Tutorial 4: Creating an Influence Diagram

We will further expand the model created in *Tutorial 1* 71 and turn it into an influence diagram. To this effect, we will add a decision node *Investment decision* and a utility node *Financial gain*. The decision will have two possible states: *Invest* and *DoNotInvest*, which will be the two decision options under consideration. Which option is chosen will impact the financial gain and this will be reflected by a directed arc from *Investment decision* to *Financial gain*. Whether the venture succeeds of fails will also impact the financial gain and this will be also reflected by a directed arc from *Success of the venture* to *Financial gain*.



We will show how to create this model using SMILE and how to save it to disk. In the subsequent tutorial, we will show how to enter observations (evidence), how to perform inference, and how to retrieve the utilities calculated for the *Financial gain* node.

The programs starts by reading the file, just like *Tutorial 2* [75] and *Tutorial 3* [79]. We convert the identfier of the node *Success* to node handle, which we will use later to create an arc between *Success* and *Gain*. Two new nodes will be created by calling a `CreateNode` helper function, which is a slightly modified version of the `CreateCptNode` from *Tutorial 1* [71]. The difference is that we now want to create different types of nodes. Therefore, `CreateNode` has one additional input parameter, an integer for specifying the node type. Another difference is that `CreateNode` needs to be able to add utility nodes, which do not have outcomes. The function checks the number of outcome identifiers passed in, and the call to `DSL_nodeDef::SetNumberOfOutcomes` is skipped when there are none. We will need this check for our utility node, which has no outcomes.

Back in `Tutorial4` function, we add nodes and arcs:

```
int i = CreateNode(net, DSL_LIST, "Invest", "Investment decision",
    { "Invest", "DoNotInvest" }, 160, 240);
int g = CreateNode(net, DSL_TABLE, "Gain", "Financial gain",
    {}, 60, 200);
net.AddArc(i, g);
net.AddArc(s, g);
```

Note that `DSL_LIST` is the node type identifier for decision nodes. `DSL_TABLE` is the node type identifier for utility nodes. Decision nodes do not have numeric parameters, but utility nodes do. The structure of the matrix in utility node's definition is similar to the CPT with the exception of the last dimension, which is always set to one (as there are no outcomes). Node *Gain* has two parents with two outcomes each and size of its definition is 2 x 2 x 1 = 4. The program specifies four numbers for the utilities.

```
res = net.GetNode(g)->Def()->SetDefinition({10000, -5000, 500, 500});
```

The influence diagram is now complete. We write its contents to file and exit the function. Tutorial 4 is now complete, *Tutorial 5* [86] will load the file and perform the inference.

7.5.1 tutorial4.cpp

```
// tutorial4.cpp
// Tutorial4 loads the XDSL file file created by Tutorial1
```

```
// and adds decision and utility nodes, which transforms
// a Bayesian Network (BN) into an Influence Diagram (ID).

#include "smile.h"
#include <cstdio>


static int CreateNode(
    DSL_network& net, int nodeType, const char* id, const char* name,
    std::initializer_list<const char*> outcomes, int xPos, int yPos);


int Tutorial4()
{
    printf("Starting Tutorial4...\n");

    DSL_errorH().RedirectToFile(stdout);

    DSL_network net;
    // load the network created by Tutorial1
    int res = net.ReadFile("tutorial1.xdsl");
    if (DSL_OKAY != res)
    {
        printf(
            "Network load failed, did you run Tutorial1 before Tutorial4?\n");
        return res;
    }

    int s = net.FindNode("Success");
    if (s < 0)
    {
        printf("Success node not found.");
        return s;
    }

    int i = CreateNode(net, DSL_LIST, "Invest", "Investment decision",
        { "Invest", "DoNotInvest" }, 160, 240);

    int g = CreateNode(net, DSL_TABLE, "Gain", "Financial gain",
        {}, 60, 200);

    net.AddArc(i, g);
    net.AddArc(s, g);

    res = net.GetNode(g)->Def()->SetDefinition({10000, -5000, 500, 500});
    res = net.WriteFile("tutorial4.xdsl");
    if (DSL_OKAY != res)
    {
        return res;
    }

    printf("Tutorial4 complete: Influence diagram written to tutorial4.xdsl\n");
    return DSL_OKAY;
}
```

```
static int CreateNode(
    DSL_network &net, int nodeType, const char *id, const char *name,
    std::initializer_list<const char*> outcomes, int xPos, int yPos)
{
    int handle = net.AddNode(nodeType, id);
    DSL_node *node = net.GetNode(handle);
    node->SetName(name);
    if (outcomes.size() > 0)
    {
        node->Def()->SetNumberOfOutcomes(outcomes);
    }
    DSL_rectangle &position = node->Info().Screen().position;
    position.center_X = xPos;
    position.center_Y = yPos;
    position.width = 85;
    position.height = 55;
    return handle;
}
```

7.6 Tutorial 5: Inference in an Influence Diagram

This tutorial loads the influence diagram created in *Tutorial 4* 83. We will perform multiple inference calls and display calculated utilities.

The tutorial starts with now-familiar sequence of redirecting error messages, loading the file and obtaining the handle to the node *Financial gain*. When we invoke `DSL_network::UpdateBeliefs` for the first time, the model has no evidence. A local helper function, `PrintFinancialGain`, is called to print out the utilities.

```
static void PrintFinancialGain(DSL_network &net, int gainHandle)
{
    DSL_node *node = net.GetNode(gainHandle);
    const char *nodeName = node->GetName();
    printf("%s:\n", nodeName);
    const DSL_nodeVal *val = node->Val();
    const DSL_Dmatrix &mtx = *val->GetMatrix();
    const DSL_intArray &parents = val->GetIndexingParents();
    PrintMatrix(net, mtx, "Utility", NULL, parents);
}
```

The function prints the name of the node specified by its 2nd parameter (which points always to node *Financial gain* in this tutorial), then prepares the input arguments for the `PrintMatrix` function: the node value matrix (utilties) and an array with handles of nodes indexing the utilities (these are all uninstantiated decision nodes and all nodes that have not been observed but should have been observed because they have outgoing arcs that enter decision nodes). `PrintMatrix` in this manual is a slightly modified version of the function from *Tutorial 3* 79. The changes are needed to properly display utility matrix with its last dimension set to 1, as utility nodes have no outcomes.

The calculated expected utilities without evidence suggest that we should not invest. Here is the relevant output from our program:

```
Financial gain:
   Utility(Invest=Invest)=-1850
   Utility(Invest=DoNotInvest)=500
```

Next, we model the analyst's forecast by setting *Forecast* to *Good* with `DSL_nodeVal::SetEvidence`, and recalculate the probabilities and utilities.

```
res = net.GetNode("Forecast")->Val()->SetEvidence("Good");
```

Based on the output from the program we should invest:

```
Financial gain:
   Utility(Invest=Invest)=4455.78
   Utility(Invest=DoNotInvest)=500
```

Now we observe the state of the economy and conclude that it is growing. We set the outcome *Up* as evidence in the *Economy* node with another `DSL_nodeVal::SetEvidence` call. Growing economy makes our chances even better:

```
Financial gain:
   Utility(Invest=Invest)=5000
   Utility(Invest=DoNotInvest)=500
```

This concludes Tutorial 5.

7.6.1 tutorial5.cpp

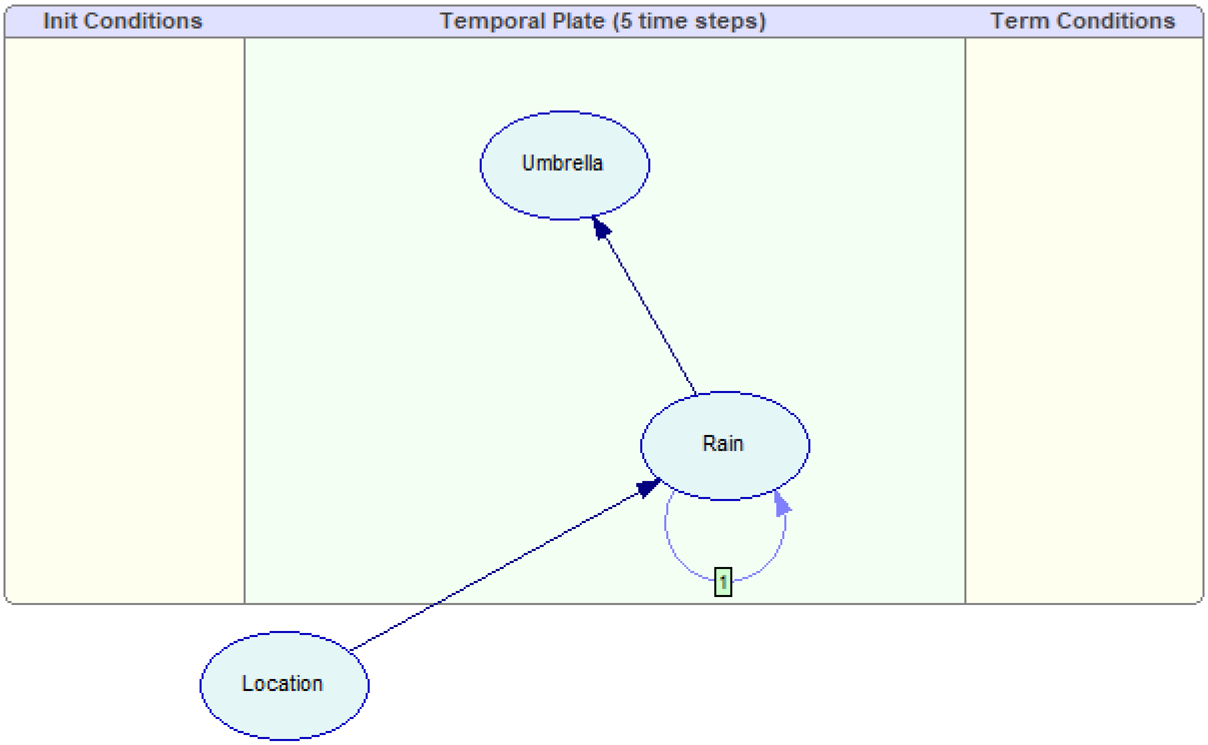
```
// tutorial5.cpp
// Tutorial5 loads the XDSL file created by Tutorial4,
// then performs the series of inference calls,
// changing evidence each time.

#include "smile.h"
#include <cstdio>


static void PrintFinancialGain(DSL_network &net, int gainHandle);


int Tutorial5()
{
    printf("Starting Tutorial5...\n");

    DSL_errorH().RedirectToFile(stdout);

    DSL_network net;
    // load the network created by Tutorial4
    int res = net.ReadFile("tutorial4.xdsl");
    if (DSL_OKAY != res)
    {
        printf(
            "Network load failed, did you run Tutorial4 before Tutorial5?\n");
        return res;
    }

    int gain = net.FindNode("Gain");
    if (gain < 0)
    {
        printf("Gain node not found.");
        return gain;
```

```
    }

    printf("Running UpdateBeliefs with no evidence set.\n");
    net.UpdateBeliefs();
    PrintFinancialGain(net, gain);

    printf("\nSetting Forecast=Good.\n");
    res = net.GetNode("Forecast")->Val()->SetEvidence("Good");
    if (DSL_OKAY != res)
    {
        return res;
    }
    net.UpdateBeliefs();
    PrintFinancialGain(net, gain);

    printf("\nAdding Economy=Up\n");
    res = net.GetNode("Economy")->Val()->SetEvidence("Up");
    if (DSL_OKAY != res)
    {
        return res;
    }
    net.UpdateBeliefs();
    PrintFinancialGain(net, gain);

    printf("\nTutorial5 complete.\n");
    return DSL_OKAY;
}


// PrintMatrix displays each probability entry in the matrix in the separate
// line, preceeded by the information about node and parent outcomes the entry
// relates to.
// The coordinates of the matrix are ordered as P1,...,Pn,S
// where Pi is the outcome index of i-th parent and S is the outcome of the node.
// If node type is utility, then S collapses and the last coordinate is
// always zero.
static void PrintMatrix(
    DSL_network &net, const DSL_Dmatrix &mtx, const char *prefix,
    const DSL_idArray *outcomes, const DSL_intArray &parents)
{
    int dimCount = mtx.GetNumberOfDimensions();
    DSL_intArray coords(dimCount);
    coords.FillWith(0);

    // elemIdx and coords will be moving in sync
    for (int elemIdx = 0; elemIdx < mtx.GetSize(); elemIdx++)
    {
        if (NULL != outcomes)
        {
            const char *outcome = (*outcomes)[coords[dimCount - 1]];
            printf("    %s(%s", prefix, outcome);
        }
        else
        {
            printf("    %s(", prefix);
```

```
        }

        if (dimCount > 1)
        {
            if (NULL != outcomes)
            {
                printf("|");
            }

            for (int parentIdx = 0; parentIdx < dimCount - 1; parentIdx++)
            {
                if (parentIdx > 0) printf(",");
                DSL_node *parentNode = net.GetNode(parents[parentIdx]);
                const DSL_idArray &parentOutcomes =
                    *parentNode->Def()->GetOutcomeIds();
                printf("%s=%s",
                    parentNode->GetId(), parentOutcomes[coords[parentIdx]]);
            }
        }

        double prob = mtx[elemIdx];
        printf(")=%g\n", prob);

        mtx.NextCoordinates(coords);
    }
}

static void PrintFinancialGain(DSL_network &net, int gainHandle)
{
    DSL_node *node = net.GetNode(gainHandle);
    printf("%s:\n", node->GetName());
    const DSL_nodeVal *val = node->Val();
    PrintMatrix(net, *val->GetMatrix(), "Utility", NULL, val->GetIndexingParents());
}
```

7.7 Tutorial 6: A dynamic model

Consider the following example, inspired by (Russell & Norvig, 1995), in which a security guard at some secret underground installation works on a shift of seven days and wants to know whether it is raining on the day of her return to the outside world. Her only access to the outside world occurs each morning when she sees the director coming in, with or without an umbrella. Furthermore, she knows that the government has two secret underground installations: one in Pittsburgh and one in the Sahara, but she does not know which one she is guarding. For each day *t*, the set of evidence contains a single variable $Umbrella_t$ (observation of an umbrella carried by the director) and the set of unobservable variables contains $Rain_t$ (a propositional variable with two states *true* and *false*, denoting whether it is raining) and *Location* (with two possible states: *Pittsburgh* and *Sahara*). The prior probability of rain depends on the geographical location and on whether it rained on the previous day. For simplicity, we do not use the initial and terminal condition nodes in this tutorial.



The model contains three discrete chance nodes (CPT), created with a call to `CreateCptNode` helper function, just like in nodes in *Tutorial 1* [71]. The *Rain* and *Umbrella* nodes are marked as belonging to the temporal plate by calling `DSL_network::SetTemporalType`:

```
net.SetTemporalType(rain, dsl_temporalType::dsl_plateNode);
net.SetTemporalType(umb, dsl_temporalType::dsl_plateNode);
```

Two of the three arcs in the model are created by `DSL_network::AddArc`. We create the temporal arc expressing the dependency of $Rain_t$ on $Rain_{t-1}$ by the `AddTemporalArc` method. Please note the temporal order of the arc passed as the 3rd parameter:

```
net.AddArc(loc, rain);
net.AddTemporalArc(rain, rain, 1);
net.AddArc(rain, umb);
```

CPTs for the nodes are initialized with `DSL_nodeDef::SetDefinition`, as in *Tutorial 1* [71]. However, the node *Rain* requires **two** CPTs, because it has an incoming temporal arc of order 1. The first CPT is initialized with a call to `DSL_nodeDef::SetDefinition`, the second (temporal) CPT is initialized `DSL_nodeDef::SetTemporalDefinition`:

```
res = rainDef->SetTemporalDefinition(1, {
    0.7,   // P(Rain=true |Location=Pittsburgh,Rain[t-1]=true)
    0.3,   // P(Rain=false|Location=Pittsburgh,Rain[t-1]=true)
    0.3,   // P(Rain=true |Location=Pittsburgh,Rain[t-1]=false)
    0.7,   // P(Rain=false|Location=Pittsburgh,Rain[t-1]=false)
    0.001, // P(Rain=true |Location=Sahara,Rain[t-1]=true)
    0.999, // P(Rain=false|Location=Sahara,Rain[t-1]=true)
    0.01,  // P(Rain=true |Location=Sahara,Rain[t-1]=false)
    0.99   // P(Rain=false|Location=Sahara,Rain[t-1]=false)
});
```

Finally, we adjust the number of slices created during the network unrolling with `DSL_network::SetNumberOfSlices`:

```
net.SetNumberOfSlices(5);
```

The network is complete. The program proceeds with a call to an inference algorithm, first without any evidence in the network, then with two observations of the *Umbrella*, set in the time steps *t=1* and *t=3*:

```
net.GetNode(umb)->Val()->SetTemporalEvidence(1, 0);
net.GetNode(umb)->Val()->SetTemporalEvidence(3, 1);
```

Note that `DSL_nodeVal::SetTemporalEvidence` does not have the overload that takes an outcome identifier (a string) as 2nd argument. We need to provide an integer outcome index. In this case, we know that *Umbrella* outcomes are *true* at index 0 and *false* at index 1, because we created this node at the start of this tutorial.

In dynamic Bayesian networks, the plate nodes have their beliefs calculated for each time slice. The number of elements in the node value matrix for these nodes is the product of outcome count and slice count. The helper function `UpdateAndShowTemporalResults` iterates over this matrix using two nested loops in order to print the results:

```
const DSL_Dmatrix *mtx = node->Val()->GetMatrix();
for (int sliceIdx = 0; sliceIdx < sliceCount; sliceIdx++)
{
    printf("\tt=%d:", sliceIdx);
    for (int i = 0; i < outcomeCount; i++)
    {
        printf(" %f", (*mtx)[sliceIdx * outcomeCount + i]);
    }
    printf("\n");
}
```

Because the probabilities for the same slice are adjacent, the inner loop uses the product of the outcome count and slice index from the outer loop as a base index.

The dynamic network is saved to a file for future reference. If you want to explore the structure of the unrolled dynamic model, load the tutorial6.xdsl file into GeNIe and use the 'Unroll' command. Please compare the unrolled network with a more complex dynamic network described in the *Unrolling* 45 section.

Tutorial 6 ends here.

7.7.1 tutorial6.cpp

```
// tutorial6.cpp
// Tutorial6 creates a dynamic Bayesian network (DBN),
// performs the inference, then saves the model to disk.

#include "smile.h"
#include <cstdio>


static int CreateCptNode(
    DSL_network& net, const char* id, const char* name,
    std::initializer_list<const char*> outcomes, int xPos, int yPos);

static void UpdateAndShowTemporalResults(DSL_network &net);
```

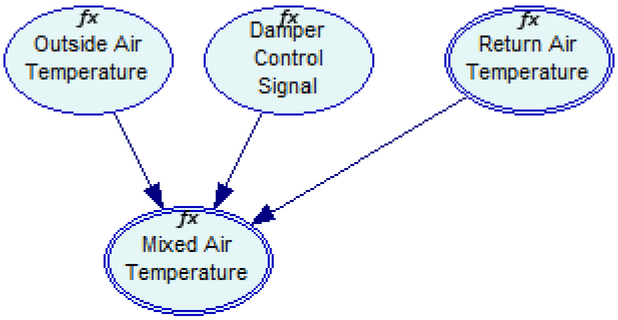
```
int Tutorial6()
{
    printf("Starting Tutorial6...\n");

    DSL_errorH().RedirectToFile(stdout);

    DSL_network net;

    int loc = CreateCptNode(net, "Location", "Location",
        { "Pittsburgh", "Sahara" }, 160, 360);

    int rain = CreateCptNode(net, "Rain", "Rain",
        { "true", "false" }, 380, 240);

    int umb = CreateCptNode(net, "Umbrella", "Umbrella",
        { "true", "false" }, 300, 100);

    net.SetTemporalType(rain, dsl_temporalType::dsl_plateNode);
    net.SetTemporalType(umb, dsl_temporalType::dsl_plateNode);

    net.AddArc(loc, rain);
    net.AddTemporalArc(rain, rain, 1);
    net.AddArc(rain, umb);

    DSL_nodeDef *rainDef = net.GetNode(rain)->Def();
    int res = rainDef->SetDefinition({
        0.7,  // P(Rain=true |Location=Pittsburgh)
        0.3,  // P(Rain=false|Location=Pittsburgh)
        0.01, // P(Rain=true |Location=Sahara)
        0.99  // P(Rain=false|Location=Sahara)
    });
    if (DSL_OKAY != res)
    {
        return res;
    }

    res = rainDef->SetTemporalDefinition(1, {
        0.7,   // P(Rain=true |Location=Pittsburgh,Rain[t-1]=true)
        0.3,   // P(Rain=false|Location=Pittsburgh,Rain[t-1]=true)
        0.3,   // P(Rain=true |Location=Pittsburgh,Rain[t-1]=false)
        0.7,   // P(Rain=false|Location=Pittsburgh,Rain[t-1]=false)
        0.001, // P(Rain=true |Location=Sahara,Rain[t-1]=true)
        0.999, // P(Rain=false|Location=Sahara,Rain[t-1]=true)
        0.01,  // P(Rain=true |Location=Sahara,Rain[t-1]=false)
        0.99   // P(Rain=false|Location=Sahara,Rain[t-1]=false)
    });
    if (DSL_OKAY != res)
    {
        return res;
    }

    res = net.GetNode(umb)->Def()->SetDefinition({
        0.9, // P(Umbrella=true |Rain=true)
        0.1, // P(Umbrella=false|Rain=true)
```

```
        0.2, // P(Umbrella=true |Rain=false)
        0.8  // P(Umbrella=false|Rain=false)
    });
    if (DSL_OKAY != res)
    {
        return res;
    }

    net.SetNumberOfSlices(5);

    printf("Performing update without evidence.\n");
    UpdateAndShowTemporalResults(net);

    printf("Setting Umbrella[t=1] to true and Umbrella[t=3] to false.\n");
    res = net.GetNode(umb)->Val()->SetTemporalEvidence(1, 0);
    if (DSL_OKAY != res)
    {
        return res;
    }
    res = net.GetNode(umb)->Val()->SetTemporalEvidence(3, 1);
    if (DSL_OKAY != res)
    {
        return res;
    }
    UpdateAndShowTemporalResults(net);

    res = net.WriteFile("tutorial6.xdsl");
    if (DSL_OKAY != res)
    {
        return res;
    }

    printf("Tutorial6 complete: Network written to tutorial6.xdsl\n");
    return DSL_OKAY;
}


static void UpdateAndShowTemporalResults(DSL_network &net)
{
    net.UpdateBeliefs();
    int sliceCount = net.GetNumberOfSlices();
    for (int h = net.GetFirstNode(); h >= 0; h = net.GetNextNode(h))
    {
        if (net.GetTemporalType(h) == dsl_temporalType::dsl_plateNode)
        {
            DSL_node *node = net.GetNode(h);
            int outcomeCount = node->Def()->GetNumberOfOutcomes();
            printf("Temporal beliefs for %s:\n", node->GetId());
            const DSL_Dmatrix& mtx = *node->Val()->GetMatrix();
            for (int sliceIdx = 0; sliceIdx < sliceCount; sliceIdx++)
            {
                printf("\tt=%d:", sliceIdx);
                for (int i = 0; i < outcomeCount; i++)
                {
                    printf(" %f", mtx[sliceIdx * outcomeCount + i]);
```

```
                }
                printf("\n");
            }
        }
    }
    printf("\n");
}


static int CreateCptNode(
    DSL_network& net, const char* id, const char* name,
    std::initializer_list<const char*> outcomes, int xPos, int yPos)
{
    int handle = net.AddNode(DSL_CPT, id);
    DSL_node* node = net.GetNode(handle);
    node->SetName(name);
    node->Def()->SetNumberOfOutcomes(outcomes);
    DSL_rectangle& position = node->Info().Screen().position;
    position.center_X = xPos;
    position.center_Y = yPos;
    position.width = 85;
    position.height = 55;
    return handle;
}
```

7.8 Tutorial 7: A continuous model

The continuous Bayesian network used in this tutorial focuses on a fragment of a forced air heating and cooling system. In order to improve the system's efficiency, return air is mixed with the air that is drawn from outside. Temperature of the outside air depends on the weather and is specified by means of a Normal distribution. Return air temperature is constant and depends on the thermostat setting. Damper control signal determines the composition of the mixture.



Temperature of the mixture is calculated according to the following equation: *tma=toa*u_d+(tra-tra*u_d)*, where *tma* is mixed air temperature, *toa* is outside air temperature, *u_d* is the damper signal, and *tra* is return air temperature. Other equation in the model are: *tra=24* for return air temperature (which is assumed to be constant), *u_d=Bernoulli(0.539)*0.8+0.2* for damper control and *toa=Normal(11,15)* for outside air temperature. Note the *fx* symbol above node captions used by GeNIe to indicate that the node is equation-based.

The nodes in this model are created by the helper function `CreateEquationNode`. It is a modified version of the `CreateCptNode` used in the previous tutorials. The bold text marks the difference between two functions.

```
static int CreateEquationNode(
    DSL_network &net, const char *id, const char *name,
    const char *equation, double loBound, double hiBound,
    int xPos, int yPos)
{
    int handle = net.AddNode(DSL_EQUATION, id);
    DSL_node *node = net.GetNode(handle);
    node->SetName(name);
    auto eq = node->Def<DSL_equation>();
    eq->SetEquation(equation);
    eq->SetBounds(loBound, hiBound);
    DSL_rectangle &position = node->Info().Screen().position;
    position.center_X = xPos;
    position.center_Y = yPos;
    position.width = 85;
    position.height = 55;
    return handle;
}
```

Instead of discrete outcomes, we specify the node's equation and bounds (domain of the node values). The equation is passed as string, which is parsed and validated. For simplicity, we do not check for the status code returned from `DSL_equation::SetEquation`. In production code, especially if the equation is specified by the user input, you should definitely add a check for `DSL_OKAY` status. Note that unlike earlier node definition methods, `DSL_equation::SetEquation` is defined in a class derived from `DSL_nodeDef`. To call this method, we need to cast the node definition pointer to the `DSL_equation` type. The easiest way is to use the templated version of the `DSL_node::Def` method, where the template argument specifies the definition type.

Another helper function, `SetUniformIntervals`, is used to define the node's discretization intervals. The intervals are used by the inference algorithm when the evidence is set for *Mixed Air Temperature* node (which has parents). Uniform intervals are chosen for simplicity here; in general case the choice of interval edges should be done based on the actual expected distribution of the node value (for example, in case of the Normal distribution, we might create narrow discretization intervals close to the mean.)

Note that while the model has three arcs, there are no calls to `DSL_network::AddArc` in this tutorial. The arcs are created implicitly by `DSL_equation::SetEquation` method (called by `CreateEquationNode` function).

The network is complete now and we can proceed with inference. The program makes three inference calls, one without evidence and two with continuous evidence specified by calling `DSL_nodeVal::SetEvidence(double)`. Setting the *Outside Air Temperature* to 28.5 degrees (`toa` is the name of `int` variable holding the handle of the *Outside Air Temperature* node):

```
net.GetNode(toa)->Val()->SetEvidence(28.5);
```

This overload of the `SetEvidence` method is easy to confuse with the one used in previous tutorials, that accepts an integer as a parameter. If the temperature to set had no fractional part, we would need to ensure the literal is of type `double` by appending ".0":

```
net.GetNode(tma)->Val()->SetEvidence(21.0);
```

The program uses `UpdateAndShowStats` helper function for inference. The helper calls `DSL_network::UpdateBeliefs` and iterates over the nodes in the network, calling another helper, `ShowStats`, for each node. `ShowStats` first checks if the node has evidence set. If it does, the evidence value is printed, and the function returns:

```
if (eqVal->IsEvidence())
{
    double v;
```

```
        eqVal->GetEvidence(v);
        printf("%s has evidence set (%g)\n", nodeId, v);
        return;
    }
```

If the node has no evidence, we need to check whether its value comes from the sampling or discretized inference. If sampling was used, the `std::vector` returned from `DSL_valEqEvaluation::GetDiscBeliefs` is empty:

```
    const std::vector<double> &discBeliefs = eqVal->GetDiscBeliefs();
    if (discBeliefs.empty())
    {
        double mean, stddev, vmin, vmax;
        eqVal->GetStats(mean, stddev, vmin, vmax);
        printf("%s: mean=%g stddev=%g min=%g max=%g\n",
            nodeId, mean, stddev, vmin, vmax);
    }
```

In such case, the output for the node contains simple statistics from sampling retrieved by `DSL_valEqEvaluation::GetStats`. To avoid excessive output, we omit the actual sample values, which could be retrieved by `DSL_valEqEvaluation::GetSample[s]`.

If the network had evidence set for the *Mixed Air Temperature* node (which has parents), the inference algorithm would fall back to discretization, and the discretized belief vector would be non-empty. The `else` part of the `if` statement looks as follows:

```
    auto eqDef = node->Def<DSL_equation>();
    const DSL_equation::IntervalVector &iv = eqDef->GetDiscIntervals();
    printf("%s is discretized.\n", nodeId);
    double loBound, hiBound;
    eqDef->GetBounds(loBound, hiBound);
    double lo = loBound;
    for (int i = 0; i < discBeliefs.GetSize(); i++)
    {
        double hi = iv[i].second;
        printf("\tP(%s in %g..%g)=%g\n", nodeId, lo, hi, discBeliefs[i]);
        lo = hi;
    }
```

Note how we need to read the discretization intervals specified in the equation node definition to display the complete information about the discretized beliefs (probability comes from node value and discretization interval edges from node definition).

All tutorials redirect SMILE's diagnostic stream to the console, and during the execution of this tutorial the following message will appear on the console when `UpdateAndShowStats` is called after setting *Mixed Air Temperature* to 21 degrees:

```
-19: Equation node u_d was discretized.
-4: Discretization problem in node toa:  Underflow samples: 807, min=-39.8255 loBound=-10
Overflow samples: 275, max=72.2534 hiBound=40  Total valid samples: 8918 of 10000
-4: Discretization problem in node tma:  Underflow samples: 15145, min=-9.65187 loBound=10
Overflow samples: 8203, max=39.9188 hiBound=30  Total valid samples: 76652 of 100000
```

The first line with status code -19 (`DSL_WRONG_NUM_STATES`) is a warning about missing discretization intervals for the *Damper Control Signal* node. In such case, SMILE uses two intervals dividing the node's domain into two equal halves. Two intervals are adequate in this case, as the equation for this node uses Bernoulli distribution. Status code -4 (`DSL_INVALID_VALUE`) is a warning that some of the discretization samples fall outside of the node's domain (defined by the lower and the upper bounds set earlier by `DSL_equation::SetBounds`). Detailed information about underflow and overflow can help during the model building and verification. However, with all unbounded probability distributions and domain bounds determined by physical properties of the modeled system, some of the generated samples will inevitably fall out of bounds.

At the end of the tutorial, the model is saved to disk. *Tutorial 8* 100 will expand it into a hybrid network by adding CPT nodes.

7.8.1 tutorial7.cpp

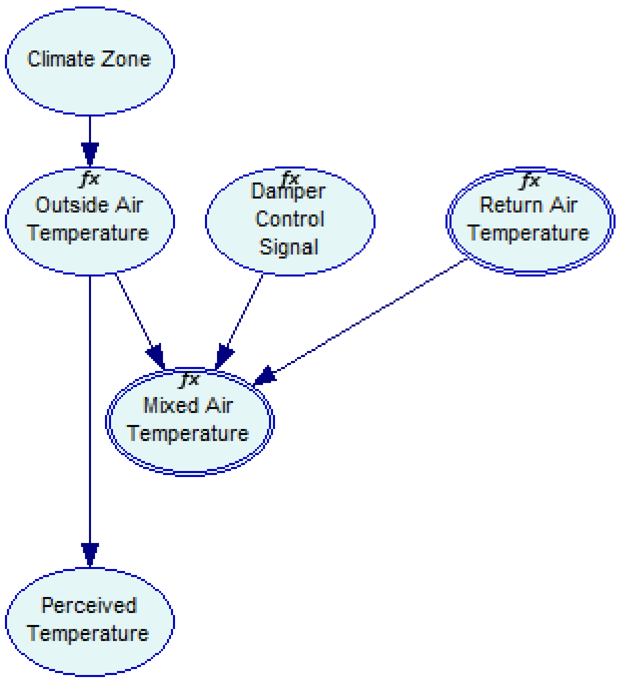
```
// tutorial7.cpp
// Tutorial7 creates a network with three equation-based nodes
// performs the inference, then saves the model to disk.

#include "smile.h"
#include <cstdio>


static int CreateEquationNode(
    DSL_network &net, const char *id, const char *name,
    const char *equation, double loBound, double hiBound,
    int xPos, int yPos);

static void UpdateAndShowStats(DSL_network &net);

static void SetUniformIntervals(DSL_network &net, int nodeHandle, int count);

int Tutorial7()
{
    printf("Starting Tutorial7...\n");

    DSL_errorH().RedirectToFile(stdout);

    DSL_network net;
    net.EnableRejectOutlierSamples(true);
    int tra = CreateEquationNode(net,
        "tra", "Return Air Temperature",
        "tra=24", 23.9, 24.1,
        280, 100);
    int u_d = CreateEquationNode(net,
        "u_d", "Damper Control Signal",
        "u_d = Bernoulli(0.539)*0.8 + 0.2", 0, 1,
        160, 100);
    int toa = CreateEquationNode(net,
        "toa", "Outside Air Temperature",
        "toa=Normal(11,15)", -10, 40,
        60, 100);
    int tma = CreateEquationNode(net,
        "tma", "Mixed Air Temperature",
        "tma=toa*u_d+(tra-tra*u_d)", 10, 30,
        110, 200);

    SetUniformIntervals(net, toa, 5);
```

```
    SetUniformIntervals(net, tma, 4);

    printf("Results with no evidence:\n");
    UpdateAndShowStats(net);

    net.GetNode(toa)->Val()->SetEvidence(28.5);
    printf("Results with outside air temperature set to 28.5:\n");
    UpdateAndShowStats(net);

    net.GetNode(toa)->Val()->ClearEvidence();
    printf("Results with mixed air temperature set to 21:\n");
    net.GetNode(tma)->Val()->SetEvidence(21.0); // ensure it's a double value
    UpdateAndShowStats(net);

    int res = net.WriteFile("tutorial7.xdsl");
    if (DSL_OKAY != res)
    {
        return res;
    }

    printf("Tutorial7 complete: Network written to tutorial7.xdsl\n");
    return DSL_OKAY;
}


static int CreateEquationNode(
      DSL_network &net, const char *id, const char *name,
    const char *equation, double loBound, double hiBound,
    int xPos, int yPos)
{
    int handle = net.AddNode(DSL_EQUATION, id);
    DSL_node *node = net.GetNode(handle);
    node->SetName(name);
    auto eq = node->Def<DSL_equation>();
    eq->SetEquation(equation);
    eq->SetBounds(loBound, hiBound);
    DSL_rectangle &position = node->Info().Screen().position;
    position.center_X = xPos;
    position.center_Y = yPos;
    position.width = 85;
    position.height = 55;
    return handle;
}


static void ShowStats(DSL_network &net, int nodeHandle)
{
    DSL_node* node = net.GetNode(nodeHandle);
    const char *nodeId = node->GetId();

    auto eqVal = node->Val<DSL_equationEvaluation>();
    if (eqVal->IsEvidence())
    {
        double v;
        eqVal->GetEvidence(v);
```

```
        printf("%s has evidence set (%g)\n", nodeId, v);
        return;
    }

    const DSL_Dmatrix &discBeliefs = eqVal->GetDiscBeliefs();
    if (discBeliefs.IsEmpty())
    {
        double mean, stddev, vmin, vmax;
        eqVal->GetStats(mean, stddev, vmin, vmax);
        printf("%s: mean=%g stddev=%g min=%g max=%g\n",
            nodeId, mean, stddev, vmin, vmax);
    }
    else
    {
        auto eqDef = node->Def<DSL_equation>();
        const DSL_equation::IntervalVector &iv = eqDef->GetDiscIntervals();
        printf("%s is discretized.\n", nodeId);
        double loBound, hiBound;
        eqDef->GetBounds(loBound, hiBound);
        double lo = loBound;
        for (int i = 0; i < discBeliefs.GetSize(); i++)
        {
            double hi = iv[i].second;
            printf("\tP(%s in %g..%g)=%g\n", nodeId, lo, hi, discBeliefs[i]);
            lo = hi;
        }
    }
}


static void UpdateAndShowStats(DSL_network &net)
{
    net.UpdateBeliefs();
    for (int h = net.GetFirstNode(); h >= 0; h = net.GetNextNode(h))
    {
        ShowStats(net, h);
    }
    printf("\n");
}


static void SetUniformIntervals(DSL_network &net, int nodeHandle, int count)
{
    auto eq = net.GetNode(nodeHandle)->Def<DSL_equation>();
    double lo, hi;
    eq->GetBounds(lo, hi);
    DSL_equation::IntervalVector iv(count);
    for (int i = 0; i < count; i++)
    {
        iv[i].second = lo + (i + 1) * (hi - lo) / count;
    }
    eq->SetDiscIntervals(iv);
}
```

7.9 Tutorial 8: Hybrid model

We extend the model described in *Tutorial 7* [94] by adding two discrete nodes: *Climate Zone* and *Perceived Temperature*. *Climate Zone* refines the probability distribution over the outside air temperature and *Perceived Temperature* is an additional input originating from a subjective perception of the temperature, useful in case of a failure in the outside temperature sensor.



After loading the network created in the previous tutorial, the program adds two discrete nodes using the `CreateCptNode` helper function first seen in *Tutorial 1* [71]. The arc from *Climate Zone* (node identifier is *zone*) to *Outside Air Temperature* (node identifier is *toa*) is created by changing the equation of the latter:

```
auto eq = net.GetNode(toa)->Def<DSL_equation>();
eq->SetEquation("toa=If(zone=\"Desert\",Normal(22,5),Normal(11,10))");
```

In the C++ code, we need to escape the double quote characters in order to use the text literal representing the *Desert* outcome of the parent. The new equation switches between two normal distribution based on the outcome of the parent. The equation could be also written in the following ways, all being functionally identical (assuming that *Climate Zone* has two outcomes, *Temperate* and *Desert*):

```
eq->SetEquation("toa=zone=\"Desert\" ? Normal(22,5) : Normal(11,10)");
eq->SetEquation("toa=Switch(zone, \"Desert\",Normal(22,5),\"Temperate\",Normal(11,10))");
eq->SetEquation("toa=Choose(zone,Normal(11,10),Normal(22,5))");
```

To create an arc from *Outside Air Temperature* to *Perceived Temperature*, the program calls `DSL_network::AddArc` (because the child node is a CPT).

The model loaded from disk had 5 discretization intervals already defined for *Outside Air Temperature*. With 5 possible discretized outcomes of its single parent and its own 3 outcomes, the CPT for *Perceived Temperature* has 3*5=15 entries. It is initialized with `DSL_nodeDef::SetDefinition`. This is exactly the same approach that we used for discrete nodes in the previous tutorials. Note that for the sake of simplicity we do not change the default uniform distribution for the binary *Climate Zone* node.

The program performs inference for each of the *Climate Zone* outcomes set as evidence. The output displayed for the *Outside Air Temperature* node (*toa*) shows changing mean and standard deviation.

Finally, the network is saved to disk. This concludes the tutorial.

7.9.1 tutorial8.cpp

```
// tutorial8.cpp
// Tutorial8 loads continuous model from the XDSL file written by Tutorial7,
// then adds discrete nodes to create a hybrid model. Inference is performed
// and model is saved to disk.

#include "smile.h"
#include <cstdio>

static int CreateCptNode(
    DSL_network& net, const char* id, const char* name,
    std::initializer_list<const char*> outcomes, int xPos, int yPos);

static void UpdateAndShowStats(DSL_network &net);


int Tutorial8()
{
    printf("Starting Tutorial8...\n");

    DSL_errorH().RedirectToFile(stdout);

    DSL_network net;
    int res = net.ReadFile("tutorial7.xdsl");
    if (DSL_OKAY != res)
    {
        printf(
            "Network load failed, did you run Tutorial7 before Tutorial8?\n");
        return res;
    }

    int toa = net.FindNode("toa");
    if (toa < 0)
    {
        printf("Outside air temperature node not found.\n");
        return toa;
    }

    CreateCptNode(net,
        "zone", "Climate Zone", { "Temperate", "Desert" },
        60, 20);

    auto eq = net.GetNode(toa)->Def<DSL_equation>();
    eq->SetEquation("toa=If(zone=\"Desert\",Normal(22,5),Normal(11,10))");
```

```
    int perceived = CreateCptNode(net,
        "perceived", "Perceived Temperature", { "Hot", "Warm", "Cold" },
        60, 300);
    net.AddArc(toa, perceived);

    res = net.GetNode(perceived)->Def()->SetDefinition({
        0   , // P(perceived=Hot |toa in -10..0)
        0.02, // P(perceived=Warm|toa in -10..0)
        0.98, // P(perceived=Cold|toa in -10..0)
        0.05, // P(perceived=Hot |toa in 0..10)
        0.15, // P(perceived=Warm|toa in 0..10)
        0.80, // P(perceived=Cold|toa in 0..10)
        0.10, // P(perceived=Hot |toa in 10..20)
        0.80, // P(perceived=Warm|toa in 10..20)
        0.10, // P(perceived=Cold|toa in 10..20)
        0.80, // P(perceived=Hot |toa in 20..30)
        0.15, // P(perceived=Warm|toa in 20..30)
        0.05, // P(perceived=Cold|toa in 20..30)
        0.98, // P(perceived=Hot |toa in 30..40)
        0.02, // P(perceived=Warm|toa in 30..40)
        0     // P(perceived=Cold|toa in 30..40)
    });
    if (DSL_OKAY != res)
    {
        return res;
    }

    net.GetNode("zone")->Val()->SetEvidence("Temperate");
    printf("Results in temperate zone:\n");
    UpdateAndShowStats(net);

    net.GetNode("zone")->Val()->SetEvidence("Desert");
    printf("Results in desert zone:\n");
    UpdateAndShowStats(net);

    res = net.WriteFile("tutorial8.xdsl");
    if (DSL_OKAY != res)
    {
        return res;
    }

    printf("Tutorial8 complete: Network written to tutorial8.xdsl\n");
    return DSL_OKAY;
}


static void ShowStats(DSL_network& net, int nodeHandle)
{
    DSL_node* node = net.GetNode(nodeHandle);
    const char* nodeId = node->GetId();

    auto eqVal = node->Val<DSL_equationEvaluation>();
    if (eqVal->IsEvidence())
    {
        double v;
```

```
        eqVal->GetEvidence(v);
        printf("%s has evidence set (%g)\n", nodeId, v);
        return;
    }

    const DSL_Dmatrix& discBeliefs = eqVal->GetDiscBeliefs();
    if (discBeliefs.IsEmpty())
    {
        double mean, stddev, vmin, vmax;
        eqVal->GetStats(mean, stddev, vmin, vmax);
        printf("%s: mean=%g stddev=%g min=%g max=%g\n",
            nodeId, mean, stddev, vmin, vmax);
    }
    else
    {
        auto eqDef = node->Def<DSL_equation>();
        const DSL_equation::IntervalVector& iv = eqDef->GetDiscIntervals();
        printf("%s is discretized.\n", nodeId);
        double loBound, hiBound;
        eqDef->GetBounds(loBound, hiBound);
        double lo = loBound;
        for (int i = 0; i < discBeliefs.GetSize(); i++)
        {
            double hi = iv[i].second;
            printf("\tP(%s in %g..%g)=%g\n", nodeId, lo, hi, discBeliefs[i]);
            lo = hi;
        }
    }
}


static void UpdateAndShowStats(DSL_network &net)
{
    net.UpdateBeliefs();
    for (int h = net.GetFirstNode(); h >= 0; h = net.GetNextNode(h))
    {
        if (net.GetNode(h)->Def()->GetType() == DSL_EQUATION)
        {
           ShowStats(net, h);
        }
    }
}


static int CreateCptNode(
    DSL_network& net, const char* id, const char* name,
    std::initializer_list<const char*> outcomes, int xPos, int yPos)
{
    int handle = net.AddNode(DSL_CPT, id);
    DSL_node* node = net.GetNode(handle);
    node->SetName(name);
    node->Def()->SetNumberOfOutcomes(outcomes);
    DSL_rectangle& position = node->Info().Screen().position;
    position.center_X = xPos;
    position.center_Y = yPos;
```

```
    position.width = 85;
    position.height = 55;
    return handle;
}
```

7.10 Tutorial 9: Structure learning

Suppose we have a file *Credit10K.csv* consisting of 10,000 records of customers collected at a bank. Each of these customers was measured on several variables, *Payment History*, *Work History*, *Reliability*, *Debit*, *Income*, *Ratio of Debts to Income*, *Assets*, *Worth*, *Profession*, *Future Income*, *Age* and *Credit Worthiness*. The file is too large to include in this manual, but it can be downloaded from https://support.bayesfusion.com/docs/Examples/Learning/Credit10K.csv. It is also included in the *Examples\Learning* directory of GeNIe installation. The first few records of the file (included among the example files with GeNIe distribution) look as follows in GeNIe:

PaymentHistory	WorkHistory	Reliability	Debit	Income	RatioDebInc	Assets	Worth	Profession	FutureIncome	Age	CreditWorthiness
Without_Reference	Unstable	Unreliable	a0_11100	s70001_more	Favorable	wealthy	High	Medium_income_profession	Promissing	a16_21	Negative
Aceptable	Unjustified_no_work	Unreliable	a0_11100	s70001_more	Favorable	average	High	Medium_income_profession	Promissing	a66_up	Negative
Aceptable	Unstable	Reliable	a25901_more	s30001_70000	Unfavorable	wealthy	High	Low_income_profession	Not_promissing	a16_21	Negative
Excellent	Unstable	Reliable	a25901_more	s30001_70000	Unfavorable	average	Medium	Medium_income_profession	Not_promissing	a16_21	Negative
Excellent	Unjustified_no_work	Unreliable	a11101_25900	s0_30000	Unfavorable	average	Low	Medium_income_profession	Not_promissing	a66_up	Negative
Without_Reference	Stable	Reliable	a0_11100	s30001_70000	Favorable	average	High	Medium_income_profession	Promissing	a16_21	Positive
NoAceptable	Stable	Unreliable	a0_11100	s70001_more	Favorable	wealthy	High	Medium_income_profession	Promissing	a66_up	Positive
Excellent	Stable	Reliable	a0_11100	s70001_more	Favorable	wealthy	High	Low_income_profession	Promissing	a66_up	Positive
Excellent	Stable	Reliable	a25901_more	s70001_more	Unfavorable	poor	High	Low_income_profession	Not_promissing	a16_21	Negative
NoAceptable	Stable	Unreliable	a0_11100	s30001_70000	Favorable	average	Medium	Medium_income_profession	Promissing	a22_65	Positive
Without_Reference	Justified_no_work	Reliable	a25901_more	s70001_more	Unfavorable	poor	High	Low_income_profession	Not_promissing	a16_21	Negative
NoAceptable	Unstable	Unreliable	a25901_more	s30001_70000	Unfavorable	wealthy	High	Medium_income_profession	Promissing	a16_21	Negative
NoAceptable	Justified_no_work	Unreliable	a25901_more	s30001_70000	Unfavorable	wealthy	High	High_income_profession	Promissing	a22_65	Negative
Excellent	Stable	Reliable	a11101_25900	s0_30000	Unfavorable	average	Low	Medium_income_profession	Not_promissing	a16_21	Negative
Aceptable	Stable	Unreliable	a25901_more	s0_30000	Unfavorable	wealthy	Medium	Medium_income_profession	Not_promissing	a66_up	Negative
Without_Reference	Unjustified_no_work	Unreliable	a0_11100	s0_30000	Favorable	poor	Low	Low_income_profession	Not_promissing	a66_up	Positive
Aceptable	Unstable	Reliable	a11101_25900	s30001_70000	Unfavorable	average	Medium	Low_income_profession	Not_promissing	a66_up	Negative
Without_Reference	Unstable	Unreliable	a0_11100	s30001_70000	Unfavorable	average	High	Medium_income_profession	Promissing	a16_21	Negative

We want to learn the structure of the Bayesian network based on the data. Our program attempts to load the data file from disk, and prints the number of data set variables and records if loading was successful.

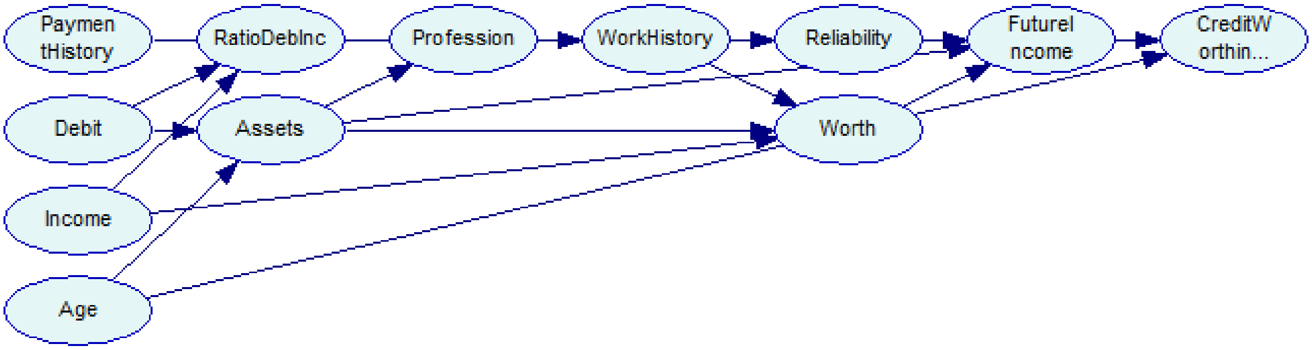
```
DSL_dataset ds;
int res = ds.ReadFile("Credit10k.csv");
if (DSL_OKAY != res)
{
    printf("Dataset load failed\n");
    return res;
}
printf("Dataset has %d variables (columns) and %d records (rows)\n",
```

The most general structure learning algorithm is Bayesian Search. It is a hill climbing procedure with random restarts, guided by log-likelihood scoring heuristic. Its search space is hyper-exponential, so for best results the algorithm can be fine-tuned with multiple options (details are available in the *Reference* [190] section). We set the number of iterations (random restarts) to 50, and fix the random number seed to ensure reproducible results. The `DSL_bs::Learn` method requires at least two parameters: (1) an input data set, and (2) an output network. We also want to record the log-likelihood of best structure among the 50 best structures created internally. The best structure will be copied into the output network, and parameter learning will be run internally. The output network will therefore be ready to use.

```
double bestScore;
DSL_bs bayesSearch;
bayesSearch.nrIteration = 50;
DSL_network net1;
bayesSearch.seed = 9876543;
```

```
res = bayesSearch.Learn(ds, net1, NULL, NULL, &bestScore);
if (DSL_OKAY != res)
{
    printf("Bayesian Search failed (%d)\n", res);
    return res;
}
net1.SimpleGraphLayout();
printf("1st Bayesian Search finished, structure score: %g\n", bestScore);
net1.WriteFile("tutorial9-bs1.xdsl");
```

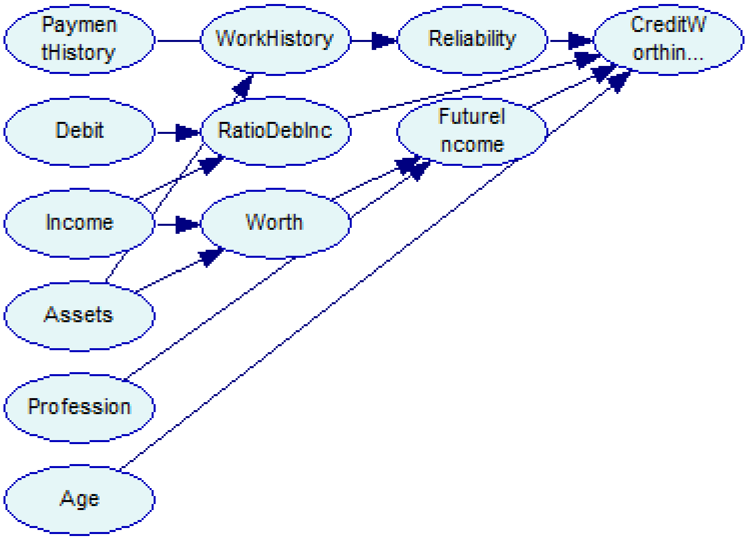
On success, we call `DSL_network::SimpleGraphLayout` to perform the parent ordering layout. Without this step, all nodes would occupy the same position and the structure would be hard to read when opened in GeNIe. The network looks as follows in GeNIe:



The network is saved for reference in the *tutorial9-bs1.xdsl* file. We can obtain a different structure by changing the seed for the random number generator and calling `DSL_bs::Learn` again.

```
DSL_network net2;
bayesSearch.seed = 3456789;
res = bayesSearch.Learn(ds, net2, NULL, NULL, &bestScore);
```

The second network, saved in *tutorial9-bs2.xdsl*, looks like this:



The log-likelihood scores for these two networks are -105473 and -105353, respectively.

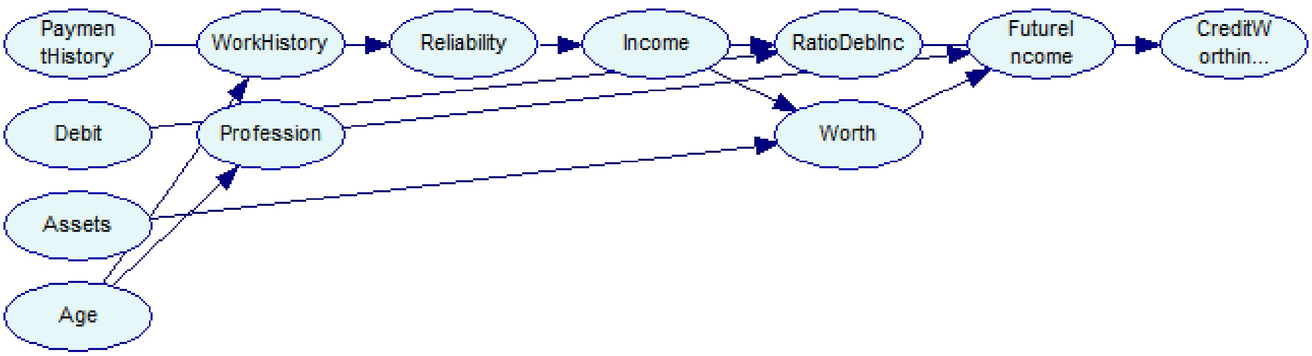
Our third attempt at structure learning with Bayesian Search will use background knowledge. We want to forbid the arc between *Age* and *CreditWorthiness*, and force an arc between *Age* and *Profession*. When specifying the background knowledge, we refer to the data set variables as their indices, so we need to convert the textual identifiers to indices first with `DSL_dataset::FindVariable`:

```
int idxAge = ds.FindVariable("Age");
int idxProfession = ds.FindVariable("Profession");
int idxCreditWorthiness = ds.FindVariable("CreditWorthiness");
```

The forbidden and forced arcs now can be specified before `DSL_bs::Learn` call.

```
bayesSearch.bkk.forbiddenArcs.push_back(make_pair(idxAge, idxCreditWorthiness));
bayesSearch.bkk.forcedArcs.push_back(make_pair(idxAge, idxProfession));
res = bayesSearch.Learn(ds, net3, NULL, NULL, &bestScore);
```

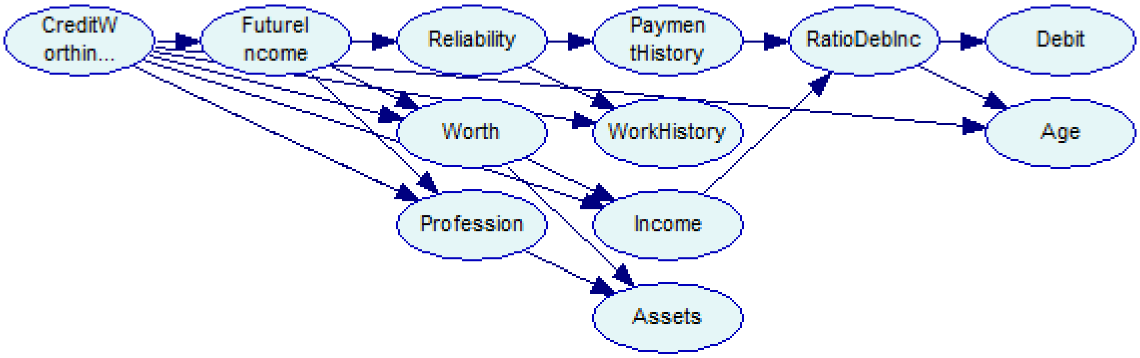
The output network now looks as follows - please note the absence of an arc from *Age* to *CreditWorthiness*, and the forced arc between *Age* and *Profession*.



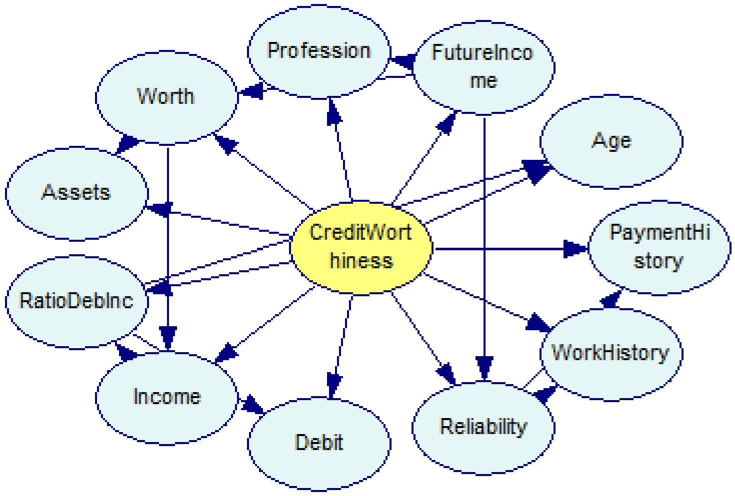
We now change our learning algorithm to Tree Augmented Naive Bayes (TAN), implemented in SMILE in `DSL_tan` class. The TAN algorithm starts with a Naive Bayes structure (i.e., one in which the class variable is the only parent of all remaining, feature variables) and adds connections between the feature variables to account for possible dependence between them, conditional on the class variable. The algorithm imposes the limit of only one additional parent of every feature variable (additional to the class variable, which is a parent of every feature variable). We need to specify the class variable with its textual identifier. In this example *CreditWorthiness* is the class variable.

```
DSL_network net4;
DSL_tan tan;
tan.seed = 777999;
tan.classvar = "CreditWorthiness";
res = tan.Learn(ds, net4);
```

The program saves the network for reference in *tutorial9-tan.xdsl* file. The network looks as follows:



The layout of the TAN output network in our tutorial program does not take the specifics of the TAN into account. The same network looks like this when more sophisticated layout algorithm in GeNIe is used. The distinction between class and feature variables is now clearer.



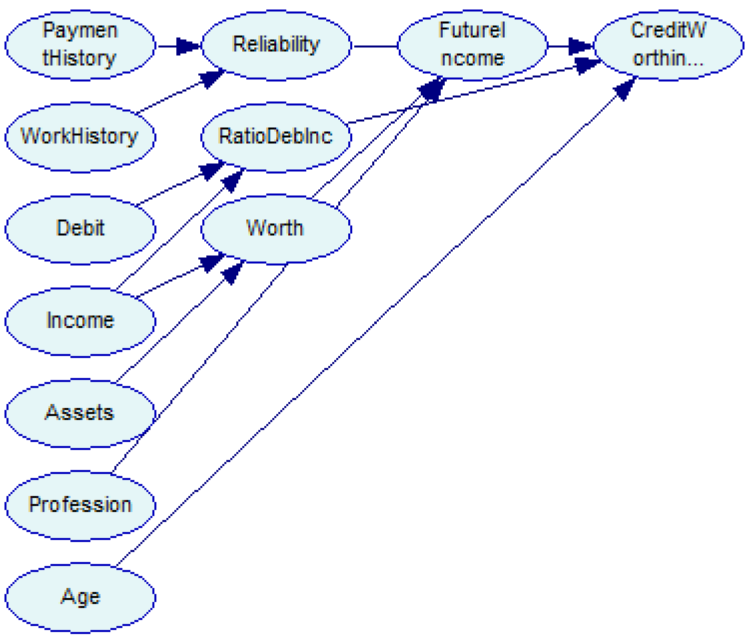
Our final structure learning algorithm will be PC, which uses independences observed in data (established by means of classical independence tests) to infer the structure that has generated them. SMILE class implementing the PC algorithm is `DSL_pc`. The output of the PC algorithm is a pattern (represented by the `DSL_pattern` class). The pattern is an adjacency matrix, which does not necessarily represent a directed acyclic graph (DAG). The pattern can be converted to `DSL_network` with `DSL_pattern::ToNetwork`, which enforces the DAG criterion on the pattern, and copies the variables and edges to the network.

```
DSL_pc pc;
DSL_pattern pattern;
res = pc.Learn(ds, pattern);
if (DSL_OKAY != res)
{
    printf("PC failed (%d)\n", res);
    return res;
}
DSL_network net5;
pattern.ToNetwork(ds, net5);
```

At this point our output network has nodes with outcomes based on the labels in the data set, but the probability distributions in node definitions are uniform over outcomes. To complete the learning process, we need to manually call a parameter learning procedure implemented in `DSL_em` class.

```
DSL_em em;
string errMsg;
vector<DSL_datasetMatch> matching;
res = ds.MatchNetwork(net5, matching, errMsg);
if (DSL_OKAY != res)
{
    printf("Can't automatically match network with dataset: %s\n", errMsg.c_str());
    return DSL_OUT_OF_RANGE;
}
em.SetUniformizeParameters(false);
em.SetRandomizeParameters(false);
em.SetEquivalentSampleSize(0);
res = em.Learn(ds, net5, matching);
```

Note the `DSL_dataset::MatchNetwork` call, which populates the `matching` variable using identifiers of data set variables and network nodes. This information is subsequently used by `DSL_em::Learn` to associate data set variables with network node. The output network looks like as follows:



The network is saved for reference in the *tutorial9-pc-em.xdsl* file and Tutorial 9 finishes.

7.10.1 tutorial9.cpp

```
// tutorial9.cpp
// Tutorial9 loads Credit10k.csv file
// and runs multiple structure learning algorithms
// using the loaded dataset.
// Use the link below to download the Credit10k.csv file:
// https://support.bayesfusion.com/docs/Examples/Learning/Credit10K.csv
```

```
#include "smile.h"
#include <cstdio>
#include <utility>

using namespace std;

int Tutorial9()
{
    printf("Starting Tutorial9...\n");
    DSL_errorH().RedirectToFile(stdout);

    DSL_dataset ds;
    int res = ds.ReadFile("Credit10k.csv");
    if (DSL_OKAY != res)
    {
        printf("Dataset load failed\n");
        return res;
    }

    printf("Dataset has %d variables (columns) and %d records (rows)\n",
        ds.GetNumberOfVariables(), ds.GetNumberOfRecords());

    double bestScore;
    DSL_bs bayesSearch;
    bayesSearch.nrIteration = 50;

    DSL_network net1;
    bayesSearch.seed = 9876543;
    res = bayesSearch.Learn(ds, net1, NULL, NULL, &bestScore);
    if (DSL_OKAY != res)
    {
        printf("Bayesian Search failed (%d)\n", res);
        return res;
    }
    net1.SimpleGraphLayout();
    printf("1st Bayesian Search finished, structure score: %g\n", bestScore);
    net1.WriteFile("tutorial9-bs1.xdsl");

    DSL_network net2;
    bayesSearch.seed = 3456789;
    res = bayesSearch.Learn(ds, net2, NULL, NULL, &bestScore);
    if (DSL_OKAY != res)
    {
        printf("Bayesian Search failed (%d)\n", res);
        return res;
    }
    net2.SimpleGraphLayout();
    printf("2nd Bayesian Search finished, structure score: %g\n", bestScore);
    net2.WriteFile("tutorial9-bs2.xdsl");


    int idxAge = ds.FindVariable("Age");
    int idxProfession = ds.FindVariable("Profession");
    int idxCreditWorthiness = ds.FindVariable("CreditWorthiness");
    if (idxAge < 0 || idxProfession < 0 || idxCreditWorthiness < 0)
```

```
{
    printf("Can't find dataset variables for background knowledge\n");
    printf("The loaded file may not be Credit10k.csv\n");
    return DSL_OUT_OF_RANGE;
}
DSL_network net3;
bayesSearch.bkk.forbiddenArcs.push_back(make_pair(idxAge, idxCreditWorthiness));
bayesSearch.bkk.forcedArcs.push_back(make_pair(idxAge, idxProfession));
res = bayesSearch.Learn(ds, net3, NULL, NULL, &bestScore);
if (DSL_OKAY != res)
{
    printf("Bayesian Search finished (%d)\n", res);
    return res;
}
net3.SimpleGraphLayout();
printf("3rd Bayesian Search complete, structure score: %g\n", bestScore);
net3.WriteFile("tutorial9-bs3.xdsl");

DSL_network net4;
DSL_tan tan;
tan.seed = 777999;
tan.classvar = "CreditWorthiness";
res = tan.Learn(ds, net4);
if (DSL_OKAY != res)
{
    printf("TAN failed (%d)\n", res);
    return res;
}
net4.SimpleGraphLayout();
printf("Tree-augmented Naive Bayes finished\n");
net4.WriteFile("tutorial9-tan.xdsl");

DSL_pc pc;
DSL_pattern pattern;
res = pc.Learn(ds, pattern);
if (DSL_OKAY != res)
{
    printf("PC failed (%d)\n", res);
    return res;
}

DSL_network net5;
pattern.ToNetwork(ds, net5);
net5.SimpleGraphLayout();
printf("PC finished, proceeding to parameter learning\n");
net5.WriteFile("tutorial9-pc.xdsl");
DSL_em em;
string errMsg;
vector<DSL_datasetMatch> matching;
res = ds.MatchNetwork(net5, matching, errMsg);
if (DSL_OKAY != res)
{
    printf("Can't automatically match network with dataset: %s\n", errMsg.c_str());
    return DSL_OUT_OF_RANGE;
}
```

```
    em.SetUniformizeParameters(false);
    em.SetRandomizeParameters(false);
    em.SetEquivalentSampleSize(0);
    res = em.Learn(ds, net5, matching);
    if (DSL_OKAY != res)
    {
        printf("EM failed (%d)\n", res);
        return res;
    }
    printf("EM finished\n");
    net5.WriteFile("tutorial9-pc-em.xdsl");

      printf("Tutorial9 complete\n");
      return DSL_OKAY;
}
```

This page is intentionally left blank.

Reference Manual

8 Reference Manual

The information provided in this section covers the most important parts of SMILE's public API. For additional information, please refer to the header files included in your SMILE distribution - each class reference in this section begins with the header file name. The public members of SMILE classes are generally placed first within the class declarations.

Note that you do not need to include the headers one-by-one in your program; the main header `smile.h` does that.

8.1 Node types

The table below describes all supported node types. The identifier listed in the first column is used when creating a node with `DSL_network::AddNode`, or when changing the node type with `DSL_node::ChangeType`. It is also returned from the overridden virtual method `DSL_nodeDef::GetType` (`DSL_cpt::GetType` returns `DSL_CPT`, `DSL_truthTable::GetType` returns `DSL_TRUTHTABLE`, etc.). Each type is associated with a specific definition class. Note that the node value class `DSL_beliefVector` is associated with multiple node types.

Identifier	Description	Node definition class	Node value class
`DSL_CPT`	Conditional probability table	`DSL_cpt`	`DSL_beliefVector`
`DSL_TRUTHTABLE`	Discrete deterministic	`DSL_truthTable`	`DSL_beliefVector`
`DSL_NOISY_MAX`	Noisy-MAX (canonical)	`DSL_noisyMAX`	`DSL_beliefVector`
`DSL_NOISY_ADDER`	Noisy-Adder (canonical)	`DSL_noisyAdder`	`DSL_beliefVector`
`DSL_DEMORGAN`	DeMorgan (qualitative)	`DSL_demorgan`	`DSL_beliefVector`
`DSL_LIST`	Decision	`DSL_decision`	`DSL_policyValues`
`DSL_TABLE`	Utility	`DSL_utility`	`DSL_expectedUtility`
`DSL_MAU`	Multi-attribute utility	`DSL_mau`	`DSL_mauExpectedUtility`
`DSL_EQUATION`	Equation	`DSL_equation`	`DSL_equationEvaluation`

8.2 Error codes

Identifier	Numeric value
DSL_OKAY	0
DSL_GENERAL_ERROR	-1
DSL_OUT_OF_RANGE	-2
DSL_NO_ITEM	-3
DSL_INVALID_VALUE	-4
DSL_NO_USEFUL_SAMPLES	-5
DSL_CANT_SOLVE_EQUATION	-6
DSL_CYCLE_DETECTED	-11
DSL_WRONG_NUM_STATES	-19
DSL_CONFLICTING_EVIDENCE	-26
DSL_ILLEGAL_ID	-30
DSL_DUPLICATED_ID	-32
DSL_OUT_OF_MEMORY	-42
DSL_ZERO_POTENTIAL	-43
DSL_WRONG_NODE_TYPE	-51
DSL_WRONG_ELEMENT_TYPE	-52
DSL_INTERRUPTED	-99
DSL_FILE_READ	-100
DSL_FILE_WRITE	-101
DSL_END_OF_FILE	-102

Identifier	Numeric value
DSL_WRONG_FILE	-103
DSL_NO_MORE_TOKENS	-111
DSL_LEXICAL_ERROR	-126
DSL_SYNTAX_ERROR	-127
DSL_UNEXPECTED_EOF	-128
DSL_FIELD_NOT_FOUND	-129

8.3 Arrays and matrices

8.3.1 DSL_stringArray

Header file: **stringarray.h**

```
DSL_stringArray();
DSL_stringArray(int arraySize);
DSL_stringArray(const DSL_stringArray& other)
DSL_stringArray(std::initializer_list<const char*> initList)
~DSL_stringArray();
DSL_stringArray& operator=(const DSL_stringArray& other);
```

The default constructor, copy constructors, operator= and destructor are implemented. The methods accepting `std::initializer_list` as input are available only if C++11 standard is enabled during compliation. The constructor accepting an integer array size initializes all elements to be empty strings.

```
const char* const* begin() const;
const char* const* end() const;
```

The methods to enable C++ 11 range-based for loops. Can also be used with any Standard Library algorithm accepting const random access iterators.

```
const char* operator[](int index) const;
const char* Subscript(int index) const;
```

Indexed read access. The debug build of the library checks the index and asserts if the index is invalid.

```
virtual int SetString(int index, const char *value) const;
```

Indexed write access. Returns `DSL_OKAY`, or a `DSL_OUT_OF_RANGE` (a negative number) when `index` is invalid.

`int FindPosition(const char* value) const;`

Uses `strcmp` to find the index of the specified value. If the value is not found, returns `DSL_OUT_OF_RANGE` (a negative number).

`bool Contains(const char* value) const;`

Returns `true` if the specified value is present in the network.

`bool IsEmpty() const`

Returns `true` if the array is empty.

`int GetSize() const;`

Returns the number of elements in the array.

`int Reserve(int newReserved);`

Reserves the buffer capacity, use in performance-critical scenarios. Does not change the size of the array returned from `GetSize`.

`void Clear();`

Sets the array size to zero.

`virtual int Add(const char *value);`

Adds new element at the end of the array.

`virtual int Insert(int index, const char *value);`

Inserts new element in the array at the specified index. Returns `DSL_OKAY`, or `DSL_OUT_OF_RANGE` if the index is negative or greater than array size. `Insert` can be called with `index` equal to the array size. In such case, `Add` is invoked.

`int Delete(int index);`

Removes the element in the array at the specified index. The elements after the removed element are shifted toward the beginning of the array. Returns `DSL_OKAY`, or `DSL_OUT_OF_RANGE` if the index is negative or greater than array size.

```
int DeleteByContent(const char *value);
```

Uses `FindPosition` to find the index of specified `value`, and if found, removes the value with a call to `Delete`. If the value is not present in the network, returns `DSL_OUT_OF_RANGE`.

```
int ChangeOrder(const DSL_intArray& permutation);
```

Changes the order of elements in the array using the specified `permutation`. Returns `DSL_OKAY` on success, or a negative error code if the specified permutation array is not actually a permutation. The permutation array indices are used as sources. For example, `{"a","b","c","d"}` will become `{"d","a","b","c"}` when `ChangeOrder` is called with the permutation `{3,0,1,2}`.

```
int NumItems() const;
```

BACKWARD COMPATIBILITY ONLY. Available when `SMILE_NO_V1_COMPATIBILITY` is not defined.

8.3.2 DSL_idArray

Header file: **`idarray.h`**

```
class DSL_idArray : public DSL_stringArray
```

`DSL_intArray` is derived from `DSL_stringArray` [116]. `DSL_idArray` defines methods for creating and validating SMILE identifiers.

```
DSL_idArray(bool enableEmptyIds=false);
```

Creates an empty array and specifies the flag enabling empty identifiers.

```
DSL_idArray(std::initializer_list<const char*> il, bool enableEmptyIds=false);
```

Initializes an array and specifies the flag enabling empty identifiers.

```
DSL_idArray(const DSL_idArray &likeThisOne);
DSL_idArray& operator=(const DSL_idArray& other);
```

Implement the copy constructor and operator=.

```
bool EmptyIdsEnabled() const;
```

Returns `true` if empty identifiers can be stored in the array.

```
void SetAllEmptyIds(int newSize=-1);
```

Sets the flag to enable empty ids to `true`, and if `newSize` is greater or equal to zero resizes the array. All elements are then set to empty strings.

```
virtual int SetString(int index, const char *value) const;
```

Indexed write access. Returns `DSL_OKAY` on success, or `DSL_OUT_OF_RANGE` if the `index` is invalid, or if `value` is not a valid identifier, or an array already contains this identifier at some other index.

```
virtual int Add(const char *value);
```

Adds new element at the end of the array. Returns `DSL_OKAY` on success or `DSL_OUT_OF_RANGE` if the `index` is invalid, if `value` is not a valid identifier, or an array already contains this identifier at some other index.

```
virtual int Insert(int index, const char *value);
```

Inserts new element in the array at the specified index. Returns `DSL_OKAY` on success, or `DSL_OUT_OF_RANGE` if the `index` is invalid, if `value` is not a valid identifier, or an array already contains this identifier at some other index.

8.3.3 DSL_numArray

Header file: **numarray.h**

```
template <class T, int LOCAL_SIZE> class DSL_numArray;
```

The `DSL_numArray` template is a base class for non-templated `DSL_intArray` [122] and `DSL_doubleArray` [123]. Elements are stored in a contiguous buffer. The memory is not initialized on construction/resize. Small object optimization is implemented.

```
DSL_numArray();
DSL_numArray(int arraySize);
DSL_numArray(const DSL_numArray& other);
DSL_numArray(std::initializer_list<T> initList);
~DSL_numArray();
DSL_numArray& operator=(const DSL_numArray& other);
DSL_numArray& operator=(std::initializer_list<T> initList);
```

Implement the default constructor, copy constructors, operator=, and destructor. The methods accepting `std::initializer_list` as input are available only if C++11 standard is enabled during compilation. The constructor accepting an integer array size does not initialize the contents of the buffer.

```
template <class A> void CopyFrom(const std::vector<T, A>& v);
template <class A> void CopyTo(std::vector<T, A>& v) const
```

Copies array elements from/to `std::vector`.

```
T* begin();
```

```
T* end();
const T* begin() const;
const T* end() const;
const T* cbegin() const;
const T* cend() const;
```

The methods to enable C++ 11 range-based for loops. Can also be used with any Standard Library algorithm accepting random access iterators.

```
T& operator[](int index);
T operator[](int index) const;
```

Indexed access. The debug build of the library checks the index and asserts if the index is invalid.

```
T* Items();
const T* Items()
```

Return a pointer to the start of the memory buffer containing the array elements. Equivalent to `begin`.

```
int FindPosition(T value) const;
```

Returns the index of the specified value, or `DSL_OUT_OF_RANGE` (a negative number) if the value is not found.

```
bool Contains(T value) const;
```

Returns `true` if the specified value is present in the array, `false` otherwise.

```
bool IsEmpty() const
```

Returns `true` if the array is empty, `false` otherwise.

```
int GetSize() const;
```

Returns the number of elements in the array.

```
int SetSize(int newSize);
```

Sets the new size of the array. If the size increases, the buffer elements with indices above previous size are not initialized. Existing elements are always preserved.

```
int Reserve(int newReserved);
```

Reserves the buffer capacity, use in performance-critical scenarios. Does not change the size of the array returned from `GetSize`.

```
void Clear();
```

Sets the array size to zero.

```
void Add(T value);
```

Adds a new element at the end of the array

```
int AddExclusive(T value);
```

Adds new element at the end of the array only when the specified value is not already present in the array. If the element already exists, returns `DSL_OUT_OF_RANGE`, otherwise `DSL_OKAY`.

```
int Insert(int index, T value);
```

Inserts new element in the array at the specified index. Returns `DSL_OKAY`, or `DSL_OUT_OF_RANGE` if the index is negative or greater than the array size. `Insert` can be called with `index` equal to the array size. In such case, `Add` is invoked.

```
int Delete(int index);
```

Removes the element in the array at the specified index. The elements after the removed element are shifted toward the beginning of the array. Returns `DSL_OKAY`, or `DSL_OUT_OF_RANGE` if the index is negative or greater than the array size.

```
int DeleteByContent(T value);
```

Removes the element in the array at the specified index. The elements after the removed element are shifted toward the beginning of the array. Returns `DSL_OKAY` on success, or `DSL_OUT_OF_RANGE` if the specified value is not found.

```
int Assign(const T* src, int count);
```

Sets the size of the array to `count`, and copies the elements from the buffer starting at `src`.

```
void FillWith(T value);
```

Fills the array with the specified value. The array size does not change.

```
void Swap(DSL_numArray& other);
```

Swaps two arrays.

```
template <class T, int LOCAL_SIZE>
bool operator==(
    const DSL_numArray<T, LOCAL_SIZE>& lhs,
    const DSL_numArray<T, LOCAL_SIZE>& rhs);
template <class T, int LOCAL_SIZE>
bool operator!=(
    const DSL_numArray<T, LOCAL_SIZE>& lhs,
    const DSL_numArray<T, LOCAL_SIZE>& rhs);
```

Non-member function `operator==` returns `true` if the arrays have the same number of elements, and each element in `lhs` is equal to the corresponding element in `rhs`. `operator!=` calls `operator==` and performs the negation on the result.

```
int NumItems() const;
int IsInList(T x) const;
void UseAsList();
```

BACKWARD COMPATIBILITY ONLY. Available when `SMILE_NO_V1_COMPATIBILITY` is not defined.

8.3.4 DSL_intArray

Header file: **`intarray.h`**

```
class DSL_intArray : public DSL_numArray<int, 8>
```

`DSL_intArray` is derived from `DSL_numArray` [119] template. `DSL_intArray` defines methods related to permutations (arrays of size N containing all elements in the 0..N-1 range).

```
bool IsPermutation() const;
```

Returns `true` if the array represents a permutation, `false` otherwise.

```
bool IsIdentityPermutation() const;
```

Returns `true` if the array represents an identity permutation (all elements have values equal to their indices), `false` otherwise..

```
void MakeIdentityPermutation(int size);
```

Creates the identify permutation of specified size by resizing the array and setting the value of each element equal to its index.

```
int ChangeOrder(const DSL_intArray& permutation);
```

Changes the order of elements in the array using the specified `permutation`. Returns `DSL_OKAY` on success, or a negative error code if the specified permutation array is not actually a permutation. The permutation array indices are used as sources. For example, `{11,22,33,44}` will become `{44,11,22,33}` when `ChangeOrder` is called with the permutation `{3,0,1,2}`.

8.3.5 DSL_doubleArray

Header file: **doublearray.h**

```
class DSL_doubleArray : public DSL_numArray<double, 4>
```

`DSL_doubleArray` is derived from `DSL_numArray` 119 template.

```
int ChangeOrder(const DSL_intArray& permutation);
```

Changes the order of elements in the array using the specified `permutation`. Returns `DSL_OKAY` on success, or a negative error code if the specified permutation array is not actually a permutation. The permutation array indices are used as sources. For example, `{11.1,22.2,33.3,44.4}` will become `{44.4,11.1,22.2,33.3}` when `ChangeOrder` is called with the permutation `{3,0,1,2}`.

8.3.6 DSL_Dmatrix

Header file: **dmatrix.h**

```
DSL_Dmatrix();
```

The default constructor, creates an empty matrix without any dimensions.

```
DSL_Dmatrix(const DSL_intArray &dims);
DSL_Dmatrix(std::initializer_list<int> dims);
```

Use specified dimensions to create a matrix. The elements of the matrix are not initialized.

```
DSL_Dmatrix(const DSL_Dmatrix &mtx);
```

Copy constructor.

```
DSL_Dmatrix& operator=(const DSL_Dmatrix &mtx);
```

Assignment operator.

```
double &operator[](int index);
double operator[](int index) const;
double &Subscript(int index);
double Subscript(int index) const;
```

Access to matrix elements with a linear index.

```
double &operator[](const DSL_intArray &coords);
double operator[](const DSL_intArray &coords) const;
```

```
double &Subscript(DSL_intArray &coords);
double Subscript(DSL_intArray &coords) const;
```

Access to matrix elements using multidimensional coordinates.

```
double &operator[](const int * const * indirectCoords);
double operator[](const int * const * indirectCoords) const;
```

Access to matrix elements using indirect multidimensional coordinates.

```
int IndexToCoordinates(int index, DSL_intArray &coords) const;
```

Converts a linear index to multidimensional coordinates. Returns `DSL_OKAY` on success, or an error code on failure.

```
int CoordinatesToIndex(const int *coords) const;
int CoordinatesToIndex(DSL_intArray &coords) const;
int CoordinatesToIndex(const int * const * indirectCoords) const;
```

Converts multidimensional coordinates to linear index. Returns a converted index, or a negative error code on failure.

```
int NextCoordinates(DSL_intArray &coords) const;
```

Modifies the specified multidimensional coordinates to be the equivalent of the next linear index. Returns `DSL_OKAY` on success, or an error code on failure. Specifying the coordinates representing the last element of the matrix will cause an error code to be returned (as there is no next element).

```
int PrevCoordinates(DSL_intArray &coords) const;
```

Modifies the specified multidimensional coordinates to be the equivalent of the next linear index. Returns `DSL_OKAY` on success, or an error code on failure. Specifying the coordinates representing the first element of the matrix will cause an error code to be returned (as there is no previous element).

```
DSL_doubleArray& GetItems();
const DSL_doubleArray& GetItems() const;
```

Returns reference to a linear buffer containing the elements of the matrix.

```
const double* Ptr(int idx) const;
const double* Ptr(const DSL_intArray& coords) const;
```

Returns a pointer to the element of the matrix. Does not check for invalid inputs.

```
const DSL_intArray& GetDimensions() const;
```

Returns the dimensions of a matrix.

`const DSL_intArray& GetPreProduct() const;`

Returns reference to the pre-product array, which is useful for converting between linear and multidimensional coordinates.

`int GetNumberOfDimensions() const;`

Returns the number of dimensions of the matrix.

`int GetLastDimension() const;`

Returns the index of the last dimension of the matrix (not the *size* of the last dimension - call `GetSizeOfLastDimension` for that).

`int GetSizeOfDimension(int dimIdx) const;`

Returns the size of the specified matrix dimension, or negative error code if the index of the dimension is invalid.

`int GetSizeOfLastDimension() const;`

Returns the size of the last dimension of the matrix.

`int GetSize() const;`

Returns the total size of the matrix (which is a product of all dimensions).

`int Sum(const DSL_Dmatrix &a, const DSL_Dmatrix &b);`

Adds two matrices and stores the result in this matrix. The matrix must have dimensions compatible with both operands, or the method returns an error. Returns `DSL_OKAY` on success.

`int Subtract(const DSL_Dmatrix &a, const DSL_Dmatrix &b);`

Subtracts two matrices and stores the result in this matrix. The matrix must have dimensions compatible with both operands, or the method returns an error. Returns `DSL_OKAY` on success.

`int Add(const DSL_Dmatrix &m);`

Adds the specified matrix to this matrix. The matrix must have dimensions compatible with both operands, or the method returns an error. Returns `DSL_OKAY` on success.

```
int Add(double x);
```

Adds a scalar value to all elements in the matrix. Returns `DSL_OKAY`.

```
int Multiply(double x);
```

Multiplies all elements in the matrix by a scalar value. Returns `DSL_OKAY`.

```
int Multiply(DSL_doubleArray &v);
```

Multiplies each element of the matrix with coordinates `(a,b,c,...,z=N)` by the Nth element of vector v.

```
int FillWith(double x);
```

Fills the matrix with a specified scalar value.

```
int FillFrom(const DSL_Dmatrix& src);
```

Copies elements from the source matrix. Returns a negative error code if the dimensions of the source matrix are not compatible, or `DSL_OKAY` on success.

```
int FillFrom(const DSL_doubleArray& src);
int FillFrom(const std::vector<double>& src);
int FillFrom(std::initializer_list<double> src);
```

Copies an element from the specified linear buffer source. Returns a negative error code if the size of the source buffer is not equal to the size of the matrix, or `DSL_OKAY` on success.

```
void FillFrom(const double* src);
```

Copies elements from the specified linear buffer source.

```
int AddDimension(int dimSize);
```

Adds a dimension with a specified size.

```
int AddDimensions(const DSL_intArray &newDimensions);
```

Adds multiple dimensions with specified sizes.

```
int InsertDimension(int dimIdx, int dimSize);
```

Inserts a dimension with a specified size at a specified index.

```
int RemoveDimension(int dimIdx);
```

Removes the specified dimension. Uses the original elements with the coordinate value of zero at `dimIdx`.

```
int RemoveDimension(int dimIdx, int elemIdx);
```

Removes the specified dimension. Uses the original elements with the coordinate value of `elemIdx` at `dimIdx`.

```
int SetSingleDimension(int dimSize);
```

Sets the number of dimensions to one, and the size of the single dimension to `dimSize`.

```
int Setup(const DSL_intArray& dims);
int Setup(const int* dims, int dimCount);
```

Resets the contents of the matrix by setting new dimensions.

```
void Clear();
```

Removes all content from the matrix by setting the number of dimensions to zero.

```
void Swap(DSL_Dmatrix &mtx);
```

Swaps two matrices.

```
bool operator==(const DSL_Dmatrix& lhs, const DSL_Dmatrix& rhs);
bool operator!=(const DSL_Dmatrix& lhs, const DSL_Dmatrix& rhs);
```

Non-member function `operator==` returns `true` if the matrices have the same dimensions, and each element in `lhs` is equal to the corresponding element in `rhs`. `operator!=` calls `operator==` and performs the negation on the result.

8.4 DSL_network

Header file: **`network.h`**

`DSL_network` manages the lifetime of its nodes. Nodes should be created with `DSL_network::AddNode` and deleted with `DSL_network::DeleteNode`.

```
DSL_network();
DSL_network(const DSL_network &src, int reserved = 0);
```

```
~DSL_network();
DSL_network& operator=(const DSL_network &src);
```

Implement the default constructor, copy constructor, destructor and operator =.

```
void Clear();
```

Removes all content from the network (nodes, submodels, extended functions, properties, etc).

```
int ReadFile(const char *filename, int fileType=0, void *reserved=NULL);
```

Reads the network contents from the file specified by `filename`. If `fileType` is set to zero, the type of file is determined based on file's extension retrieved from the `filename`. The supported file type identifiers are:

Identifier	Extension	Description
DSL_XDSL_FORMAT	.xdsl	Native SMILE format
DSL_DSL_FORMAT	.dsl	Old, deprecated SMILE format
DSL_HUGIN_FORMAT	.net	Hugin
DSL_NETICA_FORMAT	.dne	Netica
DSL_INTERCHANGE_FORMAT	.dsc	Microsoft BN
DSL_ERGO_FORMAT	.erg	Ergo
DSL_KI_FORMAT	.dxp	DXpress

Returns `DSL_OKAY` on success or negative error code otherwise. If the reason for failure is a syntax error in the file or the file is not found, the error message is logged to `DSL_errorH()`.

```
int WriteFile(const char *filename, int fileType=0, void *reserved=NULL);
```

Writes the network contents to the file system. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int ReadString(const char *xdslString, void *reserved=NULL);
```

Read the network content from `xdslString`, only XDSL format is supported. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int WriteString(std::string &xdslOut, void *reserved=NULL);
```

Writes the network content to `xdslOut` as XDSL. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int UpdateBeliefs();
```

Executes inference algorithm to update node values. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
bool IsUpdateImmediate() const;
int SetUpdateImmediate(bool immediate);
```

Returns/sets the immediate update flag. If the flag is set to true, `UpdateBeliefs` will be called after changes in the network structure, parameters or evidence.

```
void SetDefaultBNAlgorithm(int algorithm);
```

Sets the algorithm used for inference in Bayesian networks. Available algorithm identifiers are:

Identifier	Exact?	Description
`DSL_ALG_BN_LAURITZEN`	Yes	Clustering (default algorithm)
`DSL_ALG_BN_RELEVANCEDECOMP`	Yes	Linear Relevance Decomposition
`DSL_ALG_BN_RELEVANCEDECOMP2`	Yes	Recursive Relevance Decomposition
`DSL_ALG_BN_EPISSAMPLING`	No	EPIS Sampling
`DSL_ALG_BN_LBP`	No	Loopy Belief Propagation
`DSL_ALG_BN_AISSAMPLING`	No	AIS Sampling
`DSL_ALG_BN_LSAMPLING`	No	Likelihood Sampling
`DSL_ALG_BN_HENRION`	No	Logic Sampling

```
void SetDefaultIDAlgorithm(int algorithm);
```

Sets the algorithm used for inference in influence diagrams. Available algorithm identifiers are:

Identifier	Description
DSL_ALG_ID_COOPERSOLVING	Policy evaluation
DSL_ALG_ID_SHACHTER	Best policy search

```
int InvalidateAllBeliefs();
```

Invalidates values in all of network's nodes. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
bool CalcProbEvidence(double &pe, bool forceChainRule = false);
```

Calculates the probability of evidence currently set in the network and stores the result in the `pe` argument. By default this method runs a modified clustering algorithm to efficiently obtain the P(e). Set `forceChainRule` to `true` to enforce the slower chain rule algorithm. Returns `true` on success, `false` on error.

```
int CalcConfidenceIntervals(const std::vector<int> &nodeHandles,
    const DSL_instanceCounts &counts, double percentage, int iterations,
    std::vector<std::vector<std::pair<double, double> > > &intervals);
```

Calculates the confidence intervals for nodes specified by `nodeHandles`. The algorithm performs a specified number of `iterations`. In each iteration, the CPTs in the network are modified using the Dirichlet distribution parametrized by original CPT values and instance counts specified by the `counts` parameter. After CPTs have been modified, the inference is performed and posterior probabilities for the specified nodes are collected. When the loop is finished, the `percentage` of posteriors for each node is used to get the lower and upper bound. The bounds are returned in the `intervals` output parameter. Each node in `nodeHandles` has its corresponding entry in intervals - it is a vector with the size equal to node's outcome count. For each outcome count, there is a `std::pair` with lower and upper bound.

See `DSL_instanceCounts` 175 reference for information on instance counts. Instance counts (especially derived from the data) may be precalculated if intervals will be required for different sets of evidence.

The algorithm uses network's `DSL_randGen` pseudo random generator for sampling. For repeatable results, use `DSL_network::GetRandGen` and `DSL_randGen::Init` to seed the generator.

Returns `DSL_OKAY` on success, or a negative error code on failure. Original CPTs are preserved, regardless of the exit status.

```
int GetRandSeed() const;
void SetRandSeed(int seed);
```

Gets/sets the seed for random generator used by sampling algorithms applied to the network, including discretization. The default is zero, which causes the generator to be initialized with system time.

```
int GetNumberOfSamples() const;
```

```
int SetNumberOfSamples(int numSamples);
```

Gets/sets the number of samples generated by sampling inference algorithms.

```
bool IsNoisyDecompEnabled() const;
void EnableNoisyDecomp(bool enable);
```

Gets/sets the flag enabling noisyMAX decomposition.

```
int GetNoisyDecompLimit() const;
void SetNoisyDecompLimit(int limit);
```

Gets/sets the limit on the number of noisyMAX parent nodes as the result of noisyMAX decomposition runs. The number of parents applies to temporary data structures managed by the inference algorithm and does not modify the `DSL_network` or its nodes.

```
bool IsRejectOutlierSamplesEnabled() const
int EnableRejectOutlierSamples(bool enable);
```

Gets/sets the flag controlling the sample rejection during stochastic inference in continuous and hybrid models.

```
void GetExtFunctions(std::vector<std::string> &functions,
    bool includeDistributions = true) const;
```

Returns the extended functions defined for the network. By default, functions that use (explicitly and implicitly) random number generators are returned. Pass `includeDistributions` parameter set to false to retrieve only deterministic functions.

```
int SetExtFunctions(const std::vector<std::string> &functions,
    int &errFxnIdx, int &errPos, std::string &errMsg);
```

Sets the extended functions. On success, returns `DSL_OKAY`. On failure, the error code is returned and `errFxnIdx` contains the index of invalid function definition, `errPos` contains the position of the error within the definition and `errMsg` is the error message. See the *custom functions reference* 215 section for details.

```
int GetNumberOfDiscretizationSamples() const;
void SetNumberOfDiscretizationSamples(int numSamples);
```

Gets/sets the number of samples to be generated for each CPT column during the discretization of continuous nodes.

```
bool IsZeroAvoidanceEnabled() const;
void EnableZeroAvoidance(bool enable);
```

Gets/sets the flag controlling the discretization of non-deterministic continuous nodes. If set, the node discretization algorithm adds one artificial sample to each empty discretization bin. This only applies to nodes with equations explicitly including probability distribution functions.

```
int ClearAllEvidence();
```

Clears all evidence in the network.

```
int GetAllEvidenceNodes(DSL_intArray &evHandles);
```

Returns the handles of all evidence nodes in `evHandles`.

```
DSL_node* GetNode(int handle);
DSL_node* GetNode(const char *nodeId);
const DSL_node* GetNode(int handle) const;
const DSL_node* GetNode(const char *nodeId) const;
```

Returns a pointer to the node identifier by handle or nodeId, `NULL` otherwise.

```
int FindNode(const char *nodeId) const;
```

Returns a handle for the node with identifier equal to `nodeId`, negative with error code otherwise.

```
int GetFirstNode() const;
```

Returns a handle for the first node in the network, `DSL_OUT_OF_RANGE` if network has no nodes.

```
int GetNextNode(int handle) const;
```

Returns node handle following the specified handle. If the specified handle represented the last node in the network, returns `DSL_OUT_OF_RANGE`.

```
int GetLastNode() const;
```

Returns the last node handle. If the network is empty, returns `DSL_OUT_OF_RANGE`.

```
int GetNumberOfNodes() const;
```

Returns the number of nodes in the network.

```
int GetAllNodes(DSL_intArray &handles) const;
int GetAllNodeIds(DSL_idArray &ids) const;
```

Return all node handles/ids in their output parameters.

```
const DSL_intArray& PartialOrdering() const;
```

Returns reference to an array containing all node handles in partial order (the ancestor handles always precede their descendant handles). Performs actual ordering only if it could change since last PartialOrdering call.

```
int AddNode(int nodeType, const char *nodeId);
```

Creates a new node and returns its handle. If `nodeId` is `NULL`, the unique identifier will be created by the network. Supported node types are listed in the *Node types* 114 section.

Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int DeleteNode(int handle);
```

Deletes the node specified by handle. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int DeleteAllNodes();
```

Deletes all nodes in the network.

```
int AddArc(int parentHandle, int childHandle,
    dsl_arcType layer = dsl_normalArc);
```

Adds an arc between nodes specified by `parentHandle` and `childHandle`. By default, the arcs are created on the 'normal' arc layer, which represents the conditional probabilities (`layer` is set to `dsl_normalArc`). Additionally, the network can include conditional observation cost, which influences the output from diagnostic algorithms. To create the observation cost arc, set `layer` to `dsl_costObserve`.

Returns `DSL_OKAY` on success, or a negative error code on failure. If adding an arc would result in cycle in the network, the status code is `DSL_CYCLE_DETECTED`.

```
int RemoveArc(int parentHandle, int childHandle,
    dsl_arcType layer = dsl_normalArc);
```

Removes an arc. The default arc layer is `dsl_normalArc`. To remove conditional observation cost arc, use `dsl_costObserve` as the `layer` parameter. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
const DSL_intArray& GetParents(int nodeHandle) const;
```

Returns reference to the `DSL_intArray` containing handles of parent nodes of a node specified by `nodeHandle`. If `nodeHandle` is invalid, the return value is undefined.

```
const DSL_intArray& GetChildren(int nodeHandle) const;
```

Returns a reference to the `DSL_intArray` containing handles of child nodes of a node specified by `nodeHandle`. If `nodeHandle` is invalid, the return value is undefined.

```
int GetDescendants(int nodeHandle, DSL_intArray &descendants);
```

Copies all handles of node's descendants to the `descendants` output array.

```
int GetAncestors(int nodeHandle, DSL_intArray &ancestors);
```

Copies all handles of node's ancestors to the `ancestors` output array.

```
const DSL_intArray& GetCostParents(int nodeHandle) const;
```

Returns a reference to a `DSL_intArray` containing handles of cost parent nodes of a node specifed by `nodeHandle`. If `nodeHandle` is invalid, the return value is undefined.

```
const DSL_intArray& GetCostChildren(int nodeHandle) const;
```

Returns a reference to a `DSL_intArray` containing handles of cost children nodes of a node specifed by `nodeHandle`. If `nodeHandle` is invalid the return value is undefined.

```
int ReverseArc(int parentHandle, int childHandle);
```

Reverses an arc, preserving the joint probability distribution in the network. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int IsArcNecessary(int parentHandle, int childHandle,
    double epsilon, bool &necessary) const;
```

Checks if the arc between `parentHandle` and `childHandle` is necessary. The arc is considered to be necessary when child’s conditional probability distributions for different parent states are different. The check is performed using Hellinger’s distance with specified `epsilon`. The output of the check is returned in the `necessary` parameter. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int RemoveAllArcs();
```

Removes all arcs in the network.

```
int MarginalizeNode(int nodeHandle, DSL_progress *progress = NULL);
```

Removes the node specified by `nodeHandle`, but preserves the joint probability distribution over the remaining nodes. The marginalization may take a significant amount of time, therefore the caller may specify an instance of object derived from the `DSL_progress` class to monitor the progress and optionally cancel the operation. Returns `DSL_OKAY` on success, or a negative error code on failure. If canceled, the function returns the `DSL_INTERRUPTED` error code.

```
int MakeUniform(const std::vector<int> *nodeFilter=NULL);
```

Uniformizes the parameters for all nodes in the network, or, if `nodeFilter` vector is specified, only nodes with handles in the vector.

```
int MakeRandom(DSL_randGen *extRandGen=NULL,
    const std::vector<int> *nodeFilter=NULL););
```

Randomizes the parameters of all node definitions in the network, or, if `nodeFilter` vector is specified, only nodes with handles in the vector. Uses `extRandGen` if specified, or network's internal random generator.

```
int GetNumberOfTargets() const;
```

Returns the number of target nodes in the network.

```
int IsTarget(int nodeHandle);
```

For a valid node handle, returns nonzero if the node was set as target, zero otherwise. If the handle is invalid, returns a negative status code.

```
int SetTarget(int nodeHandle, bool target);
```

Sets the target flag on node specified by `nodeHandle`. Returns `DSL_OKAY` if `nodeHandle` is valid and the flag has actually changed. Returns `DSL_INVALID_VALUE` if `nodeHandle` was valid, but the value of `target` is the same as the value of the current target flag of the node (i.e., when trying to set the target flag of a node that is already a target or when trying to clear the target flag of a node that is not a target).

```
int ClearAllTargets();
```

Resets the target flags on all nodes. With no explicit target nodes all nodes in the network are updated.

```
DSL_case* AddCase(const std::string & name);
```

Adds new case with a specified name. Returns the pointer to the newly created case.

```
DSL_case* GetCase(int index) const;
```

Returns the pointer to the case specified by the zero-based case index. If the index is negative or greater or equal to the number of cases, returns `NULL`.

```
DSL_case* GetCase(const std::string & name) const;
```

Returns a pointer to the case specified by `name`. If the case is not found, returns `NULL`.

```
int DeleteCase(int index);
```

Deletes the case specified by the index. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
void DeleteAllCases();
```

Removes all cases from the network.

```
int GetNumberOfCases() const;
```

Returns the number of cases defined for the network

```
void EnableSyncCases(bool sync);
```

Enables or disables case synchronization. If enabled, structural changes in node definitions, like creation or removal of node outcomes, will be reflected in the data stored in cases. The case synchronization mode is enabled by default.

```
bool IsEnableSyncCases() const;
```

Returns `true` if case synchronization is enabled, `false` otherwise.

```
int UnrollNetwork(DSL_network &unrolled, std::vector<int> &unrollMap) const;
```

Creates unrolled network from a DBN. The `unrollMap` output parameter contains a mapping from unrolled network to the original DBN. Node with the handle `h` in the unrolled network represents the original node with handle `unrollMap[h]`. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int GetMaxTemporalOrder() const;
int GetMaxTemporalOrder(int nodeHandle) const;
```

Returns the maximum order of temporal arcs in the entire network or for a specific node.

```
int GetTemporalOrders(int nodeHandle, DSL_intArray &orders) const;
```

Gets all temporal orders for the specified node.

```
int GetNumberOfSlices() const;
int SetNumberOfSlices(int slices);
```

Gets/sets the number of slices in the unrolled DBN.

```
int AddTemporalArc(int parent, int child, int order);
```

Adds a temporal arc of a specified `order` between nodes specified by the `parent` and `child` handles. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int RemoveTemporalArc(int parent, int child, int order);
```

Removes a temporal arc of a specified `order` between nodes specified by the `parent` and `child` handles. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
bool TemporalArcExists(int parent, int child, int order) const;
```

Returns `true` if a temporal arc of a specified `order` exists between the nodes specified by the `parent` and `child` handles.

```
int IsTemporalArcNecessary(int parent, int child,
    int order, double epsilon, bool &necessary) const;
```

Checks if the temporal arc with a specified `order` between the nodes specified by the `parent` and `child` handles is necessary. The arc is considered necessary when child's conditional distributions for different parent states are different. The check is performed using Hellinger's distance with a specified `epsilon`. The output of the check is returned in the `necessary` parameter. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
dsl_temporalType GetTemporalType(int nodeHandle) const;
```

Gets the temporal type of node. The temporal type is defined by the following enumeration:

```
enum dsl_temporalType { dsl_normalNode, dsl_anchorNode, dsl_terminalNode, dsl_plateNode };
```

```
int SetTemporalType(int nodeHandle, dsl_temporalType type);
```

Sets the temporal type of node. Returns `DSL_OKAY` on success, or a negative error code on failure.

The following temporal type transitions are forbidden and result in an error:

1. From `dsl_plateNode` to another type when the node has incoming or outgoing temporal arcs.
2. To `dsl_plateNode` when the node has `dsl_terminalNode` parents, or `dsl_anchorNode` children.
3. To `dsl_anchorNode` when the node has `dsl_plateNode` or `dsl_terminalNode` parents.
4. To `dsl_terminalNode` when the node has `dsl_plateNode`, `dsl_normalNode` or `dsl_anchorNode` children.
5. To `dsl_normalNode` when the node has `dsl_plateNode` or `dsl_terminalNode` parents.

```
int GetTemporalChildren(int parent,
    std::vector<std::pair<int, int> > &children) const;
```

Retrieves information about the temporal children of a specified parent node. Each temporal arc originating in the parent has one corresponding element in the output vector. The element is the `std::pair<int,int>`, where first element is a child node handle, and second element is a temporal order. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int GetTemporalParents(int child, int order, DSL_intArray &parents) const;
```

Retrieves the handles of temporal parents of the specified temporal order. Note that parents with order *x* are in general only the subset of nodes which index temporal definition of order *x*. Use `GetUnrolledParents` to get all parents indexing the temporal definition. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int GetUnrolledParents(int child, int order,
    std::vector<std::pair<int, int> > &parents) const;
```

Retrieves the handles and orders of temporal parents indexing the temporal definition of a specified order. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int GetUnrolledParents(int child,
    std::vector<std::pair<int, int> > &parents) const;
```

Retrieves the handles and orders of temporal parents. Returns `DSL_OKAY` on success, or a negative error code on failure.

8.5 DSL_node

Header file: **`node.h`**

`DSL_node` objects are always created and destroyed by their parent `DSL_network`. While `DSL_node` class implements copy constructor and operator=, these members should never be used directly. Never use `delete` on a node pointer.

To convert an integer node handle to a `DSL_node` pointer, use `DSL_network::GetNode(int)` function.

To convert a node identifier to `DSL_node` pointer directly, use `DSL_network::GetNode(const char*)` function. Alternatively, use `DSL_network::FindNode(const char*)` first, followed by `DSL_network::GetNode(int)`.

```
int Handle();
```

Returns node's handle. The handle does not change during the lifetime of the node.

```
DSL_network* Network();
const DSL_network* Network() const;
```

Returns a pointer to node’s network. The network does not change during the lifetime of the node.

```
const char* GetId() const;
```

Returns node's identifier.

```
int SetId(const char *newId);
```

Sets node's identifier. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
const char* GetName() const;
int SetName(const char *newName);
```

Gets/sets node's name. The name is only used for display purposes, it is not validated for uniqueness, etc.

```
const char* GetComment() const;
int SetComment(const char *newComment);
```

Gets/sets node's description. Description is only used for display purposes.

```
DSL_nodeInfo &Info();
const DSL_nodeInfo& Info() const { return info; }
```

Returns a reference to a `DSL_nodeInfo` structure containing node attributes that are not directly related to calculations, including identifier, name, comment, screen position, etc. In SMILE 1.x, access to node name and description was possible only through `Info()`, SMILE 2 has methods for direct access, like `SetName`. The implementation still uses the `DSL_nodeInfo`, which means that SMILE 1.x code should compile and run correctly.

```
int ChangeType(int newType);
```

Changes node type. The `newType` is an integer node type identifier (see *Node types* 114). After successful type change, node's definition and value objects change, but node handle remains unchanged. Some incoming and outgoing arcs may be removed during type change, when the new type is not compatible with existing node parents and children.

Returns `DSL_OKAY` on success, or a negative error code on failure.

```
DSL_nodeDef* Def();
const DSL_nodeDef* Def() const;
template <class defT> defT* Def();
template <class defT> const defT* Def() const;
```

Returns a pointer to the node’s definition. The definition object is managed by the node and should never be created or deleted explicitly. The templated methods do not perform any runtime type checking for `defT`, they are just calling `static_cast` internally.

```
DSL_nodeVal* Val();
const DSL_nodeVal* Val() const;
template <class valT> valT* Val();
template <class valT> const valT* Val() const;
```

Returns a pointer to node's definition. The definition object is managed by the node and should never be created or deleted explicitly. The templated methods do not perform any runtime type checking for `valT`, they are just calling `static_cast` internally.

```
DSL_nodeDef* Definition();
DSL_nodeVal* Value();
```

BACKWARD COMPATIBILITY ONLY. Available when `SMILE_NO_V1_COMPATIBILITY` is not defined.

```
dsl_diagType GetDiagType() const;
int SetDiagType(dsl_diagType diagType);
```

Gets/sets node's diagnostic role. The `dsl_diagType` is defined as `enum dsl_diagType { dsl_diagFault, dsl_diagObservation, dsl_diagAux };`

```
bool IsDiagRanked() const;
bool IsDiagMandatory() const;
void SetDiagRanked(bool ranked);
void SetDiagMandatory(bool ranked);
```

Get/set node's diagnostic ranked and mandatory flags.

```
const DSL_Dmatrix& GetCosts() const;
```

Gets node's observation cost matrix. Observation cost is optional. If present, it is used during the calculation of the diagnostic value for ranked observations. The cost can be conditional. To make the cost conditional, use `DSL_network::AddArc` with `dsl_costObserve` as the arc layer.

```
int SetCosts(const DSL_doubleArray& costs);
int SetCosts(const DSL_Dmatrix& costs);
int SetCosts(const std::vector<double>& costs);
```

Set node's observation cost. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
const char* GetDiagQuestion() const { return diagQuestion; }
void SetDiagQuestion(const char* question);
```

Gets/sets node's diagnostic question. The diagnostic question is only used for display purposes.

8.6 Node definitions

8.6.1 DSL_nodeDef

Header file: **`nodedef.h`**

`DSL_nodeDef` is a base class in the node definition class hierarchy. Various (public) virtual `DSL_nodeDef` class methods called internally by `DSL_network` to manage its nodes are not listed in this section, because they should never be called by code that is not part of SMILE library.

Objects of classes derived from `DSL_nodeDef` are created and destroyed by their parent `DSL_node` 138. Never use `delete` on a node definition pointer.

Use `DSL_node::Def` method to obtain a pointer to node's definition.

```
template <class T> T* As();
template <class T> const T* As() const;
```

Statically cast `this` to specified type `T`. Note that there is no runtime type checking. `DSL_node::Def()->As<T>` is equivalent to `DSL_node::Def<T>()`.

```
DSL_network* Network();
const DSL_network* Network() const;
```

Returns a pointer to node's network. The network does not change during the lifetime of the node definition object.

```
int Handle() const;
```

Returns a handle of node's definition. The handle does not change during the lifetime of the node definition object.

```
const DSL_intArray& GetParents() const;
const DSL_intArray& GetChildren() const;
```

Return a reference to parent/children array. These helper methods call `DSL_network::GetParents/GetChildren`.

```
virtual int GetType() const = 0;
```

Returns a numeric identifier of the definition type.

```
virtual const char* GetTypeName() const = 0;
```

Returns a definition type name.

```
virtual int AddOutcome(const char *outcomeId);
```

Adds node outcome with the specified identifier. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeDef` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in `DSL_discDef` [145].

```
virtual int InsertOutcome(int outcomeIndex, const char *outcomeId);
```

Inserts node outcome with the specified identifier at the specified index. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeDef` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in `DSL_discDef` [145].

```
virtual int RemoveOutcome(int outcomeIndex);
```

Removes node outcome at the specified index. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeDef` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in `DSL_discDef` [145].

```
virtual int GetNumberOfOutcomes();
```

Returns the number of node outcomes or a negative error code if the node type does not have outcomes (e.g., it is not discrete). Note that equation nodes, even if they have discretization intervals, return a negative number from `GetNumberOfOutcomes`. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeDef` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in `DSL_discDef` [145].

```
virtual int RenameOutcome(int outcomeIndex, const char *newId);
```

Changes node outcome identifier at the specified index. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeDef` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in `DSL_discDef` [145].

```
virtual int RenameOutcomes(const DSL_idArray& newOutcomeIds);
virtual int RenameOutcomes(const std::vector<std::string>& newOutcomeIds);
```

Change all outcome identifiers. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeDef` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in DSL_discDef [145].

```
virtual const DSL_idArray* GetOutcomeIds() const;
```

Returns a `const` pointer to an identifier array with all outcome identifiers. Returns `NULL` when the node does not have outcomes.

Overridden in DSL_discDef [145].

```
virtual int SetNumberOfOutcomes(int outcomeCount);
```

Sets the number of node outcomes. If the outcome count increases, new outcomes will have automatically generated identifiers. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeDef` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in DSL_discDef [145].

```
virtual int SetNumberOfOutcomes(const DSL_idArray& outcomeIds);
```

Sets the number of node outcomes and renames them by using specified identifiers. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeDef` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in DSL_discDef [145].

```
virtual int ChangeOrderOfOutcomes(const DSL_intArray& newOrder);
```

Reorders node outcomes. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeDef` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in DSL_discDef [145].

```
virtual int ChangeOrderOfOutcomesWithAddAndRemove(
    const DSL_idArray& ids, const DSL_intArray& newOrder);
```

Reorders node outcomes and optionally adds/removes existing outcomes. The outcomes to add should be represented by `-1` in `newOrder`. If the original outcome index is not present in `newOrder`, the outcome is removed. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeDef` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in DSL_discDef [145].

```
virtual const DSL_Dmatrix* GetMatrix() const;
```

Returns a `const` pointer to a `DSL_Dmatrix` object with node's numeric parameters, or `NULL` when node does not have numeric parameters. For all chance and discrete deterministic nodes, the matrix is a CPT (even if the node is canonical or qualitative). The matrix returned by utility nodes contains utilities.

Overridden in DSL_discDef [145] and DSL_lazyDef [151].

```
int GetDefinition(const DSL_Dmatrix** parameters) const;
```

Sets the numeric parameters of the node. While `GetDefinition` is not virtual, the method calls a protected virtual function defined in `DSL_nodeDef` and overridden in the derived classes. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int SetDefinition(const DSL_Dmatrix& parameters);
int SetDefinition(const DSL_doubleArray& parameters);
int SetDefinition(const std::vector<double>& parameters);
int SetDefinition(std::initializer_list<double> parameters);
```

Sets the numeric parameters of the node. While none of the `SetDefinition` overloads are virtual, the methods call a protected virtual function defined in `DSL_nodeDef` and overridden in the derived classes. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
virtual int RawDefinition(DSL_Dmatrix** parameters);
```

Returns a pointer to a writable `DSL_Dmatrix` object with node's numeric parameters, or `NULL` if node does not have numeric parameters. Should be only used by performance-critical code to modify the definition of the node (parameter learning, for example). Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeDef` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in DSL_lazyDef [151].

```
const DSL_Dmatrix* GetTemporalDefinition(int order) const;
```

For a plate node in a DBN, returns a pointer to an array representing the temporal parameters of the node with the specified order, or `NULL` if node does not have temporal parameters of the specified order.

```
int SetTemporalDefinition(int order, const DSL_Dmatrix& temporal);
int SetTemporalDefinition(int order, const std::vector<double>& temporal);
int SetTemporalDefinition(int order, const DSL_doubleArray& temporal);
```

For a plate node in a DBN, sets the temporal parameters of the node for the specified temporal order. While none of the `SetTemporalDefinition` overloads are virtual, the methods delegate the parameter modification to a protected virtual function defined in `DSL_nodeDef` and overridden in the derived classes. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
virtual int MakeUniform();
```

Uniformizes the definition parameters. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeDef` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in `DSL_cpt` [149], `DSL_truthTable` [150], `DSL_ciDef` [153] and `DSL_utility` [157].

```
virtual int MakeRandom(DSL_randGen& randGen);
```

Randomizes the definition parameters using the specified random generator. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeDef` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

```
const DSL_idArray* GetOutcomesNames() const;
int NodeTypeIs(int flags) const
```

BACKWARD COMPATIBILITY ONLY. Available when `SMILE_NO_V1_COMPATIBILITY` is not defined.

8.6.2 DSL_discDef

Header file: **`discdef.h`**

```
class DSL_discDef : public DSL_nodeDef
```

The abstract class `DSL_discDef` is derived from `DSL_nodeDef` [141] and provides outcome implementation for discrete nodes. The outcomes of discrete nodes can be defined in the following ways:

1. Identifiers only, with no numeric information associated with outcomes.
2. Intervals only, with no text identifiers associated with outcomes.
3. Identifiers and intervals.
4. Identifiers and numeric point values. The point values always require text identifiers.

```
virtual int AddOutcome(const char *outcomeId);
```

Overridden method from `DSL_nodeDef` [141]. Adds node outcome with the specified identifier. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
virtual int InsertOutcome(int outcomeIndex, const char *outcomeId);
```

Overridden method from `DSL_nodeDef` [141]. Inserts node outcome with the specified identifier at the specified index. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
virtual int RemoveOutcome(int outcomeIndex);
```

Overridden method from DSL_nodeDef [141]. Removes node outcome at the specified index. Returns DSL_OKAY on success, or a negative error code on failure.

```
virtual int GetNumberOfOutcomes();
```

Overridden method from DSL_nodeDef [141]. Returns the number of node outcomes. The minimum number of outcomes for discrete nodes is two.

```
virtual int RenameOutcome(int outcomeIndex, const char *newId);
```

Overridden method from DSL_nodeDef [141]. Changes node outcome identifier at the specified index. Returns DSL_OKAY on success, or a negative error code on failure.

```
virtual int RenameOutcomes(const DSL_idArray& newOutcomeIds);
virtual int RenameOutcomes(const std::vector<std::string>& newOutcomeIds);
```

Overridden method from DSL_nodeDef [141]. Changes all outcome identifiers. Returns DSL_OKAY on success, or a negative error code on failure.

```
virtual const DSL_idArray* GetOutcomeIds() const;
```

Overridden method from DSL_nodeDef [141]. Returns the const pointer to an identifier array with all outcome identifiers. When node outcomes are defined with intervals and no identifiers, the array contains empty strings.

```
virtual int SetNumberOfOutcomes(int outcomeCount);
```

Overridden method from DSL_nodeDef [141]. Sets the number of node outcomes. If the outcome count increases, new outcomes will have automatically generated identifiers. Returns DSL_OKAY on success, or a negative error code on failure.

```
virtual int SetNumberOfOutcomes(const DSL_idArray& outcomeIds);
```

Overridden method from DSL_nodeDef [141]. Sets the number of node outcomes and renames them by using specified identifiers.Returns DSL_OKAY on success, or a negative error code on failure.

```
virtual int ChangeOrderOfOutcomes(const DSL_intArray& newOrder);
```

Overridden method from DSL_nodeDef [141]. Reorders node outcomes. Returns DSL_OKAY on success, or a negative error code on failure.

```
virtual int ChangeOrderOfOutcomesWithAddAndRemove(
    const DSL_idArray& ids, const DSL_intArray& newOrder);
```

Overridden method from DSL_nodeDef [141]. Reorders node outcomes and optionally adds/removes existing outcomes. The outcomes to add should be represented by -1 in newOrder. If the original outcome index is not present in newOrder, the outcome is removed. Returns DSL_OKAY on success, or a negative error code on failure.

```
bool HasFixedOutcomes() const;
```

Returns true when the node has a fixed set of outcomes. Currently only qualitative nodes have fixed outcomes.

```
bool HasIdentifiers() const;
```

Returns true if the node has outcome identifiers. The identifiers can be omitted only if outcomes are defined by numeric intervals.

```
bool HasIntervals() const;
```

Returns true if the node has outcome intervals.

```
int RemoveIntervals();
```

Removes outcome intervals and creates default outcome identifiers, if there were no outcome identifiers. Returns DSL_OKAY on success, or a negative error code on failure.

```
int GetInterval(int outcomeIndex, double& lo, double& hi) const;
```

Gets the bounds of the interval for the outcome specified by outcomeIndex. Returns DSL_OKAY on success, or a negative error code on failure.

```
int GetIntervals(DSL_doubleArray& intervals) const;
int GetIntervals(std::vector<double>& intervals) const;
```

Get all interval boundaries. For a node definition with N outcomes, the output array/vector will have N+1 elements. Returns DSL_OKAY on success, or a negative error code on failure.

```
int SetIntervals(const DSL_doubleArray& intervals, bool removeIds=false);
int SetIntervals(const std::vector<double>& intervals, bool removeIds=false);
int SetIntervals(std::initializer_list<double> intervals, bool removeIds=false);
```

Sets all interval boundaries. For an array/vector/initializer list with N elements, the number of outcomes is set to N-1 (therefore, at least three elements are needed). The outcome identifiers are removed if removeIds is set to true. If fewer than three interval edges are specified or the edges are not specified in increasing order, the function fails. Point intervals are allowed. Returns DSL_OKAY on success, or a negative error code on failure.

```
int Discretize(double value) const;
```

Converts the continuous value to a non-negative integer outcome index, based on the specified outcome intervals. If the node has no outcome intervals, returns `DSL_NO_ITEM`. Returns `DSL_OUT_OF_RANGE` if `value` is smaller than leftmost interval bound, or is larger than rightmost interval bound.

```
bool HasPointValues() const;
```

Returns `true` if the node has outcome point values.

```
int RemovePointValues();
```

Removes outcome point values. Returns `DSL_OKAY` on success or a negative error code if node definition has no point values.

```
int PointValueToOutcomeIndex(double pointValue) const;
```

Gets the outcome index for the point value specified by `pointValue`. Returns DSL_OKAY on success or a negative error code if node definition has no point values, or if `pointValue` is not found.

```
int GetPointValue(int outcomeIndex, double& pointValue) const;
```

Gets the point value for the outcome specified by `outcomeIndex`. Returns DSL_OKAY on success or a negative error code if node definition has no point values, or if `outcomeIndex` is invalid.

```
int GetPointValues(DSL_doubleArray& pointValues) const;
int GetPointValues(std::vector<double>& pointValues) const;
```

Gets all point values. For a node definition with N outcomes, the output array/vector will have N elements. Returns `DSL_OKAY` on success or a negative error code if node definition has no point values.

```
int SetPointValues(const DSL_doubleArray& pointValues);
int SetPointValues(const std::vector<double>& pointValues);
int SetPointValues(std::initializer_list<double> pointValues);
```

Sets all point values. The size of an array/vector/initializer list must be equal to the current number of node outcomes. Returns `DSL_OKAY` on success or a negative error code if the number of point values specified is not equal to the outcome count.

```
int GetDefaultOutcome() const;
```

Returns the index of the default outcome, or a negative number if there is no default outcome.

```
int SetDefaultOutcome(int newDefOutcome);
```

Sets the index of the default outcome. To remove the default outcome flag, specify -1 as newDefOutcome. Returns DSL_OKAY on success or a negative error code if newDefOutcome is equal to or larger than the number of outcomes.

```
bool IsFaultOutcome(int outcomeIndex) const;
```

Returns true if the outcome specified by outcomeIndex is a diagnostic fault.

```
int SetFaultOutcome(int outcomeIndex, bool fault);
```

Sets the diagnostic fault flag associated with the outcome specified by outcomeIndex. Returns DSL_OKAY on success, or a negative error code on failure.

```
const DSL_documentation* GetOutcomeDocumentation(int outcomeIndex) const;
const char* GetOutcomeLabel(int outcomeIndex) const;
const char* GetOutcomeComment(int outcomeIndex) const;
const char* GetOutcomeFix(int outcomeIndex) const;
```

Get various display-only diagnostic attributes associated with the outcome specified by outcomeIndex. If outcomeIndex is invalid or if there is no attribute associated with the outcome, returns NULL.

```
int SetOutcomeDocumentation(int outcomeIndex, const DSL_documentation* doc);
int SetOutcomeLabel(int outcomeIndex, const char* label);
int SetOutcomeComment(int outcomeIndex, const char* comment);
int SetOutcomeFix(int outcomeIndex, const char* fix);
```

Set various display-only diagnostic attributes associated with the outcome specified by outcomeIndex. Returns DSL_OKAY on success, or a negative error code on failure.

8.6.3 DSL_cpt

Header file: **defcpt.h**

```
class DSL_cpt : public DSL_discDef
```

DSL_cpt provides implementation for CPT nodes, inheriting most of the functionality from DSL_discDef 145. Conditional probabilities are stored in the DSL_Dmatrix object.

```
virtual int GetType() const;
```

Overridden method from DSL_nodeDef 141. Returns DSL_CPT.

```
virtual const char* GetTypeName() const;
```

Overridden method from DSL_nodeDef 141. Returns "CPT".

```
const DSL_Dmatrix& GetProbabilities() const;
```

Returns a reference to a matrix of conditional probabilities. Note that you do not need to cast `DSL_nodeDef` pointer obtained from `DSL_node::Def` to `DSL_cpt` in order to get the probabilities. The methods `GetMatrix` and `GetDefinition` (declared in `DSL_nodeDef`) will return a pointer to the same `DSL_Dmatrix` object. `GetProbabilities` is defined to provide a method with a name closely reflecting Bayesian network terminology.

8.6.4 DSL_truthTable

Header file: **deftruthtable.h**

```
class DSL_truthTable : public DSL_cpt
```

`DSL_truthTable` is a specialization of `DSL_cpt` [149], maintaining a deterministic CPT containing only zeros and ones.

```
virtual int GetType() const;
```

Overridden method from `DSL_nodeDef` [141]. Returns `DSL_TRUTHTABLE`.

```
virtual const char* GetTypeName() const;
```

Overridden method from `DSL_nodeDef` [141]. Returns `"TRUTHTABLE"`.

```
int GetResultingStates(DSL_intArray& states) const;
int GetResultingStates(DSL_stringArray& states) const;
```

Fill the output array with indices or identifiers of resulting states. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int SetResultingStates(const DSL_stringArray& states);
int SetResultingStates(const DSL_intArray& states);
```

Set the resulting states using state indices or identifiers. Fail when the size of `states` array is not equal to the number of columns in the truth table, or when the elements of the array do not correspond to the node outcomes. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int GetTemporalResultingStates(int order, DSL_intArray &states) const;
```

For a plate node in a DBN, gets the resulting states for a specified temporal order. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int SetTemporalResultingStates(int order, const int* states);
```

For a plate node in a DBN, sets the resulting states for a specified temporal order. Returns `DSL_OKAY` on success, or a negative error code on failure.

8.6.5 DSL_lazyDef

Header file: **lazydef.h**

```
class DSL_lazyDef : public DSL_discDef
```

`DSL_lazyDef` derives from DSL_discDef [145] and supports node definition specified by canonical or qualitative parameters. Such nodes are required to provide their complete CPTs for inference algorithms. To avoid expensive computation, the CPT is calculated on demand and invalidated when node parameters change.

The class does not define any user-callable methods.

8.6.6 DSL_qualDef

Header file: **qualdef.h**

```
class DSL_qualDef : public DSL_lazyDef
```

`DSL_qualDef` derives from DSL_lazyDef [151], and provides common implementation for qualitative node types.

The class does not define any user-callable methods.

8.6.7 DSL_demorgan

Header file: **qualdef.h**

```
class DSL_demorgan : public DSL_qualDef
```

`DSL_demorgan` derives from DSL_qualDef [151], and implements DeMorgan qualitative node. DeMorgan nodes are parametrized by a vector of parent weights and parent types, and by prior belief. These parameters are used to derive a complete conditional probability table (on demand, when inference algorithm requires one).

Parent weights are required to be in the [0..1] range. Parent types are integers. The following pre-processor defines are valid parent types:

```
DSL_DEMORGAN_INHIBITOR
DSL_DEMORGAN_REQUIREMENT
DSL_DEMORGAN_CAUSE
DSL_DEMORGAN_BARRIER
```

```
virtual int GetType() const;
```

Overridden method from DSL_nodeDef [141]. Returns `DSL_DEMORGAN`.

```
virtual const char* GetTypeName() const;
```

Overridden method from DSL_nodeDef [141]. Returns `"DEMORGAN"`.

```
double GetPriorBelief() const;
```

Returns prior belief.

```
int SetPriorBelief(double value) const;
```

Sets prior belief. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
double GetParentWeight(int parentIndex) const;
```

Returns a weight for a parent specified by `parentIndex`, or a negative number if `parentIndex` is invalid.

```
const DSL_doubleArray& GetParentWeights() const;
```

Returns a reference to an array containing parent weights.

```
int SetParentWeight(int parentIndex, double weight) const;
```

Sets `weight` for a parent specified by `parentIndex`. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int SetParentWeights(const DSL_doubleArray& weights);
```

Sets all parent weights. Returns `DSL_OKAY` on success, or a negative error code on failure. Fails when the size of the `weights` array is not equal to the number of parents or at least one element in the array is less than zero or greater than one.

```
int GetParentType(int parentIndex) const;
```

Returns parent type for a parent specified by `parentIndex` or a negative error code if `parentIndex` is invalid.

```
const DSL_intArray& GetParentTypes() const;
```

Returns a reference to an array containing parent types.

```
int SetParentType(int parentIndex, int parentType) const;
```

Sets parent type for a parent specified by `parentIndex`. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int SetParentTypes(const DSL_intArray& parentTypes);
```

Sets all parent types. Returns `DSL_OKAY` on success, or a negative error code on failure. Fails when the size of the `parentTypes` array is not equal to the number of parents, or at least one element in the array is an invalid parent type.

```
double GetTemporalParentWeight(int order, int parentIndex) const;
```

For a plate node in a DBN, gets the parent weight for a specified temporal order and parent index. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int SetTemporalParentWeight(int order, int parentIndex, double weight);
```

For a plate node in a DBN, sets the parent weight for a specified temporal order and parent index. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int GetTemporalParentType(int order, int parentIndex) const;
```

For a plate node in a DBN, gets the parent type for a specified temporal order and parent index. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int SetTemporalParentType(int order, int numParent, int parentType);
```

For a plate node in a DBN, sets the parent type for a specified temporal order and parent index. Returns `DSL_OKAY` on success, or a negative error code on failure.

8.6.8 DSL_ciDef

Header file: **`cidef.h`**

```
class DSL_ciDef : public DSL_lazyDef
```

`DSL_ciDef` derives from `DSL_lazyDef` [151], and provides common implementation for canonical node types.

```
const DSL_Dmatrix& GetCiWeights() const;
```

Returns the reference to `DSL_Dmatrix` object containing conditionally independent probabilities of the node.

```
int SetCiWeights(const DSL_Dmatrix& weights);
int SetCiWeights(const DSL_doubleArray& weights);
```

Set the conditionally independent probabilities. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
const DSL_Dmatrix* GetTemporalCiWeights(int order) const;
```

For a plate node in a DBN, returns the pointer to DSL_Dmatrix object containing conditionally independent probabilities of the given temporal order. Returns NULL on failure.

```
int SetTemporalCiWeights(int order, const DSL_Dmatrix& weights);
```

For a plate node in a DBN, sets the conditionally independent probabilities of the given temporal order. Returns DSL_OKAY on success, or a negative error code on failure.

8.6.9 DSL_noisyMAX

Header file: **defnoisymax.h**

```
class DSL_noisyMAX : public DSL_ciDef
```

DSL_noisyMax derives from DSL_ciDef [153] and implements a noisy-MAX parametrization. In addition to conditionally independent probabilities defined in DSL_ciDef, DSL_noisyMAX includes parent outcome strengths for each parent. The parent outcome strengths (by default set to 0, 1, 2, ...) can be used to reorder parent outcomes (as seen from the child node) without any modification in the parent node itself.

```
virtual int GetType() const;
```

Overridden method from DSL_nodeDef [141]. Returns DSL_NOISY_MAX.

```
virtual const char* GetTypeName() const;
```

Overridden method from DSL_nodeDef [141]. Returns "NOISY_MAX".

```
int CheckCiWeightsConsistency(const DSL_Dmatrix &weightsToTest,
    std::string &errMsg, int& errRow, int& errCol) const;
```

Checks the consistency of a specified weightsToTest. Returns DSL_OKAY on success, or a negative error code on failure. On failure errMsg, errRow and errCol are set.

```
const DSL_intArray& GetParentOutcomeStrengths(int parentIndex) const;
```

Gets the reference to the DSL_intArray object representing the outcome strengths for the specified parent. Note that the validity of parentIndex is not checked.

```
int SetParentOutcomeStrengths(int parentIndex, const DSL_intArray &newStrengths);
```

Sets the outcome strengths for the specified parent. Returns DSL_OKAY on success, or a negative error code on failure.

```
int GetTemporalParentOutcomeStrengths(int order,
    std::vector<DSL_intArray>& strengths) const;
```

For a plate node in a DBN, gets outcome strengths for all parents for a specified temporal order. Returns DSL_OKAY on success, or a negative error code on failure.

```
int SetTemporalParentOutcomeStrengths(int order,
    const std::vector<DSL_intArray>& strengths);
```

For a plate node in a DBN, sets outcome strengths for all parents for a specified temporal order. Returns DSL_OKAY on success, or a negative error code on failure.

8.6.10 DSL_noisyAdder

Header file: **defnoisyadder.h**

```
class DSL_noisyAdder : public DSL_ciDef
```

DSL_noisyAdder derives from DSL_ciDef [153] and implements the Noisy-Adder parametrization. In addition to conditionally independent probabilities from DSL_ciDef, DSL_noisyAdder defines the distinguished outcome for itself and for each parent node. By default, each distinguished outcome is the last one. Each of the parents is also associated with the numeric weight, which is equal to 1 by default. The parametrization is converted to CPT on demand (see DSL_lazyDef [151]) using one of two available algorithms, defined in the following class member enum:

```
enum Function { fun_average, fun_single_fault };
```

The average-type Noisy-Adder node calculates the CPT from the parameters by taking the average of probabilities of the effect given each of the causes in separation. The single fault type Noisy-Adder node assumes that only one of the parent nodes will be in the non-distinguished state.

```
virtual int GetType() const;
```

Overridden method from DSL_nodeDef [141]. Returns DSL_NOISY_ADDER.

```
virtual const char* GetTypeName() const;
```

Overridden method from DSL_nodeDef [141]. Returns "NOISY_ADDER".

```
Function GetFunction() const;
```

Gets the adder function.

```
int SetFunction(Function val);
```

Sets the adder function.

```
int GetDistinguishedState() const;
```

Returns the node's distinguished state.

```
int SetDistinguishedState(int thisState);
```

Sets the node's distinguished state.

```
int GetParentDistinguishedState(int parentIndex);
```

Returns the distinguished state for the parent specified by `parentIndex`, or a negative error code if `parentIndex` is invalid.

```
int SetParentDistinguishedState(int parentIndex, int newDState);
```

Sets the distinguished state to `newDState` for the parent specified by `parentIndex`. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
double GetParentWeight(int parentIndex) const;
```

Returns the parent or leak weight. The leak weight is returned when `parentIndex` is is equal to the number of parents.

```
int SetParentWeight(int parentIndex, double value);
```

Sets the parent weight or, when `parentIndex` is equal to the number of parents, the leak weight. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int GetTemporalParentInfo(int order,
    DSL_doubleArray &weights, DSL_intArray &distStates) const;
```

For a plate node in a DBN, gets parent weights and distinguished states for a specified temporal `order`. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int SetTemporalParentInfo(int order,
    const DSL_doubleArray &weights, const DSL_intArray &distStates);
```

For a plate node in a DBN, sets parent weights and distinguished states for a specified temporal `order`. Returns `DSL_OKAY` on success, or a negative error code on failure.

8.6.11 DSL_decision

Header file: **`defdecision.h`**

```
class DSL_decision : public DSL_discDef
```

`DSL_decision` provides implementation for decision nodes (used in influence diagrams), inheriting most of the functionality from `DSL_discDef` 145. Decision nodes do not have numeric parameters, but their outcomes (choices) can be associated with intervals or point values.

```
virtual int GetType() const;
```

Overridden method from `DSL_nodeDef` [141]. Returns `DSL_LIST`.

```
virtual const char* GetTypeName() const;
```

Overridden method from `DSL_nodeDef` [141]. Returns `"LIST"`.

8.6.12 DSL_utility

Header file: **`defutility.h`**

```
class DSL_utility : public DSL_nodeDef
```

`DSL_utility` provides implementation for utility nodes (used in influence diagrams). The utilities are stored in `DSL_Dmatrix` member variable.

```
virtual int GetType() const;
```

Overridden method from `DSL_nodeDef` [141]. Returns `DSL_TABLE`.

```
virtual const char* GetTypeName() const;
```

Overridden method from `DSL_nodeDef` [141]. Returns `"TABLE"`.

```
const DSL_Dmatrix& GetUtilities() const;
```

Returns a reference to the utility matrix. Note that you do not need to cast `DSL_nodeDef` pointer obtained from `DSL_node::Def` to `DSL_utility` in order to get the utilities. The methods `GetMatrix` and `GetDefinition` (declared in `DSL_nodeDef`) will return a pointer to the same `DSL_Dmatrix` object. `GetUtilities` is defined to provide a method with a name closely reflecting the influence diagram terminology.

```
int Normalize();
```

Scales utilities to 0..1 range. If all utilities are identical, they are set to one.

```
double GetMinimumUtility() const;
double GetMaximumUtility() const;
```

Return minimum and maximum utility, respectively.

8.6.13 DSL_mau

Header file: **`defmau.h`**

DSL_mau : public DSL_nodeDef

`DSL_mau` provides implementation for multi-attribute utility (MAU) nodes. By default, the representation uses a numeric weight for each utility parent (the additive linear utility, or ALU, model). It is also possible to specify the multi-attribute utility function as deterministic expressions. See the *Equations* [197] section for list of functions which can be combined into MAU expressions.

---

virtual int GetType() const;

Overridden method from `DSL_nodeDef` [141]. Returns `DSL_MAU`.

---

virtual const char* GetTypeName() const;

Overridden method from `DSL_nodeDef` [141]. Returns `"MAU"`.

---

const DSL_Dmatrix& GetWeights() const;

Returns a reference to the weights matrix. Note that you do not need to cast `DSL_nodeDef` pointer obtained from `DSL_node::Def` to `DSL_mau` in order to get the utilities. The methods `GetMatrix` and `GetDefinition` (declared in `DSL_nodeDef`) will return a pointer to the same `DSL_Dmatrix` object. `GetWeights` is defined to provide a method with a name closely reflecting the influence diagram terminology.

---

int ConvertToExpressions();

Converts the ALU representation to general MAU, with each expression derived from the existing ALU weight. Returns `DSL_OKAY`, or a negative error code when the node is already a general MAU.

---

int GetExpressions(std::vector<std::string> &expr) const;

Fills the `expr` with MAU expressions and returns `DSL_OKAY`. If node is ALU, the `expr` output vector is not modified, and function returns `DSL_WRONG_NODE_TYPE`.

---

int SetExpressions(const std::vector<std::string> &expr, std::string *errMsg=NULL);

Sets the MAU expressions. If `errMsg` is not NULL, it receives human-readable message on error. To remove expressions and change node representation to ALU, pass empty `expr` vector. Returns `DSL_OKAY` on success, or a negative error code on failure.

---

bool ValidateExpression(const std::string &str, std::string &errMsg, int *errPos=NULL);

Validates single expression for use with MAU node. Returns `true` on success. If validation fails, returns `false` and `errMsg` is set to human-readable error message. If `errPos` is not `NULL`, the index of the first invalid expression is returned in this output parameter.

8.6.14 DSL_equation

Header file: **defequation.h**

```
class DSL_equation : public DSL_nodeDef
```

`DSL_equation` provides implementation for nodes defined by equations (both chance and deterministic). The equation is specified as a string, validated, parsed, and stored in the `DSL_generalEquation` [174] member variable. Equation nodes can be unbounded, bounded or semi-bounded. Equation nodes also define discretization intervals, which are used by hybrid inference algorithms.

```
virtual int GetType() const;
```

Overridden method from `DSL_nodeDef` [141]. Returns `DSL_EQUATION`.

```
virtual const char* GetTypeName() const;
```

Overridden method from `DSL_nodeDef` [141]. Returns `"EQUATION"`.

```
const DSL_generalEquation& GetEquation() const;
```

Returns a reference to the `DSL_generalEquation` [174] object, representing node equation in parsed form. The reference is `const`, and cannot be used to modify the equation.

```
int SetEquation(const std::string &eq,
    int *errPos=NULL, std::string *errMsg=NULL);
```

Sets the node equation based on the text passed in `eq` parameter. The equation is parsed and validated. When validation fails and `errPos` or `errMsg` are not `NULL`, the error information is returned in these output parameters. The `errMsg` contains human-readable error message.

The equation should have only the node identifier on the LHS. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
bool ValidateEquation(const std::string &eq,
    std::vector<std::string> &vars,
    std::string &errMsg, int *errPos=NULL,
    bool *isConst = NULL) const;
```

Validates the equation and returns `true` if the equation is valid (i.e., using it in the `SetEquation` call would be successful). Checks for syntax errors and ensures that the referenced variable and function names are valid. The identifiers of the variables in the equation are returned in the `vars` output parameter. If validation fails, `errMsg` contains a human-readable error message. If validation fails and if `errPos` is not NULL, the index of the character in the equation string is returned.

```
void GetBounds(double &low, double &high) const;
```

Gets the domain for the variable represented by the node. If the domain of the node is unbounded or semi-bounded, the infinity is returned in one or both output parameters. Global helper function `DSL_isFinite(double)` returns true if its argument is finite.

```
void SetBounds(double low, double high);
```

Sets the domain for the variable represented by the node. Use plus or minus infinity to represent open bounds. Global helper function `DSL_inf()` returns positive infinity. Use `-DSL_inf()` for negative infinity.

```
bool HasDiscIntervals() const;
```

Returns true when the equation node has defined discretization intervals.

```
const IntervalVector& GetDiscIntervals() const;
```

Returns the reference to the defined intervals (which may be an empty vector if not defined yet). The `IntervalVector` is a `typedef` defined in `DSL_equation` class:

```
typedef std::vector<std::pair<std::string, double> > IntervalVector;
```

First element of the pair represents the interval id, is optional (can be empty), and is used only for display purposes. The second element is the upper bound of the interval. The lower bound is the upper bound of the preceding interval, or the lower bound of the first interval.

```
int GetDiscInterval(int intervalIndex, double &lo, double &hi) const;
```

Writes the edges for the discretization interval specified by `intervalIndex` to `lo` and `hi`. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
void GetIntervalIds(DSL_idArray &ids) const;
```

Returns identifiers of a discretization interval.

```
void GetIntervalEdges(std::vector<double> &edges) const;
void GetIntervalEdges(DSL_doubleArray& edges) const;
void GetIntervalEdges(double* edges) const;
```

Returns the edges of a discretization interval. For N defined intervals, the output vector/array will be resized to N+1 elements. The first and the last edge are the low and the high bound, respectively. For the last overload, the caller must ensure that the buffer specified by the pointer has enough space for N+1 elements.

```
int SetDiscIntervals(const IntervalVector &intervals);
int SetDiscIntervals(double lo, double hi, const IntervalVector &intervals);
```

Set the intervals. The second overload also sets the node bounds. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int ClearDiscIntervals();
```

Removes discretization intervals. Does not affect node bounds. Returns `DSL_OKAY` on success, `DSL_OUT_OF_RANGE` if node has no discretization intervals, or `DSL_WRONG_NODE_TYPE` when node has discretization intervals and is a parent to discrete nodes (and therefore cannot be de-discretized).

```
int Discretize(double x) const;
```

Converts a continuous value to a non-negative integer interval index, based on the specified discretization intervals.

8.7 Node values

8.7.1 DSL_nodeVal

Header file: **nodeval.h**

`DSL_nodeVal` is an abstract base class in node definition class hierarchy. Various (public) virtual `DSL_nodeVal` class methods called internally by `DSL_network` to manage its nodes are not listed in this section, because they should never be called by code that is not part of SMILE library.

Objects of classes derived from `DSL_nodeVal` are created and destroyed by their parent `DSL_node` [138]. Never use `delete` on a node definition pointer.

Use `DSL_node::Val` method to obtain a pointer to node's definition.

```
template <class T> T* As();
template <class T> const T* As() const;
```

Statically cast `this` to the specified type `T`. Note that there is no runtime type checking. `DSL_node::Val()->As<T>` is equivalent to `DSL_node::Val<T>()`.

```
const DSL_network* Network() const;
```

Returns a pointer to a network that contains value's node.

```
int Handle() const;
```

Returns a handle of value's node. The handle does not change during the lifetime of the node value.

```
virtual int GetType() const = 0;
```

Returns the numeric type identifier of the value object.

```
const DSL_intArray& GetIndexingParents() const;
```

Returns the reference to the `DSL_intArray` object with handles of the parents indexing the value. This array is non-empty only for influence diagrams. Indexing parents are are unobserved decision nodes that precede the current node or unobserved chance nodes that should have been observed.

```
int IsValueValid() const;
```

Returns non-zero if node's value is valid.

```
virtual const DSL_Dmatrix* GetMatrix() const;
int GetValue(const DSL_Dmatrix** m) const;
```

Returns the `const` pointer to the `DSL_Dmatrix` object with calculated node value. Call only if `IsValueValid` returns non-zero.

```
void SetValueValid();
void SetValueInvalid();
```

Make the node value valid or invalid. These methods should only be called by inference algorithms.

```
virtual DSL_Dmatrix* GetWriteableMatrix();
```

Returns the pointer to a writable `DSL_Dmatrix` object representing the calculated node value (posterior probabilities or expected utilities). This method should only be called by inference algorithms.

```
virtual int GetMean(double& mean) const;
```

Writes the mean value for the node in the `mean` output parameter. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeVal` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in `DSL_discVal` 166 and `DSL_equationEvaluation` 171.

```
virtual int GetStdDev(double& stddev) const;
```

Writes the standard deviation for the node in the `stddev` output parameter. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeVal` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in `DSL_discVal` 166 and `DSL_equationEvaluation` 171.

```
int IsEvidence() const;
```

Returns non-zero if the node has evidence set.

`int IsRealEvidence() const;`

Returns non-zero if the node has evidence set and the evidence is not a propagated evidence.

`int IsPropagatedEvidence() const;`

Returns non-zero if the node has a propagated evidence set.

`int IsVirtualEvidence() const;`

Returns non-zero if the node has a virtual evidence set.

`virtual int GetEvidence() const;`

Retrieves node's discrete evidence. Returns non-negative outcome index on success, or a negative error code on failure. Default implementation returns `DSL_WRONG_NODE_TYPE`.

Overridden in `DSL_discVal` 166.

`virtual const char* GetEvidenceId() const;`

If the node has discrete evidence set, returns the outcome id of the evidence outcome. Returns `NULL` otherwise. Default implementation returns `NULL`.

Overridden in `DSL_discVal` 166.

`virtual int GetEvidence(double &evidence) const;`

Returns node's continuous evidence in its output `evidence` parameter. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeVal` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in `DSL_discVal` 166 and `DSL_equationEvaluation` 171.

`virtual int SetEvidence(int evidence);`

Sets the evidence as an integer index of node's outcome. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeVal` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in `DSL_discVal` 166 and `DSL_equationEvaluation` 171.

```
virtual int SetEvidence(const char* outcomeId);
```

Sets the evidence as an integer index of node's outcome. The outcome index is determined by comparing the `outcomeId` with node outcome identifiers. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeVal` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in `DSL_discVal` 166.

```
virtual int SetEvidence(double evidence);
```

Sets the evidence as a continuous number. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeVal` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in `DSL_discVal` 166 and `DSL_equationEvaluation` 171.

```
virtual int GetVirtualEvidence(std::vector<double> &evidence) const;
virtual int GetVirtualEvidence(DSL_doubleArray &evidence) const;
```

Retrieves node's virtual evidence. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeVal` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in `DSL_beliefVector` 168.

```
virtual int SetVirtualEvidence(const std::vector<double>& evidence);
virtual int SetVirtualEvidence(const DSL_doubleArray &evidence);
int SetVirtualEvidence(std::initializer_list<double> evidence);
```

Set node's virtual evidence. The last overload creates `DSL_doubleArray` from `std::inititalizer` list and calls the `SetVirtualEvidence(DSL_doubleArray)`. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeVal` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in `DSL_beliefVector` 168.

```
virtual int ClearEvidence();
```

Removes all types of evidence from the node. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeVal` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in `DSL_discVal` 166 and `DSL_equationEvaluation` 171.

```
virtual int ClearPropagatedEvidence();
```

If node is set to propagated evidence, removes the evidence from the node. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeVal` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in DSL_discVal[166] and DSL_equationEvaluation[171].

```
virtual int ClearVirtualEvidence();
```

If node is set to virtual evidence, removes the evidence from the node. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeVal` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in DSL_beliefVector[168].

```
bool HasTemporalEvidence() const;
```

Returns `true` if node has any temporal evidence (regardless of the slice)

```
bool IsTemporalEvidence(int slice) const;
```

Returns `true` if node has a temporal evidence in the specified `slice`.

```
int GetTemporalEvidence(int slice) const;
```

Retrieves node's discrete temporal evidence for the specified `slice`. Returns non-negative outcome index on success, or a negative error code on failure.

```
int GetTemporalEvidence(int slice, std::vector<double> &evidence) const;
int GetTemporalEvidence(int slice, DSL_doubleArray &evidence) const;
```

Retrieves node's virtual temporal evidence for the specified `slice`. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int SetTemporalEvidence(int slice, int evidence);
```

Sets the temporal evidence for the specified `slice` as an integer index of node's outcome. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int SetTemporalEvidence(int slice, const std::vector<double> &evidence);
```

Sets the temporal virtual evidence for the specified `slice`. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int ClearTemporalEvidence(int slice);
```

Clears the temporal evidence for the specified `slice`. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
virtual int GetTemporalMeanStdDev(
    DSL_doubleArray& temporalMean,
    DSL_doubleArray& temporalStdDev) const;
virtual int GetTemporalMeanStdDev(
    std::vector<double>& temporalMean,
    std::vector<double>& temporalStdDev) const;
```

For a plate node in a DBN, retrieve temporal mean and standard deviation. Returns `DSL_OKAY` on success, or a negative error code on failure. The default implementation in `DSL_nodeVal` returns `DSL_WRONG_NODE_TYPE` to indicate that operation is not supported.

Overridden in `DSL_discVal` [166].

8.7.2 DSL_discVal

Header file: **`discval.h`**

```
class DSL_discVal: public DSL_nodeVal
```

`DSL_discVal` class is derived from `DSL_nodeVal` [161]. It is a base class for all node value classes used for various discrete node types.

The value is represented by a `DSL_Dmatrix` member object, which contains posterior probabilities for chance nodes and expected utilities for decision nodes. The matrix can be multi-dimensional if the node is a plate node in the DBN, or if the node is in an influence diagram and has value indexing parents. Otherwise, the matrix is one-dimensional and its size is equal to the number of node outcomes.

`DSL_discVal` supports two types of evidence:

1. discrete evidence, specified as outcome index or outcome id passed to `SetEvidence(int)` or `SetEvidence(const char*)`

2. continuous evidence, which is only valid for discrete nodes with outcome intervals, specified as a numeric value passed to `SetEvidence(double)`.

```
virtual int GetMean(double& mean) const;
```

Overridden method from `DSL_nodeVal` [161]. Writes the mean value of the node to the `mean` output parameter. Returns `DSL_OKAY` on success, or a negative error code on failure. Can succeed only if node has defined outcome intervals or point values.

```
virtual int GetStdDev(double& stddev) const;
```

Overridden method from `DSL_nodeVal` [161]. Writes the standard deviation of the node to the `stddev` output parameter. Returns `DSL_OKAY` on success, or a negative error code on failure. Can succeed only if the node has defined outcome intervals or point values.

```
virtual int ClearEvidence();
```

Overridden method from `DSL_nodeVal` [161]. Removes all types of evidence from the node. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
virtual int GetEvidence() const;
```

Overridden method from `DSL_nodeVal` [161]. If node has evidence set, returns non-negative evidence outcome index. Returns negative error code on failure.

```
virtual const char* GetEvidenceId() const;
```

Overridden method from `DSL_nodeVal` [161]. If the node has evidence set, returns the identifier of the evidence outcome. Returns `NULL` if node has no evidence.

```
virtual int GetEvidence(double &evidence) const;
```

Overridden method from `DSL_nodeVal` [161]. If the node has continuous evidence, the evidence value is written to the output `evidence` parameter. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
virtual int SetEvidence(int evidence);
```

Overridden method from `DSL_nodeVal` [161]. Sets the evidence as an integer index of the node's outcome. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
virtual int SetEvidence(const char* outcomeId);
```

Overridden method from `DSL_nodeVal` [161]. Sets the evidence as an integer index of the node's outcome. The outcome index is determined by comparing the `outcomeId` with the node outcome identifiers. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
virtual int SetEvidence(double evidence);
```

Overridden method from `DSL_nodeVal` [161]. Sets the evidence as a continuous number. Valid only if the node has defined outcome intervals. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
bool IsContinuousEvidence() const;
```

Returns `true` when the node has evidence set and the evidence is continuous. Only nodes with outcome intervals can have continuous evidence.

8.7.3 DSL_beliefVector

Header file: **valbeliefvector.h**

```
class DSL_beliefVector : public DSL_discVal
```

`DSL_beliefVector` class is derived from [DSL_discVal](166). It provides a value implementation for all discrete chance and deterministic nodes.

In addition to discrete and continuous evidence supported by `DSL_discVal`, `DSL_beliefVector` implements the `SetVirtualEvidence` and `GetVirtualEvidence` methods declared in `DSL_nodeVal`. The virtual evidence is specified as a probability distribution over the node outcomes.

`DSL_beliefVector` also provides methods for controlling the value. Controlling means that the value has been set from the outside. SMILE's implementation of controlling the value follows the so called arc-cutting semantics, which means that the incoming arcs of the controlled node become inactive (nothing inside the model influences the node, as its value is set from the outside).

```
virtual int GetType() const;
```

Overridden method from [DSL_nodeVal](161). Returns `DSL_BELIEFVECTOR`.

```
const DSL_Dmatrix& GetBeliefs() const;
```

Returns a reference to the matrix of posterior probabilities. Note that you do not need to cast `DSL_nodeVal` pointer obtained from `DSL_node::Val` to `DSL_beliefVector` in order to get the posteriors. The methods `GetMatrix` and `GetValue` (declared in `DSL_nodeVal`) will return a pointer to the same `DSL_Dmatrix` object. `GetBeliefs` is defined to provide a method with a name closely reflecting the Bayesian network terminology.

The matrix can be multi-dimensional if the node has value indexing parents. Indexing parents are unobserved decision nodes that precede the current node or unobserved chance nodes that should have been observed. The size of the last dimension of the matrix (or its only dimension, if there are no value indexing parents) is always equal to the number of node outcomes.

If the node is a plate node in a DBN, the matrix is two-dimensional. The first dimension size is equal to the number of DBN timeslices, the second dimension size is equal to the number of node outcomes.

Otherwise, the matrix is one-dimensional, and the size of its only dimension is equal to the number of the node outcomes.

```
int ControlValue(int outcomeIndex);
```

Controls node's value by specifying the outcome index. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int ClearControlledValue();
```

Makes node's value non-controlled. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int GetControlledValue() const;
```

Returns the outcome index set by `ControlValue`, or a negative error code if node value was not controlled.

```
int IsControlled() const;
```

Returns non-zero value if the node value is controlled.

```
bool IsControllable() const;
```

Returns `true` if the node value can be controlled. A node is controllable if no evidence is set in any of its descendants.

```
virtual int GetVirtualEvidence(std::vector<double> &evidence) const;
virtual int GetVirtualEvidence(DSL_doubleArray &evidence) const;
```

Overridden method from `DSL_nodeVal` [161]. Retrieves node's virtual evidence. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
virtual int SetVirtualEvidence(const std::vector<double>& evidence);
virtual int SetVirtualEvidence(const DSL_doubleArray &evidence);
```

Overridden method from `DSL_nodeVal` [161]. Set node's virtual evidence. Returns `DSL_OKAY` on success, or a negative error code on failure. Fail when the size of the `evidence` array/vector is not equal to the number of node outcomes.

```
virtual int ClearVirtualEvidence();
```

Overridden method from `DSL_nodeVal` [161]. If the node is set to virtual evidence, removes the evidence from the node. Returns `DSL_OKAY` on success, or a negative error code on failure.

8.7.4 DSL_policyValues

Header file: **`valpolicyvalues.h`**

```
class DSL_policyValues : public DSL_discVal
```

`DSL_policyValues` class is derived from `DSL_discVal` [166]. It provides a value implementation for decision nodes, which have definitions of `DSL_decision` [156] type.

```
virtual int GetType() const;
```

Overridden method from `DSL_nodeVal` [161]. Returns `DSL_POLICYVALUES`.

```
const DSL_Dmatrix& GetPolicyValues() const;
```

Returns a reference to the policy values matrix. Note that you do not need to cast `DSL_nodeVal` pointer obtained from `DSL_node::Val` to `DSL_policyValues` in order to get the posteriors. The methods `GetMatrix` and `GetValue` (declared in `DSL_nodeVal`) will return a pointer to the same `DSL_Dmatrix` object. `GetPolicyValues` is defined to provide a method with a name closely reflecting the Bayesian network terminology.

The matrix can be multi-dimensional if the node has value indexing parents. Indexing parents are are unobserved decision nodes that precede the current node or unobserved chance nodes that should have been observed. The last dimension of the matrix (or its only dimension, if there are no value indexing parents) is always equal to the number of node outcomes.

8.7.5 DSL_expectedUtility

Header file: **`valexpectedutility.h`**

```
class DSL_expectedUtility : public DSL_nodeVal
```

`DSL_expectedUtility` class is derived from derived from `DSL_nodeVal` [161]. It provides a value implementation for utility nodes, which have definitions of `DSL_utility` [157] type.

The value is represented by a `DSL_Dmatrix` member object, which contains expected utilitities.

```
virtual int GetType() const;
```

Overridden method from `DSL_nodeVal` [161]. Returns `DSL_EXPECTEDUTILITY`.

```
const DSL_Dmatrix& GetExpectedUtilities() const;
```

Returns a reference to the matrix of expected utilities. Note that you do not need to cast `DSL_nodeVal` pointer obtained from `DSL_node::Val` to `DSL_expectedUtilities` in order to get the expected utilities. The methods `GetMatrix` and `GetValue` (declared in `DSL_nodeVal`) will return a pointer to the same `DSL_Dmatrix` object. `GetExpectedUtilities` is defined to provide a method with a name closely reflecting the Bayesian network terminology.

The matrix can be multi-dimensional if the node has value indexing parents. Indexing parents are are unobserved decision nodes that precede the current node or unobserved chance nodes that should have been observed. The last dimension of the matrix (or its only dimension, if there are no value indexing parents) is always equal to one.

```
double GetMinimumUtility() const;
double GetMaximumUtility() const;
```

Return minimum or maximum expected utility.

8.7.6 DSL_mauExpectedUtility

Header file: **`valmauexpectedutility.h`**

```
class DSL_mauExpectedUtility : public DSL_nodeVal
```

`DSL_mauExpectedUtility` class is derived from derived from `DSL_nodeVal` 161. It provides a value implementation for utility nodes, which have definitions of `DSL_mau` 157 type.

The value is represented by a `DSL_Dmatrix` member object, which contains expected utilities.

```
virtual int GetType() const;
```

Overridden method from `DSL_nodeVal` 161. Returns `DSL_MAUEXPECTEDUTILITY`.

```
const DSL_Dmatrix& GetExpectedUtilities() const;
```

Returns a reference to the matrix of expected utilities. Note that you do not need to cast `DSL_nodeVal` pointer obtained from `DSL_node::Val` to `DSL_mauExpectedUtilities` in order to get the expected utilities. The methods `GetMatrix` and `GetValue` (declared in `DSL_nodeVal`) will return a pointer to the same `DSL_Dmatrix` object. `GetExpectedUtilities` is defined to provide a method with a name closely reflecting the Bayesian network terminology.

The matrix can be multi-dimensional if the node has value indexing parents. Indexing parents are unobserved decision nodes that precede the current node or unobserved chance nodes that should have been observed. The last dimension of the matrix (or its only dimension, if there are no value indexing parents) is always equal to one.

8.7.7 DSL_equationEvaluation

Header file: **valequationevaluation.h**

```
class DSL_equationEvaluation : public DSL_nodeVal
```

`DSL_equationEvaluation` class is derived from `DSL_nodeVal` 161. `DSL_equationEvaluation` is always associated with equation nodes, which have definitions of `DSL_equation` 159 type.

The value is represented either by samples, or when sampling algorithm could not be invoked given the current network evidence, by discretized beliefs. During the lifetime of `DSL_equationEvaluation` object, its representation can change. Call `IsDiscretized` to determine which representation is used.

The only evidence type that can be set on `DSL_equationEvaluation` is continous evidence.

```
virtual int GetType() const;
```

Overridden method from `DSL_nodeVal` 161. Returns `DSL_EQUATIONEVALUATION`.

```
virtual int GetMean(double& mean) const;
```

Overridden method from `DSL_nodeVal` 161. Writes the mean value of the node to the `mean` output parameter. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
virtual int GetStdDev(double& stddev) const;
```

Overridden method from `DSL_nodeVal` 161. Writes the standard deviation of the node's marginal probability distribution to the `stddev` output parameter. Returns `DSL_OKAY` on success, or a negative error code on failure.

`virtual int GetEvidence(double &evidence) const;`

Overridden method from `DSL_nodeVal` 161. Writes node evidence in its output `evidence` parameter. Returns `DSL_OKAY` on success, or a negative error code on failure.

`virtual int SetEvidence(double evidence);`

Overridden method from `DSL_nodeVal` 161. Sets the evidence as a continuous number. Returns `DSL_OKAY` on success, or a negative error code on failure.

`virtual int ClearEvidence();`

Overridden method from `DSL_nodeVal` 161. Removes the evidence from the node. Returns `DSL_OKAY` on success, or a negative error code on failure.

`virtual int ClearEvidence();`

Overridden method from `DSL_nodeVal` 161. Removes the propagated evidence from the node. Returns `DSL_OKAY` on success, or a negative error code on failure.

`bool IsDiscretized() const;`

Returns `true` if node value is discretized.

`const DSL_Dmatrix& GetDiscBeliefs() const;`

Returns a reference to the discretized beliefs stored in one-dimensional `DSL_Dmatrix`. If the matrix is non-empty, the equation was evaluated over a temporary discrete network and contains no samples. Therefore, all sample-related methods should not be called and the only information about node probabilities is contained in the beliefs vector. `GetMean` and `GetStdDev` will return the mean and standard deviation based on the discretized beliefs and the node's discretization intervals.

`const std::vector<double>& GetSamples() const;`

Returns a reference to the vector of samples generated for the node during stochastic inference. The size of the vector is no larger than a value passed to `DSL_network::SetNumberOfSamples`. If outlier rejection was enabled by `DSL_network::EnableRejectOutlierSamples`, some of the samples may be rejected if their value does not fall inside the bounds specified for the node with `DSL_equation::SetBounds`.

`bool HasSamplesOutOfBounds() const;`

Returns `true` if the node's value has samples out of bounds. If outlier rejection was enabled by `DSL_network::EnableRejectOutlierSamples`, the samples out of bounds are not recorded, and `HasSamplesOutOfBounds` will return `false`.

```
void GetStats(double &mean, double &stddev, double &vmin, double &vmax) const;
```

If node value is sample-based, writes mean, standard deviation, minimum and maximum to its output parameters.

```
int GetHistogram(double lo, double hi, int binCount, std::vector<int> &histogram) const;
```

Calculates the number of samples that fall into evenly spaced bins between `lo` and `hi`. Returns DSL_OKAY, or a negative error code on failure.

```
void SamplingStart(int samplesToReserve=0);
void AddSample(double sample);
int SamplingEnd();
```

Methods invoked by sampling algorithms to record samples.

8.8 DSL_userProperties

Header file: **`general.h`**

`DSL_userProperties` objects are key-value maps. Both key and value are strings. Keys must be unique and valid SMILE identifiers.

```
int GetNumberOfProperties() const;
```

Returns the number of stored properties.

```
const char* GetPropertyName(int index) const;
```

Returns the name (key) of the property.

```
const char* GetPropertyValue(int index) const;
```

Returns the value of the property.

```
int FindProperty(const char* name) const;
```

Returns the index of the property with a specified name, or a negative value if the property was not found.

```
int AddProperty(const char* name, const char* value);
```

Adds a property. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int InsertProperty(int index, const char* name, const char* value);
```

Inserts a property at the specified `index`. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int ChangePropertyName(int index, const char* name);
```

Modifies the property name at the specified `index`. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int ChangePropertyValue(int index, const char* value);
```

Modifies the property name at the specified `index`. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int DeleteProperty(int index);
```

Deletes the property at the specified `index`. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
void Clear();
```

Removes all properties.

8.9 DSL_generalEquation

Header file: **`generalequation.h`**

`DSL_generalEquation` objects are used to store the parsed representation of node equations. Equation nodes manage their `DSL_generalEquation` members. This reference only lists `const` methods.

```
void Write(std::string &s) const;
```

Writes the equation to its output string parameter.

```
bool IsConstant() const;
```

Returns `true` if the equation is a constant. For example, *x=3.14* and *x=Sin(3+0.13)* are constants. *x=a+1* is not constant, even if node *a* is deterministic.

```
bool IsDeterministic() const;
```

Returns `true` if the equation is deterministic (does not contain random generator functions). *x=Normal(0,1)* is not deterministic. *x=a+1* is deterministic, even if node *a* is not deterministic.

8.10 DSL_instanceCounts

Header file: **instancecounts.h**

```
int Calculate(const DSL_network &network, const DSL_dataset &data,
        const std::vector<DSL_datasetMatch> &matching);
```

Calculates the instance counts for `network` and `data` using `matching`. The instance count is calculated for each CPT column of each node in the network based on the number of occurrences in the data set. Returns DSL_OKAY or a negative status code on error.

```
int SetUniform(const DSL_network &net, int uniformCount);
```

Sets uniform instance counts for all nodes in the network. Returns DSL_OKAY or a negative status code on error.

```
int Override(int nodeHandle, int count);
```

Overrides the instance count for all columns in the CPT of the specified node. Returns DSL_OKAY or a negative status code on error.

8.11 DSL_dataset

Header file: **dataset.h**

```
DSL_dataset();
DSL_dataset(const DSL_dataset &src);
DSL_dataset& operator=(const DSL_dataset &src);
~DSL_dataset();
```

Default constructor, copy constructor, assignment operator, and destructor are defined.

```
int ReadFile(const std::string &filename,
    const DSL_datasetParseParams *params = NULL,
    std::string *errOut = NULL);
```

Reads the contents of the data set from the text file. Returns `DSL_OKAY` on success or an error code on failure. If `errOut` is not `NULL`, additional information about the error is returned.

The parser reads the first line from the file and searches for the following separator characters: tab, comma, semicolon, space (in this order). The first character found is considered to be the separator.

The types of data set variables are determined as follows:

- If the data column in the file contains non-numeric entries, the corresponding data set variable is string discrete.
- If the data column in the file contains only numeric entries and there is at least one fractional value, the corresponding data set variable is numeric continuous.
- Otherwise the data set variable is numeric discrete.

To customize parsing, you can pass the pointer to the `DSL_datasetParseParams` struct. The structure is declared in `dataset.h` as:

```
struct DSL_datasetParseParams
{
    DSL_datasetParseParams() :
        missingValueToken("*"),
        missingInt(DSL_MISSING_INT),
        missingFloat(DSL_MISSING_FLOAT),
        columnIdsPresent(true) {}
    std::string missingValueToken;
    int missingInt;
    float missingFloat;
    bool columnIdsPresent;
};
```

```
int WriteFile(const std::string &filename,
    const DSL_datasetWriteParams *params = NULL,
    std::string *errOut = NULL) const;
```

Writes the contents of the data set to a text file. Returns `DSL_OKAY` on success or an error code on failure. If `errOut` is not `NULL`, additional information about the error is returned.

To customize parsing, you can pass the pointer to the `DSL_datasetWriteParams` struct. The structure is declared in `dataset.h` as:

```
struct DSL_datasetWriteParams
{
    DSL_datasetWriteParams() :
        missingValueToken("*"),
        columnIdsPresent(true),
        useStateIndices(false),
        separator('\t'),
        floatFormat("%g") {}
    std::string missingValueToken;
    bool columnIdsPresent;
    bool useStateIndices;
    char separator;
    std::string floatFormat;
};
```

```
int MatchNetwork(const DSL_network &net,
```

```
std::vector<DSL_datasetMatch> &matching,
std::string &errMsg);
```

Attempts to match the contents of the data set to the structure of the network specified as the first argument (typically before parameter learning or network validation). May change the integer indices in the data set to ensure the correct fit with outcome ids in the network nodes, therefore it is a non-const method.

On success, the vector of `DSL_datasetMatch` objects is returned in the `matching` argument and the method returns `DSL_OKAY`. To successfully match the network and the data, at least one node and one data set variable have to have identical identifier, and

- either both the node and the data set variable are continuous, or
- both the node and the data set variable are discrete, and all values in the data set variable can be mapped onto node outcomes

When the network and the data set cannot be matched, an error code is returned and additional human-readable information about the error is writter to `errMsg` parameter.

```
int AddIntVar(const std::string id = std::string(),
    int missingValue = DSL_MISSING_INT);
```

Adds discrete integer variable to the data set. Note that you need to call `DSL_dataset::SetStateNames` later if you want to assign string values to integer indices. Returns `DSL_OKAY` on success or error code on failure.

Multiple variables with empty identifiers are allowed.

```
int AddFloatVar(const std::string id = std::string(),
    float missingValue = DSL_MISSING_FLOAT);
```

Adds continuous, floating point variable to the data set. Returns `DSL_OKAY` on success or error code on failure.

Multiple variables with empty identifiers are allowed.

```
int RemoveVar(int var);
```

Removes a variable from the data set. Returns `DSL_OKAY` on success or error code on failure.

```
void AddEmptyRecord();
```

Appends a record with all values missing.

```
void SetNumberOfRecords(int numRecords);
```

Sets the number of records in the data set. If the new number is greater than the current number, new records will have all values missing.

```
int RemoveRecord(int rec);
```

Removes the specified record from the data set. Returns `DSL_OKAY` on success or error code on failure.

```
int FindVariable(const std::string &id) const;
```

Returns the index of the variable with the specified identifier, or a negative error code on failure.

```
int GetNumberOfVariables() const;
```

Returns the number of variables in the data set.

```
int GetNumberOfRecords() const;
```

Returns the number of records in the data set.

```
int GetInt(int var, int rec) const;
```

Returns an integer data value in the specified variable and record.

```
float GetFloat(int var, int rec) const;
```

Returns a floating data value in the specified variable and record.

```
void SetInt(int var, int rec, int value);
```

Sets an integer data value in the specified variable and record.

```
void SetFloat(int var, int rec, float value);
```

Sets a floating data value in the specified variable and record.

```
void SetMissing(int var, int rec);
```

Marks a data element in the specified variable and record as missing.

```
bool IsMissing(int var, int rec) const;
```

Returns true if the data element in the specified variable and record is missing.

```
int GetMissingInt(int var) const;
```

Returns an integer value representing missing data in the specified discrete variable.

```
float GetMissingFloat(int var) const;
```

Returns a float value representing missing data in the specified continuous variable.

```
bool IsDiscrete(int var) const;
```

Returns true if the specified variable is discrete.

```
enum DiscretizeAlgorithm { Hierarchical, UniformWidth, UniformCount };
int Discretize(int var, DiscretizeAlgorithm alg, int intervals,
    const std::string &statePrefix, std::vector<double> &edges);
int Discretize(int var, DiscretizeAlgorithm alg, int intervals,
    const std::string &statePrefix);
```

Discretizes a data set variable. Returns `DSL_OKAY` on success or error code on failure. The first overload also returns the values of discretization interval edges.

8.12 DSL_dataGenerator

Header file: **`datagenerator.h`**

```
DSL_dataGenerator(DSL_network &net);
```

To create a `DSL_dataGenerator` instance, you need to pass a reference to `DSL_network`, which will be used as a source probability distribution for the data generator.

```
int GenerateData(DSL_dataset &ds);
```

Generate data and store the results in the `DSL_dataset`

```
int GenerateData(const char *filename,
    const DSL_datasetWriteParams *params = NULL);
```

Generate data and write the results to a text file. To fine tune the output format, pass the pointer to the `DSL_datasetWriteParams` object.

```
int GenerateData(DSL_dataGeneratorOutput &out);
```

Generate data and write the results to an abstracted output. In order to use this method, create a class derived from `DSL_dataGeneratorOutput` , which is a pure abstract class declared in `datagenerator.h` header.

```
void SetNumberOfRecords(int numrec);
int GetNumberOfRecords() const;
```

Set/get the number of records to generate.

```
void SetRandSeed(int seed);
int GetRandSeed() const;
```

Set/get the seed used to initialize the random generator. Defaults to zero, which causes the value based on system clock to be used as seed.

```
void SetMissingValuePercent(int perc);
int GetMissingValuePercent() const;
```

Set/get the percentage of missing values. Defaults to zero.

```
void SetBiasSamplesByEvidence(bool bias);
bool GetBiasSamplesByEvidence() const;
```

If set to true, generates a data file from the posterior joint probability distribution (i.e., biased by the observations) rather than from the original joint probability distribution. Defaults to false.

```
int SetSelectedNodes(const std::vector<int> &selection);
const std::vector<int>& GetSelectedNodes() const;
```

Set/get the nodes included in the output from `GenerateData`. By default the selection vector is empty, which means that all nodes will be included.

8.13 DSL_validator

Header file: **`validator.h`**

```
DSL_validator(
    DSL_dataset& ds, const DSL_network &net,
    const std::vector<DSL_datasetMatch> &matching,
    const std::vector<int> *fixedNodes = 0);
```

The constructor requires a reference to a data set, a network and a vector of `DSL_datasetMatch` objects. If `KFold` or `LeaveOneOut` are going to be used, it is also possible to specify which nodes in the network do not change their parameters by passing their handles in the `fixedNodes` argument.

```
int AddClassNode(int classNodeHandle);
```

Adds class node to the internal list of class nodes. Returns `DSL_OKAY` on success or an error code on failure.

```
int Test(DSL_progress *progress = 0);
```

Performs testing using the data set specified in the constructor. The network does not change its parameters during the procedure. Returns `DSL_OKAY` on success or an error code on failure.

The optional argument `progress` can be used to stop the testing by returning false from `DSL_progress::Tick` method, which is called periodically within the main loop of the learning algorithm. In such case, the method returns `DSL_INTERRUPTED`.

```
int KFold(DSL_em &em, int foldCount, int randSeed = 0,
    DSL_progress *progress = 0);
```

Performs K-fold cross-validation using the data set specified in the constructor. Returns `DSL_OKAY` on success or an error code on failure.

The internal parameter learning is performed with the `em` object. The network specified in the constructor does not change its parameters; EM runs on a copy of the network.

The number of folds is specified with the `foldCount` parameter.

The folds are created by randomly splitting records in the data set into subsets. The random generator is initialized with the `randSeed` parameter. The value of this parameter defaults to zero, which causes the value based on system clock to be used as seed.

The optional argument `progress` can be used to stop testing by returning false from `DSL_progress::Tick` method, which is called periodically within the main loop of the learning algorithm. In such case, the method returns `DSL_INTERRUPTED`.

```
int LeaveOneOut(DSL_em &em, DSL_progress *progress = 0);
```

Performs the Leave-One-Out crossvalidation using the data set specified in the constructor. The network does not change its parameters during the procedure. Returns `DSL_OKAY` on success or an error code on failure.

The internal parameter learning is performed with the `em` object. The network specified in the constructor does not change its parameters; EM runs on a copy of the network.

The optional argument `progress` can be used to stop the testing by returning false from `DSL_progress::Tick` method, which is called periodically within the main loop of the learning algorithm. In such case, the method returns `DSL_INTERRUPTED`.

```
int GetPosteriors(int classNodeHandle, int recordIndex,
    std::vector<double> &posteriors) const;
```

Fills the `posteriors` vector with the probabilities calculated for `classNodeHandle` using the record from the data set with the index specified by the `recordIndex`.

Returns `DSL_OKAY` on success or an error code on failure.

```
int GetAccuracy(int classNodeHandle, int outcome, double &acc) const;
```

Gets the accuracy for the specified class node and its outcome. Returns `DSL_OKAY` on success or an error code on failure.

```
int GetConfusionMatrix(int classNodeHandle,
    std::vector<std::vector<int> > &matrix) const;
```

Gets the confusion matrix for the specified class node and its outcome. Returns `DSL_OKAY` on success or an error code on failure.

```
int GetPredictedOutcome(int classNodeHandle, int recordIndex) const;
```

Gets the predicted outcome for the specified class node and record index. Returns `DSL_OKAY` on success or an error code on failure.

```
int GetPredictedNode(int recordIndex) const;
int GetPredictedNodeIndex(int recordIndex) const;
```

Gets the predicted node handle or index for the specifed record index. The node prediction is based on the probabilities of the outcomes in class nodes (essentially, the outcome with the highest probability is chosen). Returns `DSL_OKAY` on success or an error code on failure.

```
int GetFoldIndex(int recordIndex) const;
```

Gets the fold to which the specified data set record belongs. Returns `DSL_OKAY` on success or an error code on failure.

```
void GetResultDataset(DSL_dataset &output) const;
```

Fills the `output` data set with the content of the input data set (specified in the constructor) and calculated class node probabilities and predictied outcomes.

```
int CreateROC(int classNodeHandle, int outcomeIndex,
    std::vector<std::pair<double, double> > &curve,
    std::vector<double> &thresholds, double &areaUnderCurve) const;
```

Creates the ROC curve for the specified class node and its outcome. Returns DSL_OKAY on success or an error code on failure.

```
int CalibrateByBinning(int classNodeHandle, int outcomeIndex, int binCount,
    std::vector<std::pair<double, double> > &curve,
    double &hosmerLemeshTest) const;
int CalibrateByMovingAverage(int classNodeHandle, int outcomeIndex,
    int windowSize, std::vector<std::pair<double, double> > &curve) const;
```

Create the calibration curve for the specified class node and its outcome. Returns DSL_OKAY on success or an error code on failure.

8.14 DSL_progress

Header file: **progress.h**

```
virtual bool Tick(double percComplete=-1, const char *msg=NULL) = 0;
```

`DSL_progress` class declares single pure virtual function that must be overridden in the derived classes. SMILE algorithms supporting `DSL_progress` periodically call the `Tick` method with `percComplete` parameter specifying the percentage (0..100) of work performed so far. If the algorithm cannot predict the number of iterations of its main loop, `-1` is used as the percentage.

The string pointed to by `msg` parameter should be copied if the class derived from `DSL_progress` uses it for display or logging purposes after the `Tick` call is finished.

Returning `false` from `Tick` causes the calling algorithm to quit with `DSL_INTERRUPTED` error code.

8.15 DSL_diagSession

Header file: **diagsession.h**

`DSL_diagSession` uses an `DSL_network` instance passed to its constructor to perform cross-entropy calculation for diagnostic observation nodes. When a diagnostic session is in progress, your program should not modify the underlying `DSL_network`.

```
DSL_diagSession(DSL_network &diagNetwork);
```

Initializes the diagnostic session, instantiates the mandatory observations.

```
int RestartDiagnosis();
```

Restarts the diagnostic session by removing all evidence from the network.

```
void UpdateFaultBeliefs();
```

Performs single inference call to calculate fault beliefs. Fault beliefs are required for `FindMostLikelyFault` to work.

```
int UpdateTestStrengths();
```

Runs the main diagnostic algorithm, which is computationally complex and involves a series of runs of a belief updating algorithm. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int GetSingleFaultAlgorithm() const { return singleFaultAlg; }
int SetSingleFaultAlgorithm(int algId);
```

Gets/sets the diagnostic measure algorithm for single fault diagnosis. The available algorithms are:

- DSL_DIAG_SINGLE_PROB_CHANGE
- DSL_DIAG_SINGLE_NORM_CROSSENTROPY
- DSL_DIAG_SINGLE_CROSSENTROPY

The default algorithm is `DSL_DIAG_SINGLE_PROB_CHANGE`.

```
int GetMultiFaultAlgorithm() const { return singleFaultAlg; }
int SetMultieFaultAlgorithm(int algId);
```

Gets/sets the diagnostic measure algorithm for single fault diagnosis. The available algorithms are:

- DSL_DIAG_MULTI_MAX_PROB_CHANGE
- DSL_DIAG_MULTI_L2_NORMALIZED_DISTANCE
- DSL_DIAG_MULTI_COSINE_DISTANCE
- DSL_DIAG_MULTI_CITYBLOCK_DISTANCE
- DSL_DIAG_MULTI_AVG_L2_CITY_DISTANCE
- DSL_DIAG_MULTI_DEPENDENCE_ATLEAST1
- DSL_DIAG_MULTI_DEPENDENCE_ONLY1
- DSL_DIAG_MULTI_DEPENDENCE_ALL
- DSL_DIAG_MULTI_INDEPENDENCE_ATLEAST1
- DSL_DIAG_MULTI_INDEPENDENCE_ONLY1
- DSL_DIAG_MULTI_MARGINAL1
- DSL_DIAG_MULTI_MARGINAL2

The default algorithm is `DSL_DIAG_MULTI_MAX_PROB_CHANGE`.

```
const std::vector<DSL_diagTestInfo>& GetTestStatistics() const;
```

After a successful completion, `ComputeTestStrengths` returns a reference to an unordered vector of `DSL_diagTestInfo` structures, which contain calculated test statistics. The definition of the `DSL_diagTestInfo` structure is:

```
struct DSL_diagTestInfo
{
    int    test; // node handle
    double entropy; // cross entropy
    double cost; // observation cost
    double strength; // diagnostic value
    int observationPriorStartIndex; // detailed entropy
    int faultPosteriorsStartIndex; // detailed entropy
};
```

The diagnostic value is equal to entropy if cost is not specified for the observation. The detailed value of cross-entropy is available from `GetEntropyDetails`.

`const DSL_intArray& GetUnperformedTests()`

Returns a reference to an array with handles of observation nodes that have not yet been observed.

`const std::vector<DSL_diagFaultState>& GetFaults() const { return faults; }`

Returns a vector of DSL_diagFaultState structures containing information about all fault node/state pairs in the network. The definition of the `DSL_diagFaultState` structure is:

```
struct DSL_diagFaultState
{
    int node; // fault node handle
    int state; // fault outcome index
};
```

The order of the elements in the vector does not change during the diagnostic session. Pursued fault indices are the indices of the elements in this vector.

`int GetPursuedFault() const;`

Returns the index of the pursued fault or a negative error code if there is no pursued fault, or more than one fault is pursued.

`const DSL_intArray& GetPursuedFaults() const;`

Returns a reference to an array containing indices of pursued faults. The array will be empty if there are no pursued faults, or will have one element when a single fault is pursued.

`int SetPursuedFault(int faultIndex);`

Sets the pursued fault to a specified `faultIndex`. Returns `DSL_OKAY` on success, or a negative error code on failure.

`int SetPursuedFaults(const DSL_intArray & faultindices);`

Sets the pursued faults to specified `faultIndices`. Returns `DSL_OKAY` on success, or a negative error code on failure.

`int FindMostLikelyFault();`

Finds the index of most likely fault. The fault probabilities must be calculated by `UpdateFaultBeliefs`. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
int FindFault(int faultNodeHandle, int faultStateIndex) const;
```

Finds the index of the fault specified by `faultNodeHandle` and `faultStateIndex`. Returns `DSL_OKAY` on success, or a negative error code on failure.

```
bool MandatoriesInstantiated() const;
```

Returns `true` if all mandatory observations have been instantiated.

```
bool IsDSepEnabled() const;
void EnableDSep(bool enable);
```

Gets/sets the flag to enable D-separation. When D-separation is enabled, the diagnostic algorithm ranks only those observations that are relevant (in terms of being connected to) to the observed set of evidence nodes. This improves performance on very large networks.

```
bool IsDetailedEntropyEnabled() const;
void EnableDetailedEntropy(bool enable);
```

Gets/sets the flag to keep the detailed entropy information after `ComputeTestStrengths`. See `GetEntropyDetails` for details.

```
const std::vector<double>& GetEntropyDetails();
```

Returns a reference to a vector containing inputs to entropy calculations. The vector is non-empty only if detailed entropy is enabled, and there is a single pursued fault, or all pursued faults are states of the same node. The vector contains interleaved blocks of numbers representing the posterior probabilities of fault node and prior probabilities of observation nodes. To locate the blocks of probabilities, use `observationPriorStartIndex` and `faultPosteriorsStartIndex` members of `DSL_diagTestInfo` structure. The size of the probability block is equal to the number of states of the respective node (pursued fault for fault posteriors and each observation node for observation priors).

8.16 DSL_sensitivity

Header file: **`sensitivity.h`**

`DSL_sensitivity` performs sensitivity calculation on a specified network and stores the results. Given a set of target nodes, the algorithm calculates a complete set of derivatives of the posterior probability distributions over the target nodes over each of the numerical parameters of the Bayesian network.

Any non-target node will have multiple coefficients calculated, one for each combination of target node and its outcome. To represent a node/outcome pair, the following typedef is provided in the `DSL_sensitivity` class.

```
typedef std::pair<int, int> Target;
```

```
void GetTargets(std::vector<Target> &targets) const;
```

Returns information about `targets` in the network. The `Target` `typedef` is defined as

```
typedef std::pair<int, int> Target;
```

The first element of the pair is a target node handle, the second is its outcome index. Any non-target node will have multiple coefficients calculated, one for each combination of a target node and its outcome. Each target node will have all its outcomes represented in the `targets` vector.

```
int Calculate(DSL_network &net, bool relevance=true);
```

Calculates sensitivity coefficients, taking current evidence in `net` into account. If `net` is not an influence diagram, it must have at least one explicit target node. The `relevance` flag controls the relevance split before main coefficient calculation.

Returns `DSL_OKAY` on success, or a negative error code on failure.

```
double GetMaxSensitivity() const;
```

Returns maximum sensitivity over all targets.

```
double GetMaxSensitivity(Target target) const;
```

Returns maximum sensitivity over a specified `target`.

```
double GetMaxSensitivity(int node) const;
```

Returns maximum sensitivity over all targets for a specified `node`.

```
double GetMaxSensitivity(int node, Target target) const;
```

Returns maximum sensitivity over a specified `target` for a specified `node`.

```
void GetMaxSensitivity(int node, DSL_Dmatrix &maxSens) const;
```

For a specified `node`, fills the `maxSens` matrix with maximum sensitivity for each CPT parameter over all targets.

```
const DSL_Dmatrix* GetSensitivity(int node, Target target) const;
```

Returns a pointer to a matrix containing sensitivity value for each CPT parameter of `node` over a specified `target`.

```
void GetCoefficients(int node, Target target, std::vector<const DSL_Dmatrix *> &coeffs) const;
```

For a specified `node` and `target`, returns the calculated sensitivity coefficients. For chance nodes, `coeffs` is a 4-element vector with a, b, c, d. For probabilitites, the target posterior (tp) is a function of given CPT entry (p):

*tp = (a * p + b) / (c * p + d)*

This applies to BNs and IDs alike. For utilities the relationship is linear, the target utility (*tu*) is a function of given utility or ALU weight (*u*):

*tu = a * u + b*

The `coeffs` vector will have only two entries, the values of *a* and *b*.

```
const std::vector<int>& GetIndexingNodes() const;
```

For sensitivity calculated in an influence diagram, the method returns a reference to the vector with handles of indexing parents of the utility node. The combinations of the outcomes of the indexing parents are sensitivity configurations. In order to obtain sensitivity for a specific configuration, its index must be set with `SetCurrentConfig`.

```
int GetNumberOfConfigs() const
```

For sensitivity calculated in an influence diagram, returns a number of configurations (combinations of the utility indexing parents).

```
bool IsConfigPossible(int configIndex) const;
```

For sensitivity calculated in an influence diagram, returns `true` if a sensitivity configuration at specific `configIndex` is possible.

```
int SetCurrentConfig(int configIndex);
```

For sensitivity calculated in an influence diagram, sets the current configuration index.

```
int GetCurrentConfig() const;
```

Returns the current configuration index.

8.17 Learning

8.17.1 DSL_em

Header file: **`em.h`**

```
DSL_em();
```

The default constructor sets equivalent sample size to one, random seed to zero, and parameter randomization to true.

```
int Learn(const DSL_dataset& ds, DSL_network& orig,
    const std::vector<DSL_datasetMatch> &matches,
    double *loglik = NULL, DSL_progress *progress = 0);
int Learn(const DSL_dataset& ds, DSL_network& orig,
    const std::vector<DSL_datasetMatch> &matches,
    const std::vector<int> &fixedNodes,
    double *loglik = NULL, DSL_progress *progress = 0);
```

Learns network parameters by means of the EM algorithm in the specified network using data from the data set. Returns `DSL_OKAY` on success or an error code on failure.

Network nodes and data set variables are matched through the `DSL_datasetMatch` vector specified through `matches` argument. Typically, this vector is obtained by a call to `DSL_dataset::MatchNetwork`, but it can also be created by your program if node and variable identifiers do not match.

The second overload should be used when some nodes' parameters are assumed to be fixed. The handles of these nodes are passed in the `fixedNodes` argument.

The optional argument `loglik` can be used to obtain the log likelihood from the EM algorithm. This value, ranging from minus infinity to zero, is a measure of fit of the model to the data.

The optional argument `progress` can be used to stop learning by returning `false` from `DSL_progress::Tick` method, which is called periodically within the main loop of the learning algorithm. In such a case, the `Learn` method returns `DSL_INTERRUPTED`.

```
void SetRandomizeParameters(bool r);
bool GetRandomizeParameters() const;
```

Sets/gets the value of the parameter randomization flag. If set to `true`, the network parameters will be randomized before entering the main loop of the EM algorithm. Defaults to `true`. If `SetRandomizeParameters(true)` is called, the uniformization is disabled and the equivalent sample size is set to 1.

```
void SetUniformizeParameters(bool u);
bool GetUniformizeParameters() const;
```

Sets/gets the value of the parameter uniformization flag. If set to true, the network parameters will be uniformized before entering the main loop of the EM algorithm. Defaults to `false`. If `SetUniformizeParameters(true)` is called, the uniformization is disabled and equivalent sample size is set to 1.

```
void SetSeed(int seed);
int GetSeed() const;
```

Sets/gets the seed used to initialize the random number generator. Defaults to zero, which causes the value based on the system clock to be used as seed. Calling `SetSeed` does not automatically enable randomization.

```
int SetEquivalentSampleSize(float eqs);
float GetEquivalentSampleSize() const;
```

Sets/gets the equivalent sample size. The equivalent sample size, also known as confidence, can be interpreted as the number of records that the current network parameters are based on. The larger the value, the less weight is assigned to new cases,which gives a mechanism for a gentle refinement of the model numerical parameters. The interpretation of this parameter is obvious when the entire network or its parameters have been learned from data - it should be equal to the number of records in the data file from which they were learned.

Equivalent sample size defaults to 1. Call `SetRandomizeParameters(false)` and `SetUniformizeParameters(false)` if you want to use larger values as equivalent sample sizes. `SetEquivalentSampleSize` fails if either randomization or uniformization is enabled.

8.17.2 DSL_bs

Header file: **bs.h**

```
DSL_bs();
```

The default constructor.

```
int Learn(const DSL_dataset &ds_, DSL_network &net,
    DSL_progress *progress = NULL, DSL_bsEvaluator *eval = NULL,
    double *bestScore = NULL, int *bestIteration = NULL) const;
```

Creates a network structure using the Bayesian Search algorithm, then learns the parameters with EM using the specified data set. Each variable in the data set is represented by a node in the network after learning is complete. Returns `DSL_OKAY` on success or an error code on failure.

The optional argument `progress` can be used to stop learning by returning `false` from `DSL_progress::Tick` method, which is called periodically within the main loop of the learning algorithm. In such case, the `Learn` method returns `DSL_INTERRUPTED`.

The optional argument `eval` may be used to provide an alternative structure evaluator, see `DSL_bsEvaluator` 195 reference for details.

The optional output arguments `bestScore` and `bestIteration` can be used to obtain the score for the network structure selected by the algorithm and the iteration index corresponding to that network structure.

```
int nrIteration;
```

Number of iterations (restarts) to be performed in the main structure learning loop. Each iteration starts with random structure and is refined until convergence. Defaults to 20.

```
int maxParents;
```

Maximum number of parents (in-degree) in the learned network. Defaults to 5.

```
int maxSearchTime;
```

Maximum search time (in seconds) for the structure learning to run. Elapsed time is checked after each iteration is complete. Defaults to zero, which means no time limit.

```
int seed;
```

The seed used to initialize the random generator. Defaults to zero, which causes the value based on system clock to be used as seed.

```
int priorSampleSize;
```

Takes part in the score calculation, representing the inertia of the current parameters when introducing new data. Defaults to 50.

```
double linkProbability;
```

The parameter used when generating a random network at the outset of each of the iterations. It essentially influences the connectivity of the starting network. Defaults to 0.1.

```
double priorLinkProbability;
```

Influences the score, by offering a prior probability over all edges. It comes into the formula in the following way:

*log Posterior score = log marginal likelihood (i.e., the BDeu) + |parents|*log(pll) + (|nodes|-|parents|-1) *log(1-pll))*

Defaults to 0.001.

```
DSL_bkgndKnowledge bkk;
```

Background knowledge used to constrain the network structures created by structure learning algorithm. Empty by default.

8.17.3 DSL_pc

Header file: **pc.h**

```
DSL_pc();
```

The default constructor.

```
int Learn(const DSL_dataset &ds, DSL_pattern &pat,
```

```
    DSL_progress *progress = NULL) const;
```

Based on the specified data set, creates a graph using the PC algorithm and stores the graph edges in the specified `DSL_pattern`. Each variable in the data set is represented by a node in the pattern after learning is complete. Returns `DSL_OKAY` on success or an error code on failure.

The output of the PC algorithm (`DSL_pattern` object) can be converted to `DSL_network` with uniform probability distributions with a call to `DSL_pattern::ToNetwork`.

The optional argument `progress` can be used to stop the learning by returning `false` from `DSL_progress::Tick` method, which is called periodically within the main loop of the learning algorithm. In such a case, the `Learn` method returns `DSL_INTERRUPTED`.

```
int maxAdjacency;
```

Maximum number of neighbors of a node (similar, although not identical, to in-degree of the resulting network). Defaults to 8.

```
int maxSearchTime;
```

Maximum search time (in seconds) for the learning to run. Elapsed time is checked after each iteration is complete. Defaults to zero, meaning no time limit.

```
double significance;
```

Statistical significance threshold (alpha value) used in classical independence tests on which the PC algorithm rests. Defaults to 0.05.

```
DSL_bkgndKnowledge bkk;
```

Background knowledge used to constrain the network structures created by structure learning algorithm. Empty by default.

8.17.4 DSL_tan

Header file: **`tan.h`**

```
DSL_tan();
```

The default constructor.

```
int Learn(DSL_dataset &ds, DSL_network &net,
    DSL_progress *progress = NULL,
    double *emLogLik = NULL) const;
```

Creates a network structure using the Tree Augmented Naive Bayes (TAN) algorithm, then learns the parameters with EM from the specified data set. Each variable in the data set is represented by a node in the network after learning is complete. Returns `DSL_OKAY` on success or an error code on failure.

The algorithm produces an acyclic directed graph with the class variable being the parent of all the other (feature) variables and additional connections between the feature variables.

The optional argument `progress` can be used to stop the learning by returning false from `DSL_progress::Tick` method, which is called periodically within the main loop of the learning algorithm. In such a case, the `Learn` method returns `DSL_INTERRUPTED`.

```
std::string classvar;
```

Identifier of the class variable. If the class variable identifier is not specified or there is no variable with the specified identifier in the data set, the `Learn` method fails.

```
int maxSearchTime;
```

Maximum search time (in seconds) for the structure learning to run. Elapsed time is checked after each iteration is complete. Defaults to zero, meaning no time limit.

```
int seed;
```

The seed used to initialize the random generator. Defaults to zero, which causes a value based on the system clock to be used as seed.

8.17.5 DSL_abn

Header file: **`abn.h`**

```
DSL_abn();
```

The default constructor.

```
int Learn(DSL_dataset &ds, DSL_network &net,
    DSL_progress *progress = NULL,
    double *bestScore = NULL, int *bestIteration = NULL,
    double *emLogLik = NULL) const;
```

Creates a network structure using the Augmented Naive Bayes (ABN) algorithm, then learns its parameters with EM from the specified data set. Each variable in the data set is represented by a node in the network after learning is complete. Returns `DSL_OKAY` on success or an error code on failure.

The algorithm produces an acyclic directed graph with the class variable being the parent of all the other (feature) variables and additional connections between the feature variables.

The optional argument `progress` can be used to stop learning by returning `false` from `DSL_progress::Tick` method, which is called periodically within the main loop of the learning algorithm. In such case, the `Learn` method returns `DSL_INTERRUPTED`.

```
std::string classvar;
```

Identifier of the class variable. If the class variable identifier is not specified or there is no variable with the specified identifier in the data set, the `Learn` method fails.

```
bool feature_selection;
```

If `true`, invokes an additional function that removes from the feature set those features that do not contribute enough to the classification.

```
int maxParents;
int maxSearchTime;
int nrIteration;
double linkProbability;
double priorLinkProbability;
int priorSampleSize;
int seed;
```

See the *DSL_bs* 190 reference. The ABN algorithm uses Bayesian Search internally, and can be fine tuned using options of the Bayesian Search algorithm.

8.17.6 DSL_nb

Header file: **nb.h**

```
DSL_nb();
```

The default constructor.

```
int Learn(DSL_dataset ds, DSL_network &net,
    DSL_progress *progress = NULL,
    double *emLogLikelihood = NULL) const;
```

Creates a naive Bayes network, then learns its parameters with EM using the specified data set. Returns `DSL_OKAY` on success or an error code on failure.

The structure of the Naive Bayes network is not learned but rather fixed by assumption: the class variable is the only parent of all remaining, feature variables and there are no other connections between the nodes of the network.

The optional argument `progress` can be used to stop the learning by returning false from `DSL_progress::Tick` method, which is called periodically within the main loop of the learning algorithm. In such case, the `Learn` method returns `DSL_INTERRUPTED`.

```
std::string classVariableId;
```

Identifier of the class variable. If the class variable identifier is not specified or there is no variable with the specified identifier in the data set, the `Learn` method fails.

8.17.7 DSL_bkgndKnowledge

Header file: **bkgndknowledge.h**

```
typedef std::vector<std::pair<int, int> > IntPairVector;
```

Type defined for convenience.

```
IntPairVector forcedArcs;
```

Arcs that are required to be present in the learned network structure.

```
IntPairVector forbiddenArcs;
```

Arcs that are forbidden in the learned network structure.

```
IntPairVector tiers;
```

Enforces temporal tiers in the network: there will be no arcs from nodes in higher tiers to nodes in lower tiers. Each element of the vector contains node handle in `pair::first` and the index of the tier in the `pair::second` member.

8.17.8 DSL_bsEvaluator

Header file: **bs.h**

```
virtual int Evaluate(int iteration, double bsScore, double bestScore,
    DSL_network &net, const DSL_dataset &ds,
    const std::vector<DSL_datasetMatch> &matching,
    DSL_progress *progress,
    double &outputScore) = 0;
```

`DSL_bsEvaluator` class declares a single pure virtual function that must be overridden in the derived classes. The `Evaluate` method should return an `outputScore` for the `net` using the data set `ds` (the learning data set passed as input to `DSL_bs::Learn`). A possible implementation of `Evaluate` can calculate the prediction accuracy using K-fold cross-validation and return the calculated accuracy as the output score. However, any algorithm can be used with any additional data available. The additional data should be stored as members of the class derived from `DSL_bsEvaluator`.

8.17.9 DSL_pattern

Header file: **`pattern.h`**

```
enum EdgeType {None,Undirected,Directed};
```

Edge type identifiers.

```
int GetSize() const;
void SetSize(int size);
```

Gets/sets the number of nodes in the pattern.

```
EdgeType GetEdge(int from, int to) const;
```

Returns the edge type between `from` and `to`, where `from` and `to` are zero-based indices of the nodes in the pattern. If there is no edge, `EdgeType::None` is returned.

```
void SetEdge(int from, int to, EdgeType type);
```

Sets the edge type between `from` and `to`, where `from` and `to` are zero-based indices of the nodes in the pattern. To remove the edge, set the `type` parameter to `EdgeType::None`.

```
bool HasDirectedPath(int from, int to) const;
```

Returns `true` if there is a directed path between `from` and `to`, where `from` and `to` are zero-based indices of the nodes in the pattern.

```
bool HasCycle() const;
```

Returns `true` if the pattern contains a cycle.

```
bool IsDAG() const;
```

Returns `true` if the pattern is an acyclic directed graph - there are no cycles and all edges are directed.

```
bool ToDAG();
```

Attempts the conversion of pattern to an acyclic directed graph. Returns `true` if successful.

```
bool ToNetwork(const DSL_dataset &ds, DSL_network &net);
```

Attempts the conversion of pattern to an acyclic directed graph. If successful, creates `DSL_network` with node identifiers and outcomes based on the specified data set. It is assumed that the data set was used previously to obtain this pattern, therefore the order of variables in the data set corresponds to the order of nodes in the pattern (and in the resulting network). Returns `true` if successful.

```
bool HasIncomingEdge(int to) const;
bool HasOutgoingEdge(int from) const;
```

Return `true` if there is an edge incoming or outgoing to/from the specified node.

```
void GetAdjacentNodes(const int node, std::vector<int>& adj) const;
void GetParents(const int node, std::vector<int>& par) const;
void GetChildren(const int node, std::vector<int>& child) const;
```

Return the adjacent/parent/children nodes of the specified node.

8.18 Equations

The subsections of this chapter describe the notation used when specifying the definition for equation nodes and expression-based MAU nodes.

8.18.1 Operators

Arithmetic operators

+ addition, e.g., if x=3 and y=2, x+y produces 5

- subtraction and unary minus, e.g., if x=3 and y=2, x-y produces 1 and -x produces -3

^ exponentiation (a^b means a^b), e.g., if x=3 and y=2, x^y produces 9

***** multiplication, e.g., if x=3 and y=2, x*y produces 6

/ division, e.g., if x=3 and y=2, x/y produces 1.5

Comparison operators

> greater than, e.g., if x=3 and y=2, x>y produces 1

< smaller than, e.g., if x=3 and y=2, x<y produces 0

>= greater or equal than, e.g., if x=3 and y=2, x>=y produces 1

<= smaller or equal than, e.g., if x=3 and y=2, x<=y produces 0

<> not equal, e.g., if x=3 and y=2, x<>y produces 1

= equal, e.g., if x=3 and y=2, x=y produces 0

Conditional selection operator

? : ternary conditional operator like in C, C++ or Java programming languages.

This operator is essentially a shortcut to the `If` function. For example, `a=b?5:3` is equivalent to `If(a=b, 5, 3)`.

Order of calculation, operator precedence, and parentheses

Expressions are evaluated from left to right, according to the precedence order specified below (1 denotes the highest precedence).

Precedence order	Operator
1	**-** (unary minus)
2	**^** (exponentiation)
3	***** and **/** (multiplicative operators)
4	**+** and **-** (additive operators)
5	**>**, **<**, **>=**, **<=**, **=** (comparison operators)
6	**?:** (conditional selection)

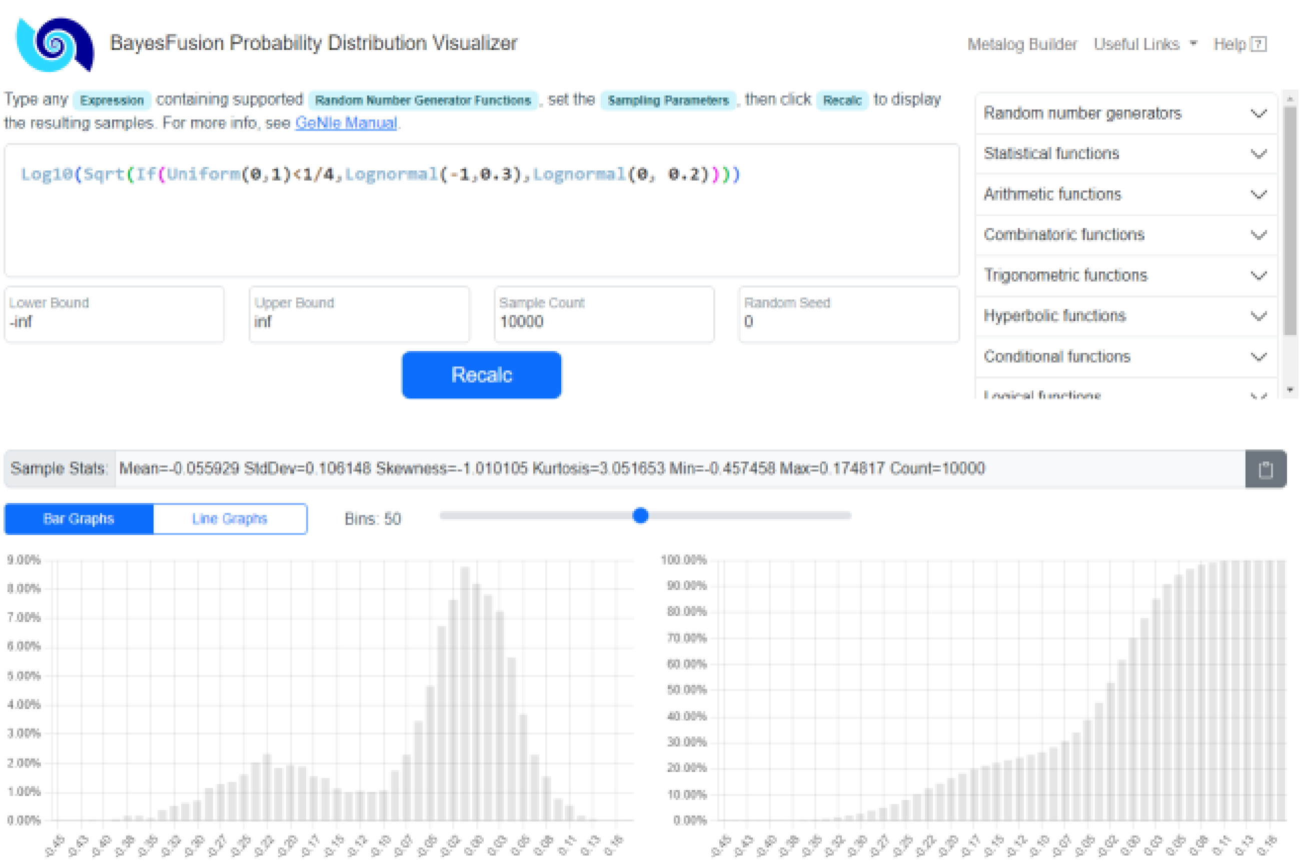
To change the order of calculation, enclose in parentheses those parts of the formula that should be calculated first. This can be done recursively, i.e., parentheses can be nested indefinitely. For example, if x=3 and y=2, 2*y+x/3-y+1 produces 4, 2*(y+x)/(3-y)+1 produces 11, and 2*((y+x)/(3-y)+1) produces 12.

8.18.2 Random Number Generators

Random number generators **each generate a single sample** from the distributions defined below. In most equations, they can be imagined as random noise that distorts the equation. Because the fundamental algorithm for inference in continuous and hybrid models is stochastic simulation, it is possible to visualize what probability distributions these single samples result in for each of the variables in the model.

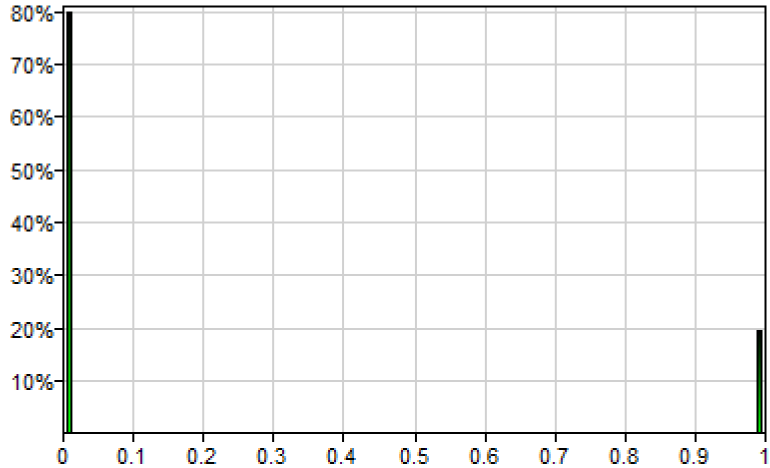
Caution: Probability distributions are not allowed in expression-based MAU nodes, as these are deterministic functions by definition.

Choosing the right probability distribution over continuous data is a skill that requires some statistical insight. When the distribution is transformed by an equation expression, the task is daunting even for an experienced decision analyst. GeNIe offers an interactive tool for visualizing expressions, the Distribution Visualizer, with probability distributions through Monte Carlo simulation. The same functionality is available online at https://prob.bayesfusion.com. The screenshot below shows the online visualizer with the probability distribution sampled from the expression `Log10(Sqrt(If(Uniform(0,1)<1/4,Lognormal(-1,0.3),Lognormal(0, 0.2))))`.



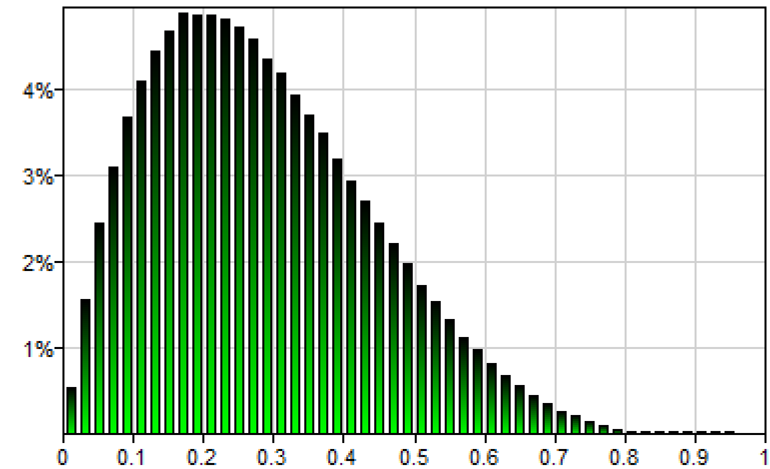
Bernoulli(p)

Bernoulli is a discrete distribution that generates 0 with probability 1-*p* and 1 with probability *p*. `Bernoulli(0.2)` will generate a single sample (0 or 1) from the following distribution, i.e., 1 with probability 0.2 and 0 with probability 0.8:



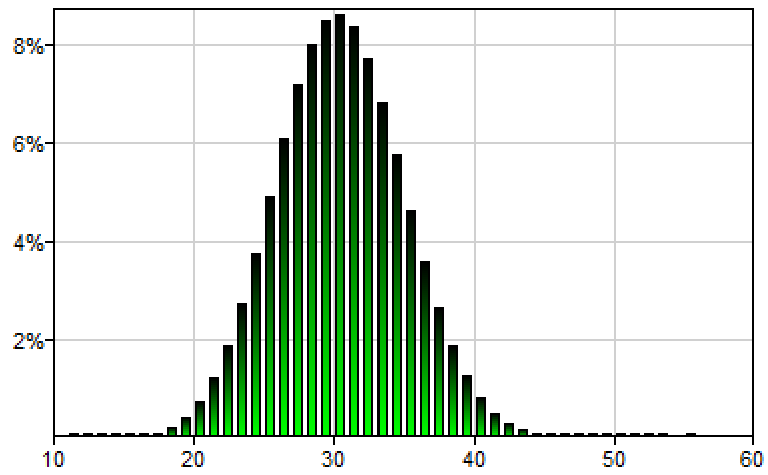
Beta(a,b)

The Beta distribution is a family of continuous probability distributions defined on the interval [0, 1] and parametrized by two positive shape parameters, *a* and *b* (typically denoted by α and β), that control the shape of the distribution. `Beta(2,5)` will generate a single sample from the following distribution:



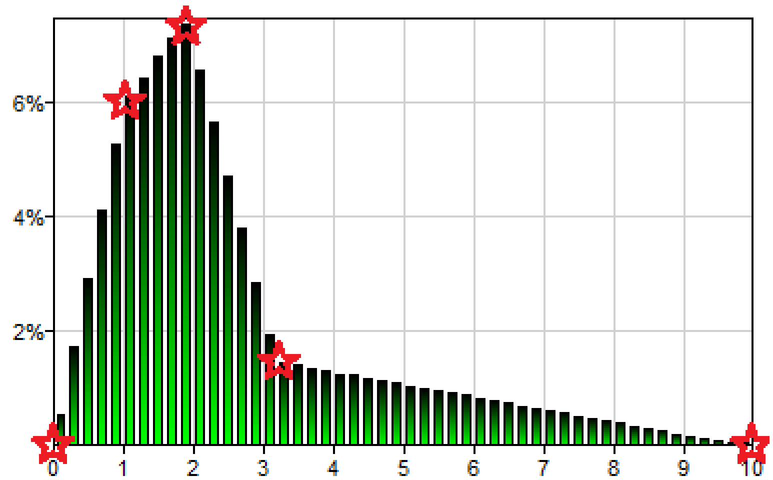
Binomial(n,p)

Binomial is a discrete probability distribution over the number of successes in a sequence of *n* independent trials, each of which yields a success with probability *p*. It will generate a single sample, which will be an integer number between 0 and *n*. A success/failure experiment is also called a Bernoulli trial. Hence, `Binomial(1,p)` is equivalent to `Bernoulli(p)`. `Binomial(100,0.3)` will generate a single sample from the following distribution:



CustomPDF(x1,x2,...y1,y2,...)

The `CustomPDF` distribution allows for specifying a non-parametric continuous probability distribution by means of a series of points on its probability density (PDF) function. Pairs (x_i,y_i) are coordinates of such points. The total number of parameters of *`CustomPDF`* function should thus to be even. Please note that x coordinates should be listed in increasing order. The PDF function specified does not need to be normalized, i.e., the area under the curve does not need to add up to 1.0. For example, *`CustomPDF(0,1.02,1.9,3.2,10,0,4,5,1,0)`* generates a single sample from the following distribution:

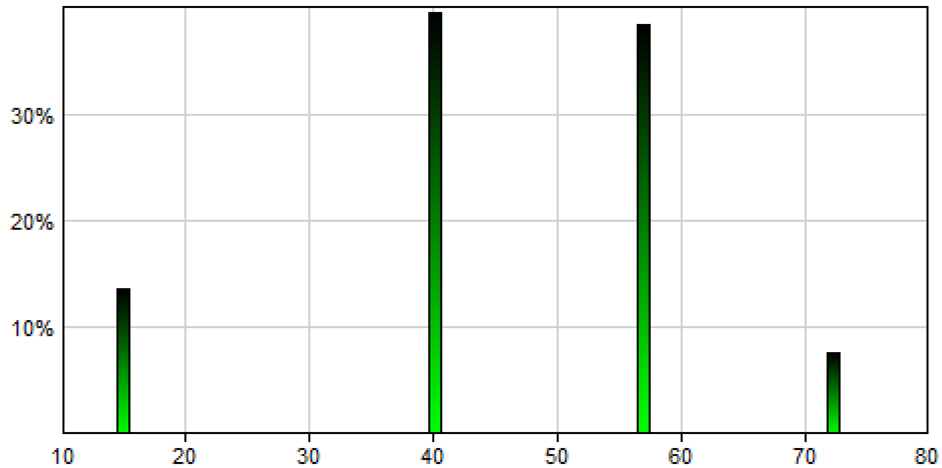


Stars on the plot mark the points defined by the *`CustomPDF`* arguments, i.e., (0, 0), (1.02, 4), (1.9, 5), (3.2, 1), and (10, 0).

Discrete(x1,x2,...,xn, p1,p2,...,pn)

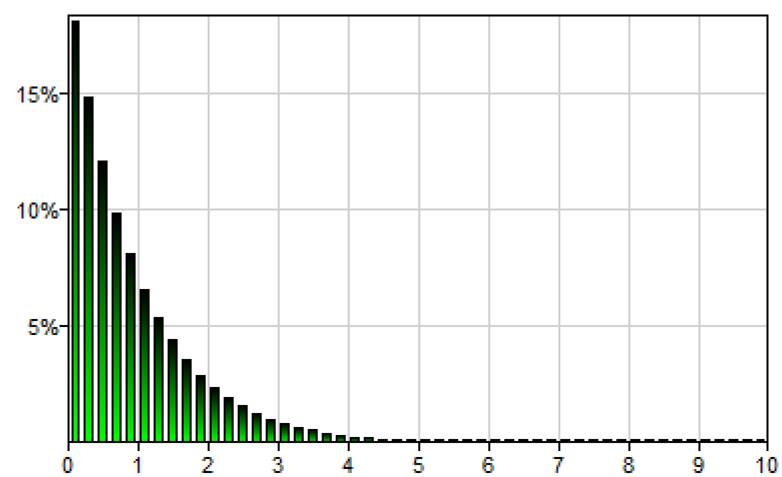
The Discrete distribution allows for specifying a discrete probability distribution over a collection of numerical values. It is one of the simplest random number generators, essentially replicating a discrete distribution and producing values x1, x2, ..., xn with probabilities p1, p2, ..., xn. This distribution is useful in simulating a discrete node using an equation node. The total number of parameters of *`Discrete()`* function should be even. Please note that x values should be listed in increasing order. Even though the p values should in theory add up to 1.0

and we advise that they do, GeNIe perform normalization, i.e., modifies them proportionally to add up to 1.0. For example, `Discrete(15,40,57.5,72.5,0.137339,0.397711,0.387697,0.0772532)` replicates the definition of the variable Age in the HeparII model and generates a single sample from the following distribution:



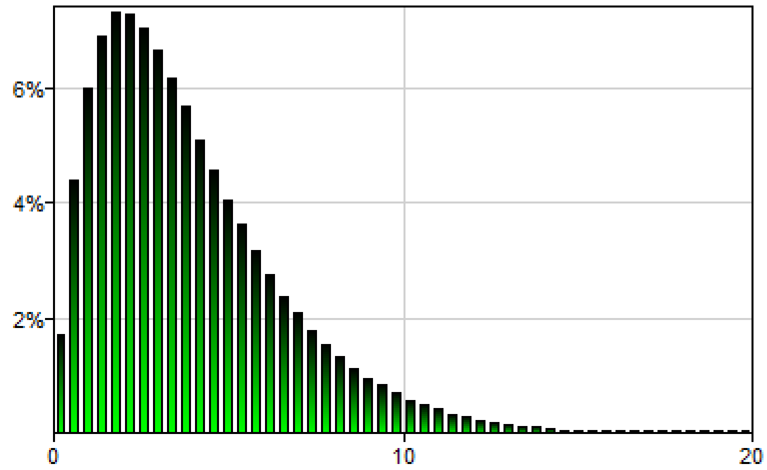
Exponential(lambda)

The exponential distribution is a continuous probability distribution that describes the time between events in a Poisson process, i.e., a process in which events occur continuously and independently at a constant average rate. Its only real-valued, positive parameter *lambda* (typically denoted by λ) determines the shape of the distribution. It is a special case of the Gamma distribution. `Exponential(lambda)` generates a single sample from the domain (0,∞). `Exponential(1)` will generate a single sample from the following distribution:



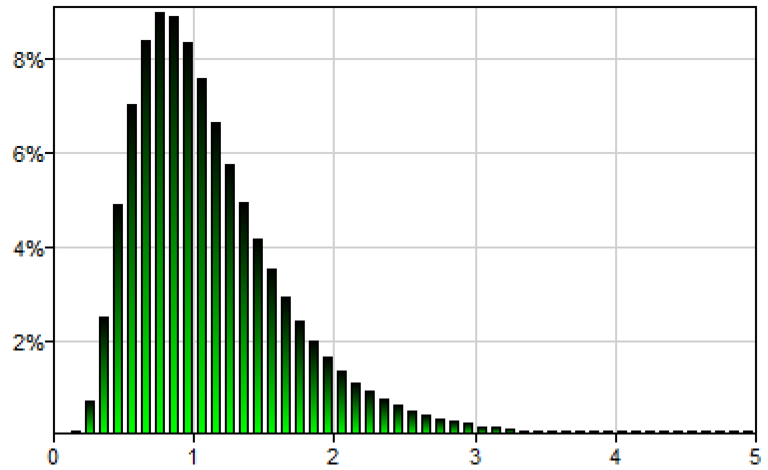
Gamma(shape,scale)

The Gamma distribution is a two-parameter family of continuous probability distributions. There are different parametrizations of the Gamma distribution in common use. SMILE parametrization follows one of the most popular parametrizations, with *shape* (often denoted by *k*) and *scale* (often denoted by θ) parameters, both positive real numbers. `Gamma(2.0,2.0)` will generate a single sample from the following distribution:



Lognormal(mu,sigma)

The lognormal distribution is a continuous probability distribution of a random variable, whose logarithm is normally distributed. Thus, if a random variable X is lognormally distributed, then a variable Y=Ln(X) has a normal distribution. Conversely, if Y has a normal distribution, then $X=e^Y$ has a lognormal distribution. A random variable which is lognormally distributed takes only positive values. `Lognormal(0,0.5)` will generate a single sample from the following distribution:



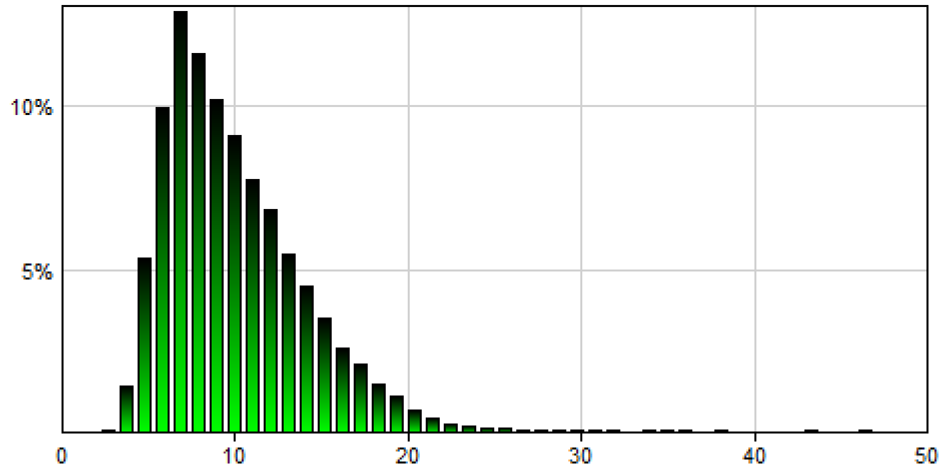
Metalog(lower,upper,k,x1,x2,...,y1,y2,...)

Metalog (also known as the Keelin) distribution is a very flexible distribution, capable of fitting many naturally occurring distributions. It can be specified by probability quantiles, which are values of the variable `xi` and their corresponding cumulative probabilities `yi`. Metalogs are able to represent distributions that are unbounded, semi-bounded, and bounded. lower and upper are the bounds of the distribution (`-Inf()` and `Inf()` denote lower and upper infinite bound respectively). `k` is a parameter of the metalog distribution, running from `2` to n, where n is the number of probability quantiles specified. Generally the higher the value of `k`, the more flexible the distribution but it is worth looking at the distributions generated for different values of `k` to find a compromise between complexity and goodness of fit. The choice of `k` is best performed interactively, looking at the family of metalog distributions generated from the probability quantiles. GeNIe contains a built-in tool, the

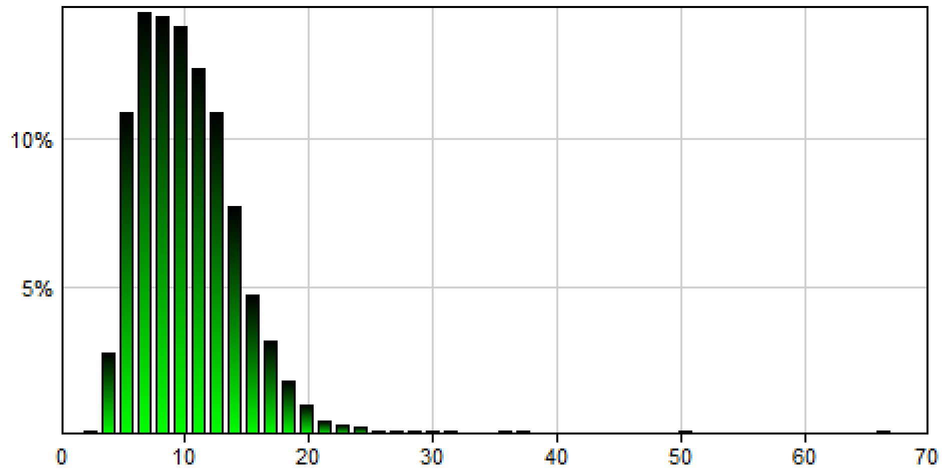
Metalog Builder, which displays metalog PDF and CDF charts. This functionality is also available online at https://metalog.bayesfusion.com.

The examples below use identical bounds and probability quantiles with k equal 4, 6 and 8:

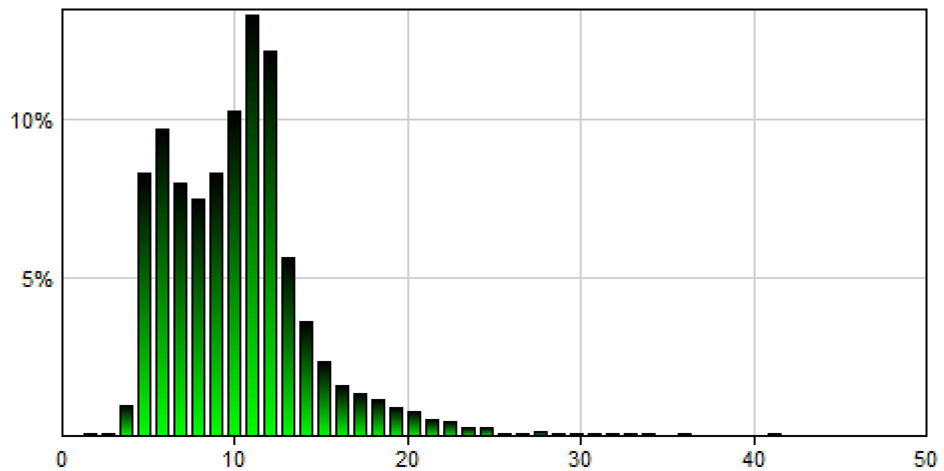
`Metalog(0,Inf(),4,3,4,5,5,7,10,12,15,18,32,0.001,0.01,0.05,0.1,0.25,0.5,0.75,0.9,0.95,0.999)` will generate a single sample from the following distribution:



`Metalog(0,Inf(),6,3,4,5,5,7,10,12,15,18,32,0.001,0.01,0.05,0.1,0.25,0.5,0.75,0.9,0.95,0.999)` will generate a single sample from the following distribution:



`Metalog(0,Inf(),8,3,4,5,5,7,10,12,15,18,32,0.001,0.01,0.05,0.1,0.25,0.5,0.75,0.9,0.95,0.999)` will generate a single sample from the following distribution:

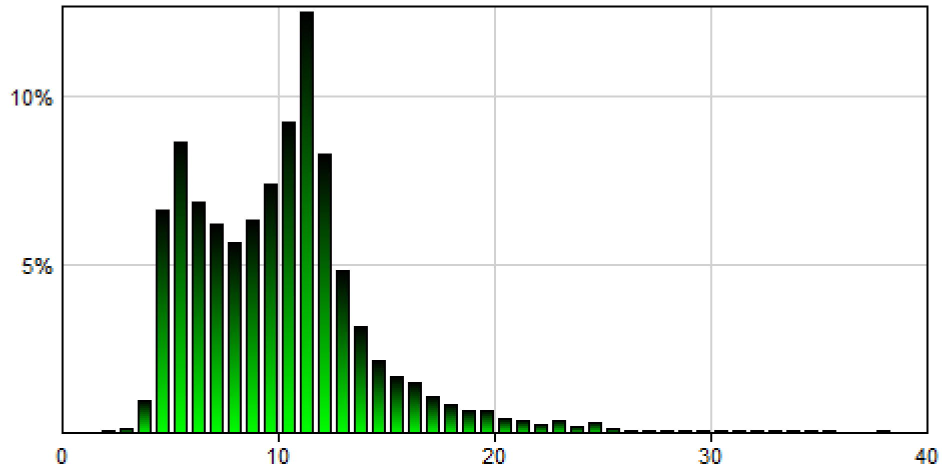


For more information about metalog distributions, please look at the comprehensive article on the topic on Wikipedia (https://en.wikipedia.org/wiki/Metalog_distribution), the Metalog Distribution web site created by Tom Keelin (http://metalogdistributions.com/) or the Metalog Distributions YouTube channel (https://www.youtube.com/channel/UCyHZ5neKhV1mSsedzDBoqyA).

MetalogA(lower,upper,a1,a2,...)

MetalogA function uses what one could call internal metalog coefficients (*ai* and, additionally, the *lower* and *upper* bound of the distribution) that, contrary to percentiles of the distribution used as parameters of *Metalog*, do not have easily interpretable meaning. One might expect that *MetalogA* is more efficient in sample generation, as it skips the whole process of deriving the distribution from which it subsequently generates a sample. However, SMILE has an efficient caching scheme that makes *Metalog* equally efficient in practice.

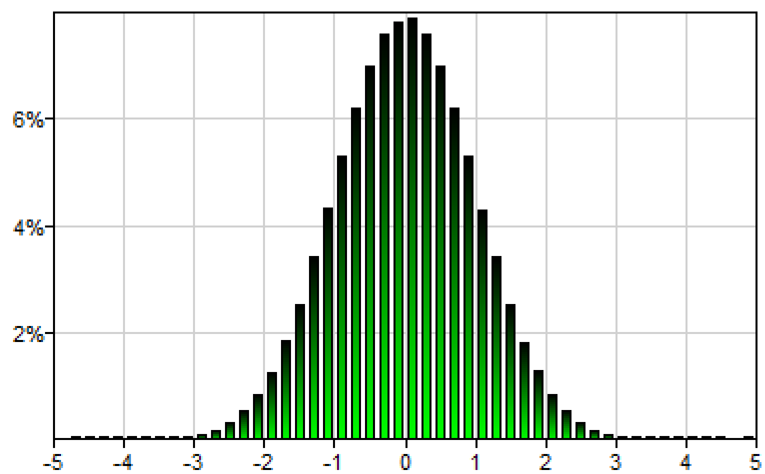
`MetalogA(0,Inf(),2.30769,0.164148,-0.731388,0.343231,0.883249,-0.170727,1.40341,2.64853)`, which is equivalent to `Metalog(0,Inf(),8,3,4,5,5,7,10,12,15,18,32,0.001,0.01,0.05,0.1,0.25,0.5,0.75,0.9,0.95,0.999)`, will generate a single sample from the following distribution:



Please note that the number of ai parameters of `MetalogA` is the same as the `k` parameter in *Metalog*. Obtaining the parameters *ai* outside of tools like Metalog Builder is rather challenging.

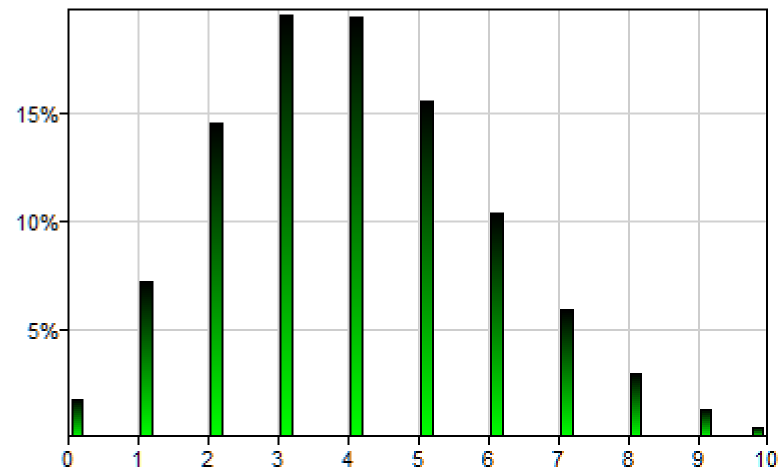
Normal(mu,sigma)

Normal (also known as Gaussian) distribution is the most commonly occurring continuous probability distribution. It is symmetric and defined over the real domain. Its two parameters, *mu* (mean, μ) and *sigma* (standard deviation, σ), control the position of its mode and its spread respectively. `Normal(0,1)` will generate a single sample from the following distribution:



Poisson(lambda)

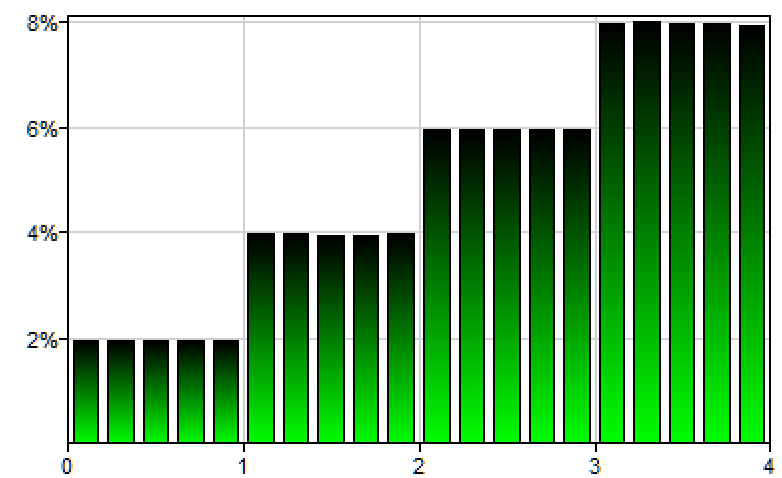
Poisson distribution is a discrete probability distribution typically used to express the probability of a given number of events occurring in a fixed interval of time or space if these events occur with a known constant rate and independently of the time since the last event. Its only parameter, lambda, is the expected number of occurrences (which does not need to be integer). *Poisson(4)* will generate a single sample from the following distribution:



Steps(x1,x2,...y1,y2,...)

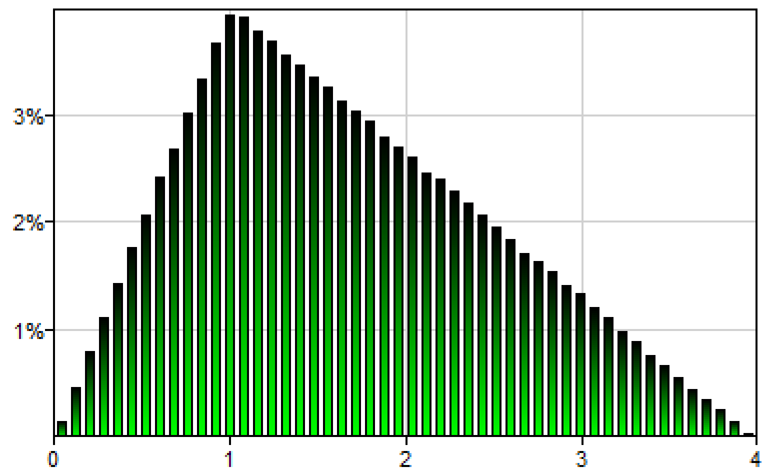
The *Steps* distribution allows for specifying a non-parametric continuous probability distribution by means of a series of steps on its probability density (PDF) function. It is similar to the *CustomPDF* function, although it does not specify the inflection points but rather intervals and the height of a step-wise probability distribution in each of the intervals. Because the number of interval borders is always one more than the number of intervals between them, the total number of parameters of *Steps* function should be odd. Please note that x coordinates should be listed in increasing order. The PDF function specified does not need to be normalized, i.e., the area under the curve does not need to add up to 1.0.

Example: *Steps(0,1,2,3,4,1,2,3,4)* generates a single sample from the following distribution::



Triangular(min,mod,max)

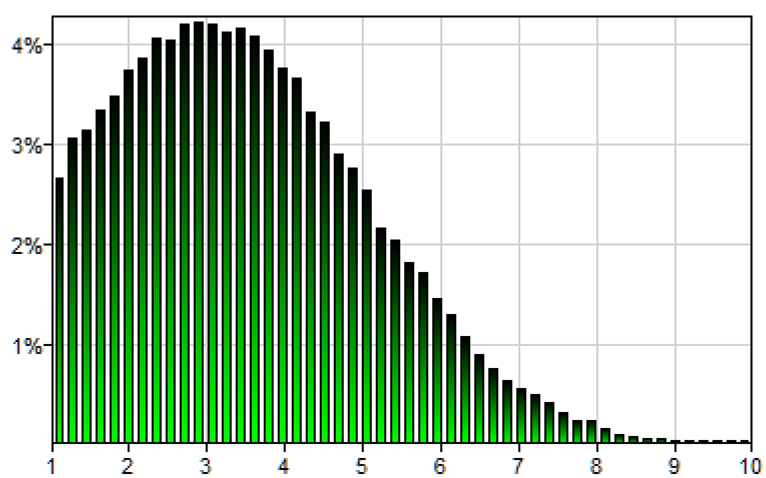
Triangular distribution is a continuous probability distribution with lower limit *min*, upper limit *max* and mode *mod*, where *min* ≤ *mod* ≤ *max*. `Triangular(0,1,3)` will generate a single sample from the following distribution:



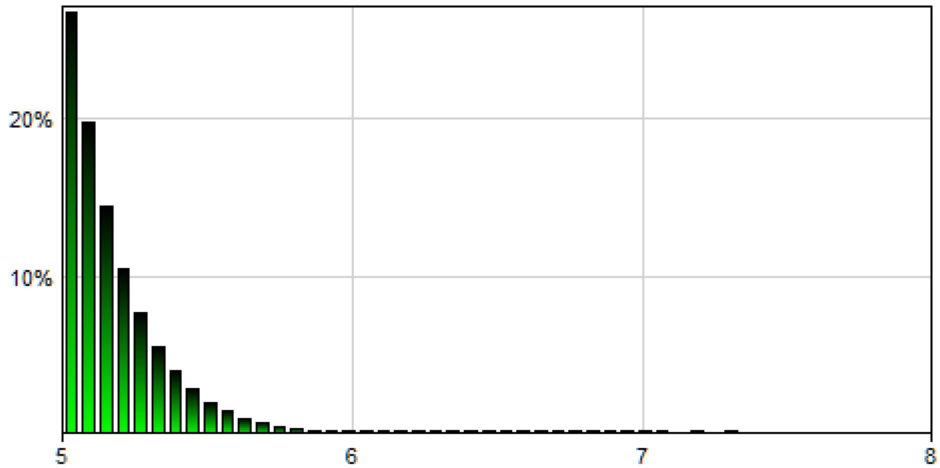
TruncNormal(mu,sigma,lower[,upper])

Truncated Normal distribution is essentially a Normal distribution that is truncated at the values lower and upper. This distribution is especially useful in situation when we want to limit physically impossible values in the model.

`TruncNormal(3,2,1)` will generate a single sample from the following distribution:

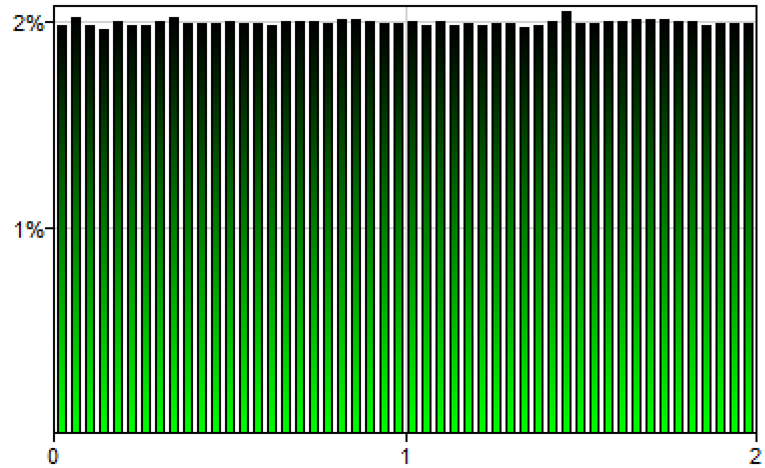


`TruncNormal(0,1,5,10)` will generate a single sample from the following distribution:



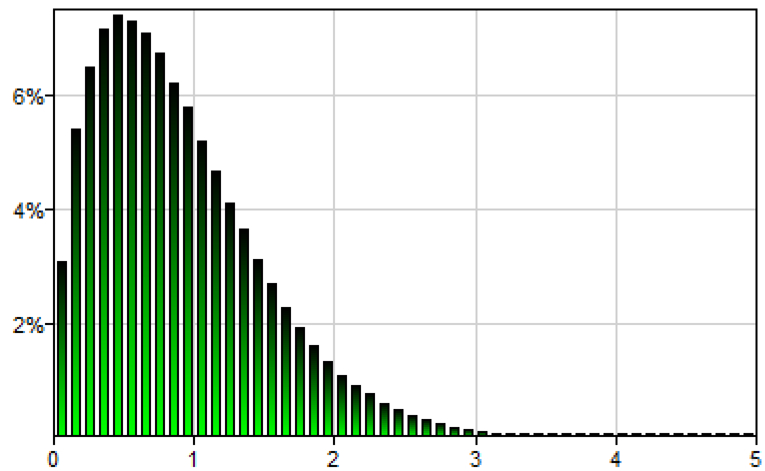
Uniform(a,b)

The continuous uniform distribution, also known as the rectangular distribution, is a family of probability distributions under which any two intervals of the same length are equally probable. It is defined two parameters, *a* and *b*, which are the minimum and the maximum values of the random variable. `Uniform(0,2)` will generate a single sample from the following distribution:



Weibull(lambda,k)

Weibull distribution is a continuous probability distribution named after a Swedish mathematician Waloddi Weibull, used in modeling such phenomena as particle size. It is characterized by two positive real parameters: the scale parameter *lambda* (λ) and the shape parameter *k*. `Weibull(1,1.5)` will generate a single sample from the following distribution:



8.18.3 Statistical Functions

Erf(x)

Returns error function ((also called the Gauss error function, inverse of the Normal PDF), e.g., Erf(1.0) =0.842701.

NormalPDF(x, mu, sigma)

Returns the value of PDF for the normal distribution given by mu and sigma at x.

NormalCDF(x, mu, sigma)

Returns the value of CDF for the normal distribution given by mu and sigma at x.

8.18.4 Arithmetic Functions

Abs(x)

Returns the absolute value of a number, e.g., `Abs(5.3)=Abs(-5.3)=5.3`.

Exp(x)

Returns *e* (Euler's number) raised to the power of *x*, e.g., `Exp(2.2)=`$e^{2.2}$`=9.02501`.

Gammaln(x)

Returns the natural logarithm of the Gamma function ($\Gamma(x)$), e.g., `Gammaln(2.2)=0.0969475`.

GCD(n,k)

Returns the greatest common integer divisor of its two integer arguments *n* and *k*, e.g., `GCD(15,25)=5.` When the arguments are not integers, their fractional part is ignored.

Inf()

Returns positive infinity. For negative infinity, please use `-Inf()`.

LCM(n,k)

Returns the least common integer multiple of its two integer arguments *n* and *k*, e.g., `LCM(15,25)=75.`When the arguments are not integers, their fractional part is ignored.

Ln(x)

Returns the natural logarithm of *x*, which has to be non-negative, e.g., `Ln(10)=2.30259`.

Log(x,b)

Returns the base *b* logarithm of *x*, which has to be non-negative, e.g., `Log(10,2)=3.32193`.

Log10(x)

Returns decimal logarithm of *x*, which has to be non-negative, e.g., `Log10(100)=2`.

Pow10(x)

Returns 10 raised to the power of *x*, e.g., `Pow10(2)=`10^2`=100`.

Round(x)

Returns the integer that is nearest to x, e.g., `Round(2.2)=2, Round(3.5)=4`.

Sign(x)

Returns 1 if *x*>0, 0 when *x*=0, and -1 if *x*<0, e.g., `Sign(2.2)=1, Sign(0)=0, Sign(-3.5)=-1`.

Sqrt(x)

Returns the square root of *x*, which has to be non-negative, e.g., `Sqrt(2)=1.41421`.

SqrtPi(x)

Returns square root of π multiplied by *x*, which has to be non-negative, e.g. `SqrtPi(2)=Sqrt(Pi()*2) =2.50663.` This function is provided for the sake of compatibility with Microsoft Excel.

Sum(x1,x2,...)

Returns the sum of its arguments, e.g., `Sum(2.2, 3.5, 1.3)=7.0.` `Sum()` requires at least two arguments.

SumSq(x1,x2,...)

Returns the sum of squares of its arguments, e.g., `SumSq(2.2, 3.5, 1.3)=18.78.` `SumSq()` requires at least two arguments.

Trim(x,lo,hi)

Trims the value of the argument *x* to a value in the interval <*lo*, *hi*>. If $x \le lo$, the function returns *lo*, if $x \ge hi$, the function returns *hi*, if $lo < x < hi$, the function returns *x*. The function is a shortcut to two nested conditional functions `If()` and is equivalent to `If(x<lo,lo,If(x>hi,hi,x))`. For example, `Trim(-0.5,0,1)=0, Trim(0.5,0,1)=0.5, Trim(1.5,0,1)=1`.

Truncate(x)

Returns the integer part of *x*, e.g., `Truncate(2.2)=2`.

8.18.5 Combinatoric Functions

Combin(n,k)

Returns the number of combinations of distinct k elements from among n elements, e.g., `Combin(10,2)=45`.

Fact(n)

Returns the factorial of *n*, e.g., `Fact(5)=5!=120.` `Fact` of a negative number returns 0.

FactDouble(n)

Returns the product of all even (when *n* is even) or all odd (when *n* is odd) numbers between 1 and *n*, e.g., `FactDouble(5)=15, FactDouble(6)=48.` `FactDouble` of a negative number returns 0.

Multinomial(n1,n2,...)

Factorial of sum of arguments, divided by the factorials of all arguments, e.g., `Multinomial(2,5,3) =Fact(2+5+3)/(Fact(2)*Fact(5)*Fact(3)=10!/(2!*5!*3!)=2520.` All arguments of `Multinomial` have to be positive.

8.18.6 Trigonometric Functions

Acos(x)

Returns arccosine (*arcus cosinus*) of *x*, e.g., `Acos(-1)=3.14159`.

Asin(x)

Returns arcsine (*arcus sinus*) of *x*, e.g., `Asin(1)=1.5708`.

Atan(x)

Returns arctangent (*arcus tangens*) of *x*, e.g., `Atan(1)=0.785398`.

Atan2(y,x)

Returns arctangent (*arcus tangens*) from *x* and *y* coordinates, e.g., `Atan2(1,1)=0.785398`.

Cos(x)

Returns cosine (*cosinus*) of *x*, e.g., `Cos(1)=0.540302`.

Pi()

Returns constant π, e.g., `Pi()=3.14159`.

Sin(x)

Returns sine (*sinus*) of *x*, e.g., `Sin(1)=0.841471`.

Tan(x)

Returns tangent (*tangens*) of *x*, e.g., `Tan(1)=1.55741`.

8.18.7 Hyperbolic Functions

Cosh(x)

Returns the hyperbolic cosine of *x*, e.g., `Cosh(1)=1.54308`.

Sinh(x)

Returns the hyperbolic sine of *x*, e.g., `Sinh(1)=1.1752`.

Tanh(x)

Returns the hyperbolic tangent of *x*, e.g., `Tanh(1)=0.761594`.

8.18.8 Logical/Conditional functions

And(b1,b2,...)

Returns the logical conjunction of the arguments, which are all interpreted as Boolean expressions. If any of the expressions evaluates to a zero, `And` returns 0. For example, `And(1=1,2,3)=1, And(1=2,2,3)=0`.

Choose(index,v0,v1,...,vn)

Returns v_i if *index* is equal to *i*. When *index* evaluates to a value smaller than 0 or larger than *n*-1, the function returns 0. Examples:

```
Choose(0,1,2,3,4,5)=1
Choose(4,1,2,3,4,5)=5
Choose(7,1,2,3,4,5)=0
```

If(cond,tval,fval)

If *cond* evaluates to non-zero return *tval*, *fval* otherwise, e.g., `If(1=2,3,4)=4, If(1,5,10)=5`.

Max(x1,x2,...)

Returns the largest of the arguments x_i. Examples:

```
Max(1,2,3,4,5)=5
Max(-3,2,0,2,1)=2
```

Min(x1,x2,...)

Returns the smallest of the arguments x_i. Examples:

```
Min(1,2,3,4,5)=1
Min(-3,2,0,2,1)=-3
```

Or(b1,b2,...)

Returns the logical disjunction of the arguments, which are all interpreted as Boolean expressions. If any of the expressions evaluates to a non-zero value, `Or` returns 1. For example, `Or(1=1,0,0)=1, Or(1=0,0,0)=0`.

Switch(x,a1,b1,a2,b2,...,[def])

If *x*=*a1*, return *b1*, if *x*=*a2*, return *b2*, when *x* is not equal to any of *a*s, return *def* (default value), which is an optional argument. When *x* is not equal to any of *a*s and no *def* is defined, return 0. Examples:

```
Switch(3,1,111,2,222,3,333,4,444,5,555,999)=333
Switch(8,1,111,2,222,3,333,4,444,5,555,999)=999
Switch(8,1,111,2,222,3,333,4,444,5,555)=0
```

Xor(b1,b2,...)

Returns logical exclusive OR of all arguments, which are all interpreted as Boolean expressions. `Xor` returns a 1 if the number of logical expressions that evaluate to non-zero is odd and a 0 otherwise.

Examples:

```
Xor(0,1,2,-3,0)=1
Xor(1,1,0,2,2)=0
```

```
Xor(-5,1,1,-2,2)=1
```

8.18.9 Custom Functions

It is possible to extend the available function set by calling `DSL_network::SetExtFunctions`. The functions are defined at the network level and stored within XDSL file. `SetExtFunctions` requires a `std::vector<std::string>` with the textual definition of the functions. Each function definition has the following form:

function-name([parameter1[,parameter2...]]) = expression

For example, a function adding two numbers can be defined by the following string:

add(a,b)=a+b

Recursion is not allowed, but it is possible to use the preceding custom functions in the function expression, for example:

dbl(x)=x+x
*quad(x)=2*dbl(x)*

Reverse ordering of *dbl* and *quad* in the vector passed to `SetExtFunctions` will result in a parser error for *quad,* because the *dbl* function is not defined when *quad* is parsed.

Each parameter specified on the LHS of the equality sign must be used in the function body on the RHS. The following is an incorrect custom function definition, because parameter *c* is not used:

myFunc(a,b,c)=sin(a)+cos(b)

Custom function can be defined as a constant. In such case the function has no parameters, but its definition and function calls in node equations still require parenthesis:

gEarth()=9.80665

Random number generators can be used in function definitions:

StdNormal()=Normal(0,1)
TwoStep(lo,hi)=Steps(lo,(lo+hi)/2,hi,1,2)

To retrieve the custom functions defined for the network, call `DSL_network::GetExtFunctions`. By default, there are no custom functions. Each call to `SetExtFunctions` sets up a complete set of custom functions. To add new functions to the existing set, call `GetExtFunctions` first, then add new function definition to the output vector before calling `SetExtFunctions`.

8.19 Global functions

`DSL_errorStringHandler& DSL_errorH();`

Returns a reference to the global error handler.

`bool DSL_isFinite(double x);`

Returns true if `x` is a finite number.

`double DSL_nan();`

Returns a value representing not-a-number.

```
double DSL_inf();
```

Returns double-precision positive infinity value. For negative infinity, use `-DSL_inf()`.

```
bool DSL_isPermutation(const int* permutation, int size);
```

Returns `true` if the buffer pointed to by the `permutation` parameter with the specified `size` is a permutation (contains each number from the `0..size-1` range).

```
bool DSL_isCharValidForIdentifier(char c, bool start);
```

Returns `true` if character `c` can be used in a valid SMILE identifier. If `start` is `true`, the check is performed for the first character in the SMILE identifier (which cannot be a number or an underscore).

```
bool DSL_isValidIdentifier(const char *idToCheck);
```

Returns `true` if `idToCheck` is a valid SMILE identifier.

```
void DSL_appendInt(std::string &s, int x);
void DSL_appendDouble(std::string &s, double x);
```

Append the string representation of `x` to `s`.

Appendix M: Matlab and SMILE

9 Appendix M: Matlab and SMILE

It is possible to use C/C++ code with MATLAB by compiling it into `mex` (MATLAB executable) file. The next section of this manual contains a complete wrapper code (`matsmile.cpp` file), that contains functions for reading networks from file, setting evidence, and performing inference in discrete Bayesian networks and can be easily extend to include other functionality.

Before proceeding, refer to MATLAB documentation for detailed information about configuring your C++ compiler to work MATLAB. Also, please make sure that you have SMILE library compatible with the C++ compiler that you are going to use with MATLAB.

To compile the C++ code into `mex` file, use the following MATLAB command:

```
mex matsmile.cpp
```

The name of the compiled file depends on the operating system that you are using. On 64-bit Windows, it will be `matsmile.mexw64`. If the compilation was successful, MATLAB code can call new function `matsmile`. Here is the example MATLAB code that reads the network file, sets the evidence for the *Forecast* node to *Moderate*, and then performs inference and displays the probabilities calculated for node *Success*.

```
net = matsmile('newNetwork');
matsmile('readFile', net, 'VentureBN.xdsl');
matsmile('setEvidence', net, 'Forecast', 'Moderate');
matsmile('updateBeliefs', net);
beliefs = matsmile('getValue', net, 'Success');
outcomeIds = matsmile('getOutcomeIds', net, 'Success');
fprintf("%s\n", outcomeIds + "=" + beliefs);
matsmile('deleteNetwork', net);
```

MATLAB code can access wrapper's functionality through `matsmile` function. First argument is always the name of the wrapper sub-function passed as string. Note that the network object created by the `matsmile('newNetwork')` call must be deallocated by the corresponding `matsmile('deleteNetwork', net)`. Omitting the latter will cause memory leaks.

The MATLAB wrapper is implemented as a class `MexFunction` derived form `matlab::mex::Function`. The `MexFunction::findFunction` method code contains `wrapFx` array, which has an element for each function accessible through a `matsmile` call. Each of these includes function name, the number of input parameters (used for validation at runtime) and a pointer to a class member function actually implementing the call. Each member function referenced in `wrapFx` returns `void` and has two arguments of the type `matlab::mex::ArgumentList&` for values passed from MATLAB side as output and input arguments, respectively. Other than `newNetwork`, all functions expect network object to be passed as the second argument in MATLAB code.

To extend the wrapper, you can add a new class member function implementing the functionality that you want to add. Next, you need to add a new element to the `wrapFx` array. At runtime, the `findFunction` method will be able to recognize the function name passed as first argument in the MATLAB code and invoke the appropriate member function. You can get the pointer to the `DSL_network` pointer created earlier by `matsmile('newNetwork')` call:

```
matlab::data::TypedArray<uint64_t> arr = inputs[1];
auto net = decodePtr(arr[0]); // type of net is DSL_network*
```

9.1 matsmile.cpp

```
// matsmile.cpp
// Simple MEX wrapper for use with existing discrete Bayesian networks

#include "mex.hpp"
#include "mexAdapter.hpp"
#include "smile_license.h"
#include "smile.h"

using namespace std;

class MexFunction : public matlab::mex::Function {
public:
    void operator()(matlab::mex::ArgumentList outputs, matlab::mex::ArgumentList inputs) {
        auto f = findFunction(outputs, inputs);
        (this->*(f.ptr))(outputs, inputs);
    }

private:
    shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();
    matlab::data::ArrayFactory factory;
    union uint64netptr { uint64_t integer; DSL_network* pointer; };
    typedef void (MexFunction::* PTR)(matlab::mex::ArgumentList&, matlab::mex::ArgumentList&);
    struct MatSmileFunction {
        const char* name;
        int inputSize;
        PTR ptr;
    };

    void error(const string& msg) {
        matlabPtr->feval(u"error",
            0, vector<matlab::data::Array>({ factory.createScalar(msg.c_str()) }));
    }

    matlab::data::TypedArray<matlab::data::MATLABString> createStringArray(const vector<string>& vec)
        return factory.createArray({ vec.size() }, vec.begin(), vec.end());
    }

    const MatSmileFunction& findFunction(matlab::mex::ArgumentList& outputs, matlab::mex::ArgumentLis
        static const MatSmileFunction wrapFx[] = {
            "newNetwork", 1, &MexFunction::newNetwork,
            "deleteNetwork", 2, &MexFunction::deleteNetwork,
            "getNodeCount", 2, &MexFunction::getNodeCount,
            "readFile", 3, &MexFunction::readFile,
            "updateBeliefs", 2, &MexFunction::updateBeliefs,
            "setEvidence", 4, &MexFunction::setEvidence,
            "getValue", 3, &MexFunction::getValue,
            "getAllNodeIds", 2, &MexFunction::getAllNodeIds,
            "getOutcomeIds", 3, &MexFunction::getOutcomeIds,
            "isEvidence", 3, &MexFunction::isEvidence,
            "getEvidence", 3, &MexFunction::getEvidence,
            "getOutcomeCount", 3, &MexFunction::getOutcomeCount
        };
```

```
    matlab::data::CharArray charArray = inputs[0];
    auto functionName = charArray.toAscii().c_str();
    const auto iterator = find_if(begin(wrapFx), end(wrapFx), [&functionName](auto f) {
        return !strcmp(f.name, functionName);
        });
    if (iterator == end(wrapFx)) {
        string msg = "Cannot find function with name ";
        msg += functionName;
        error(msg);
    }
    int inputSize = iterator->inputSize;
    if (inputs.size() != inputSize) {
        string msg = "Invalid input size for function: ";
        msg += functionName;
        msg += ". Expected: ";
        DSL_appendInt(msg, inputSize);
        msg += ", given: ";
        DSL_appendInt(msg, (int)inputs.size());
        error(msg);
    }
    if (outputs.size() > 1) {
        error("Outputs size cannot be greater than 1.");
    }
    return *iterator;
}

int validateNodeId(const DSL_network& net, const char* nodeId) {
    int handle = net.FindNode(nodeId);
    if (handle < 0) {
        string msg = "Cannot find node with ID '";
        msg += nodeId;
        msg += '\'';
        error(msg);
    }
    return handle;
}

int validateOutcomeId(const DSL_network& net, int nodeHandle, const char* outcomeId) {
    const DSL_node* node = net.GetNode(nodeHandle);

    const DSL_idArray* outcomeNames = node->Def()->GetOutcomeIds();
    int outcomeIndex = outcomeNames->FindPosition(outcomeId);
    if (outcomeIndex < 0) {
        string msg = "Invalid outcome identifier '";
        msg += outcomeId;
        msg += "' for node '";
        msg += node->GetId();
        msg += '\'';
        error(msg);
    }
    return outcomeIndex;
}

uint64_t encodePtr(DSL_network* pointer) {
    uint64netptr ivp;
```

```
    ivp.pointer = pointer;
    return ivp.integer;
}

DSL_network* decodePtr(uint64_t integer) {
    uint64netptr ivp;
    ivp.integer = integer;
    return ivp.pointer;
}

void newNetwork(matlab::mex::ArgumentList& outputs, matlab::mex::ArgumentList& inputs) {
    auto net = new DSL_network();
    outputs[0] = factory.createScalar<uint64_t>(encodePtr(net));
}

void deleteNetwork(matlab::mex::ArgumentList& outputs, matlab::mex::ArgumentList& inputs) {
    matlab::data::TypedArray<uint64_t> arr = inputs[1];
    auto net = decodePtr(arr[0]);
    delete net;
}

void getNodeCount(matlab::mex::ArgumentList& outputs, matlab::mex::ArgumentList& inputs) {
    matlab::data::TypedArray<uint64_t> arr = inputs[1];
    auto net = decodePtr(arr[0]);
    outputs[0] = factory.createScalar(net->GetNumberOfNodes());
}

void readFile(matlab::mex::ArgumentList& outputs, matlab::mex::ArgumentList& inputs) {
    matlab::data::TypedArray<uint64_t> arr = inputs[1];
    auto net = decodePtr(arr[0]);
    matlab::data::CharArray charArray = inputs[2];
    auto filePath = charArray.toAscii();
    int result = net->ReadFile(filePath.c_str());
    if (result != DSL_OKAY) {
        string msg = "Cannot read file from '";
        msg += filePath;
        msg += "' ErrNo ";
        DSL_appendInt(msg, result);
        error(msg);
    }
}

void updateBeliefs(matlab::mex::ArgumentList& outputs, matlab::mex::ArgumentList& inputs) {
    matlab::data::TypedArray<uint64_t> arr = inputs[1];
    auto net = decodePtr(arr[0]);
    int result = net->UpdateBeliefs();
    if (result != DSL_OKAY) {
        string msg = "Update Beliefs failed. ErrNo ";
        DSL_appendInt(msg, result);
        error(msg);
    }
}

void setEvidence(matlab::mex::ArgumentList& outputs, matlab::mex::ArgumentList& inputs) {
    matlab::data::TypedArray<uint64_t> arr = inputs[1];
```

```
    auto net = decodePtr(arr[0]);
    matlab::data::CharArray nodeIdCharArray = inputs[2];
    matlab::data::CharArray outcomeIdCharArray = inputs[3];
    int nodeHandle = validateNodeId(*net, nodeIdCharArray.toAscii().c_str());
    int outcomeIndex = validateOutcomeId(*net, nodeHandle, outcomeIdCharArray.toAscii().c_str())
    net->GetNode(nodeHandle)->Val()->SetEvidence(outcomeIndex);
}

void getValue(matlab::mex::ArgumentList& outputs, matlab::mex::ArgumentList& inputs) {
    matlab::data::TypedArray<uint64_t> arr = inputs[1];
    auto net = (DSL_network*)decodePtr(arr[0]);
    matlab::data::CharArray nodeIdCharArray = inputs[2];
    auto nodeId = nodeIdCharArray.toAscii();
    int nodeHandle = validateNodeId(*net, nodeId.c_str());
    const DSL_nodeVal* nodeValue = net->GetNode(nodeHandle)->Val();
    if (!nodeValue->IsValueValid()) {
        string msg = "Invalid node value for node ";
        msg += nodeId;
        error(msg);
    }
    const DSL_Dmatrix* m = nodeValue->GetMatrix();
    const double* p = m->GetItems().Items();
    size_t arraySize = m->GetSize();
    matlab::data::TypedArray<double> resArr = factory.createArray({ arraySize }, p, p + arraySize
    outputs[0] = resArr;
}

void getAllNodeIds(matlab::mex::ArgumentList& outputs, matlab::mex::ArgumentList& inputs) {
    matlab::data::TypedArray<uint64_t> arr = inputs[1];
    auto net = (DSL_network*)decodePtr(arr[0]);
    size_t count = net->GetNumberOfNodes();
    vector<string> ids;
    ids.resize(count);
    DSL_intArray nodes;
    net->GetAllNodes(nodes);
    for (int i = 0; i < count; i++) {
        ids[i] = string(net->GetNode(nodes[i])->GetId());
    }
    auto resArr = createStringArray(ids);
    outputs[0] = resArr;
}

void getOutcomeCount(matlab::mex::ArgumentList& outputs, matlab::mex::ArgumentList& inputs) {
    matlab::data::TypedArray<uint64_t> arr = inputs[1];
    auto net = decodePtr(arr[0]);
    matlab::data::CharArray nodeIdCharArray = inputs[2];
    int nodeHandle = validateNodeId(*net, nodeIdCharArray.toAscii().c_str());
    outputs[0] = factory.createScalar(net->GetNode(nodeHandle)->Def()->GetNumberOfOutcomes());
}

void getOutcomeIds(matlab::mex::ArgumentList& outputs, matlab::mex::ArgumentList& inputs) {
    matlab::data::TypedArray<uint64_t> arr = inputs[1];
    auto net = decodePtr(arr[0]);
    matlab::data::CharArray nodeIdCharArray = inputs[2];
    int nodeHandle = validateNodeId(*net, nodeIdCharArray.toAscii().c_str());
```

```
        auto def = net->GetNode(nodeHandle)->Def();
        const DSL_idArray* names = def->GetOutcomeIds();
        size_t count = def->GetNumberOfOutcomes();
        vector<string> namesVec(names->begin(), names->end());
        auto resArr = createStringArray(namesVec);
        outputs[0] = resArr;
    }

    void isEvidence(matlab::mex::ArgumentList& outputs, matlab::mex::ArgumentList& inputs) {
        matlab::data::TypedArray<uint64_t> arr = inputs[1];
        auto net = decodePtr(arr[0]);
        matlab::data::CharArray nodeIdCharArray = inputs[2];
        int nodeHandle = validateNodeId(*net, nodeIdCharArray.toAscii().c_str());
        outputs[0] = factory.createScalar(0 != net->GetNode(nodeHandle)->Val()->IsEvidence());
    }

    void getEvidence(matlab::mex::ArgumentList& outputs, matlab::mex::ArgumentList& inputs) {
        matlab::data::TypedArray<uint64_t> arr = inputs[1];
        auto net = decodePtr(arr[0]);
        matlab::data::CharArray nodeIdCharArray = inputs[2];
        int nodeHandle = validateNodeId(*net, nodeIdCharArray.toAscii().c_str());
        DSL_node* node = net->GetNode(nodeHandle);
        int evidence = node->Val()->GetEvidence();
        if (evidence < 0) {
            string msg = "Evidence for node ";
            msg += node->GetId();
            msg += " does not exist";
            error(msg);
        }
        std::string outcomeId = node->Def()->GetOutcomeIds()->Subscript(evidence);
        outputs[0] = factory.createCharArray(outcomeId);
    }
};
```

This page is intentionally left blank.

Acknowledgments

10 Acknowledgments

SMILE internally uses portions of the following two software libraries: micro-ECC and Expat. Both require an acknowledgment that we are reproducing below.

micro-ECC

Expat