# Table of Contents

1. Introduction .......................................................... 7
2. Licensing .............................................................. 9
3. Compiler and Linker Options .................................. 11
   3.1 Visual C++ .......................................................... 12
   3.2 gcc ................................................................. 13
4. Hello, SMILE! ....................................................... 15
   4.1 Success/Forecast model ....................................... 16
   4.2 The program ....................................................... 16
   4.3 hello.cpp .......................................................... 18
   4.4 VentureBN.xdsl .................................................. 19
5. Using SMILE ......................................................... 21
   5.1 Main header file .................................................. 22
   5.2 Naming conventions ............................................. 22
   5.3 Error handling ..................................................... 22
   5.4 Networks, nodes and arcs .................................... 23
      5.4.1 Network ......................................................... 23
      5.4.2 Nodes ........................................................... 23
      5.4.3 Arcs .............................................................. 25
   5.5 Anatomy of a node .............................................. 25
      5.5.1 Multidimensional arrays ................................ 25
      5.5.2 Node definition .............................................. 27
      5.5.3 Node value ..................................................... 27
      5.5.4 Other node attributes ..................................... 28
   5.6 Input and output ................................................. 29
   5.7 Inference .......................................................... 29
   5.8 User properties .................................................. 30
   5.9 Cases ............................................................... 30
   5.10 Sensitivity analysis ............................................. 31
   5.11 Diagnosis ........................................................ 35
   5.12 Dynamic Bayesian networks ............................. 35
   5.13 Canonical nodes ............................................... 35
   5.14 Continuous models .......................................... 35
   5.15 Hybrid models .................................................. 35
# Table of Contents

## 5.16 Datasets
- 5.16.1 Text file I/O ................................................................. 36
- 5.16.2 Discrete and continuous variables ......................................................... 37
- 5.16.3 Generating data from network ................................................................. 38
- 5.16.4 Discretization .................................................................................. 39

## 5.17 Learning
- 5.17.1 Learning network structure ................................................................. 40
- 5.17.2 Learning network parameters ............................................................... 41
- 5.17.3 Validation ........................................................................................ 42

- 6.1 DSL_network ......................................................................................... 46
- 6.2 DSL_node ............................................................................................. 53
- 6.3 DSL_nodeDefinition .............................................................................. 54
- 6.4 DSL_nodeValue .................................................................................... 56
- 6.5 DSL_Dmatrix ......................................................................................... 58
- 6.6 DSL_dataset ........................................................................................ 62
- 6.7 DSL_dataGenerator .............................................................................. 67
- 6.8 DSL_validator ....................................................................................... 68
- 6.9 DSL_em ................................................................................................. 71
- 6.10 DSL_bs ............................................................................................... 73
- 6.11 DSL_pc ................................................................................................. 74
- 6.12 DSL_tan ............................................................................................... 75
- 6.13 DSL_abn ............................................................................................... 76
- 6.14 DSL_nb ................................................................................................. 77
- 6.15 DSL_bkgndKnowledge ...................................................................... 78
- 6.16 DSL_pattern ....................................................................................... 79
- 6.17 DSL_progress ..................................................................................... 80

## 7. Tutorials
- 7.1 main.cpp ............................................................................................. 84
- 7.2 Tutorial 1: Creating a Bayesian Network ............................................... 85
  - 7.2.1 tutorial1.cpp .................................................................................... 87
- 7.3 Tutorial 2: Inference with a Bayesian Network ........................................ 89
  - 7.3.1 tutorial2.cpp .................................................................................... 92
- 7.4 Tutorial 3: Exploring the contents of a model ........................................ 94
  - 7.4.1 tutorial3.cpp .................................................................................... 96
Table of Contents

7.5 Tutorial 4: Creating the Influence Diagram ................................................................. 98
  7.5.1 tutorial4.cpp ........................................................................................................... 100

7.6 Tutorial 5: Inference in an Influence Diagram ......................................................... 102
  7.6.1 tutorial5.cpp ....................................................................................................... 103

8. Acknowledgments 107

Index 111
This page is intentionally left blank.
Introduction
Welcome to SMILE Programmer’s Manual, version 2.5.1, built on 1/18/2019. For the most recent version of this manual, please visit http://support.bayesfusion.com/docs/.

SMILE (Structural Modeling, Inference, and Learning Engine) is the 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 (formerly known as OS X) and Linux. We assume that the reader has basic knowledge of the C++ language. We also provide wrappers exposing SMILE functionality to programs written in Java (jSMILE), Python (PySMILE), .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. The documentation for wrappers is provided in the 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! 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.

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

If you have used SMILE before SMILE 1.2 (released in November 2017), here are three important changes introduced in version 1.2:
- programs linking with SMILE are required to define and initialize the specific licensing key variables; see the Licensing section below
- the learning library (SMILearn) is included in SMILE now (so there's just one library). The functionality previously included in the smilearn.h header file is now accessed through smile.h; smilearn.h is no longer available.
- the global ErrorH object was replaced by the global DSL_errorH function.
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). This file is not included in SMILE distribution, it is personalized by BayesFusion, LLC, for you or your organization. 6-month academic and free 30-day evaluation license keys can be obtained directly at https://download.bayesfusion.com.

6-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.
3 Compiler and Linker Options

SMILE is distributed as a C++ library along with the 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.

3.1 Visual C++

We support compiled SMILE library for multiple versions of Visual C++. Each zip file 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. 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 **don't need** to add smile*.lib files to the list of libraries in project’s linker settings. Doing that is very likely to cause linker errors.

However, you still need to tell 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 preprocessor settings.

**3.2 gcc**

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’ll 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 file(s) with the .cpp extension.

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

This command compiles the program, then links it with libsmile.a. The output executable binary has the default filename a.out.

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.
Hello, SMILE!
4 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 the GeNIe on-line help (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](http://support.bayesfusion.com/docs). Alternatively, create a file named *VentureBN.xdsl* with any text editor by copying the content of the *VentureBN.xdsl* section below.

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

4.1 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.

<table>
<thead>
<tr>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>0.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Good</th>
<th>Moderate</th>
<th>Poor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.4</td>
<td>0.4</td>
<td>0.2</td>
</tr>
</tbody>
</table>

4.2 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’ll need to #include your SMILE license key. See
Hello, SMILE!

the Licensing 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 don’t expect to see any messages. If VentureBN.xdsl is not in the current directory, you’ll 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;
    }

We want to set the evidence on the Forecast node to Moderate. SMILE uses integer handles to identify nodes, so we need to convert known node identifier ("Forecast") to handle. Handles are always non-negative; if node identifier is not found in the network the value returned by the FindNode method will be less than zero.

    int handle = net.FindNode("Forecast");
    if (handle < 0)
    {
        return handle;
    }

With node handle we can proceed to retrieve the index of the outcome by searching for outcome identifier ("Moderate"). Again, if the identifier is not found, the return value from FindPosition method will be negative.

    DSL_node *f = net.GetNode(handle);
    int idx = f->Definition()->GetOutcomesNames()->FindPosition("Moderate");
    if (idx < 0)
    {
        return idx;
    }

Now we can set the evidence and update the network.

    f->Value()->SetEvidence(idx);
    net.UpdateBeliefs();

Note that in the real-world program we would test the status codes returned by both SetEvidence and UpdateBeliefs. Our sample program only checks for errors caused by missing network file or nodes/outcomes not present in the network.

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 *s = net.GetNode(handle);
Hello, SMILE!

const DSL_Dmatrix &beliefs = *s->Value()->GetMatrix();
const DSL_idArray &outcomes = *s->Definition()->GetOutcomesNames();
for (int i = 0; i < outcomes.NumItems(); i++)
{
    printf("%s=%g\n", outcomes[i], beliefs[i]);
}

This ends the source of the program. If you compile and run it, the output should be:

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 section of this manual.

4.3  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;
    }

    int handle = net.FindNode("Forecast");
    if (handle < 0)
    {
        return handle;
    }

    DSL_node *f = net.GetNode(handle);
    int idx =
        f->Definition()->GetOutcomesNames()->FindPosition("Moderate");
    if (idx < 0)
    {
        return idx;
    }
    f->Value()->SetEvidence(idx);

    net.UpdateBeliefs();

    handle = net.FindNode("Success");
    if (handle < 0)
    {
        return handle;
    }

DSL_node *s = net.GetNode(handle);
const DSL_Dmatrix &beliefs = *s->Value()->GetMatrix();
const DSL_idArray &outcomes =
  *s->Definition()->GetOutcomesNames();
for (int i = 0; i < outcomes.NumItems(); i++)
{
    printf("%s=%g\n", outcomes[i], beliefs[i]);
}
return DSL_OKAY;

4.4 VentureBN.xdsl

<?xml version="1.0" encoding="ISO-8859-1"?>
<smile version="1.0" id="VentureBN" numsamples="1000">
<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 2.1.1104.2" name="VentureBN"
         faultnameformat="nodestate">
    <node id="Success">
      <name>Success of the venture</name>
      <interior color="e5f6f7" />
      <outline color="0000bb" />
      <font color="000000" name="Arial" size="8" />
      <position>54 11 138 62</position>
    </node>
    <node id="Forecast">
      <name>Expert forecast</name>
      <interior color="e5f6f7" />
      <outline color="0000bb" />
      <font color="000000" name="Arial" size="8" />
      <position>63 105 130 155</position>
    </node>
  </genie>
</extensions>
</smile>
Using SMILE
Using SMILE

5 Using SMILE

5.1 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).

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 section for more details, including information on how to obtain evaluation or academic license from our website.

5.2 Naming conventions

The names of the majority of 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
```

5.3 Error handling

Most of the SMILE functions and methods return an integer status code. Negative numbers are used for specific error codes. See errors.h in SMILE’s distribution directory for the list of codes.

Sometimes SMILE emits the warning or error message. These are stored by default in the global object of the class DSL_errorStringHandler, access to which you can get by calling the DSL_errorH function. By default, your program is not notified about these messages: you need to query DSL_errorH’s stored codes and strings.

Alternatively, your program can redirect these messages to FILE* (including stdout and stderr):

```
DSL_errorH().RedirectToFile(stdout);
```
For the complete control over the messages, use \texttt{DSL\_errorStringHandler::Redirect} method. Its only argument is a pointer to the instance of the class derived from \texttt{DSL\_errorStringRedirect}.

\begin{verbatim}
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);
\end{verbatim}

Tutorials included in this manual redirect errors to \texttt{stdout}.

### 5.4 Networks, nodes and arcs

The most important class defined by SMILE is \texttt{DSL\_network}. The objects of this class act as containers for nodes and are responsible for node creation and destruction. Nodes and arcs are always created and destroyed by invoking \texttt{DSL\_network}'s methods.

Nodes are created by the \texttt{DSL\_network::AddNode} method and destroyed by \texttt{DSL\_network::DeleteNode}.

The arcs are created by the \texttt{DSL\_network::AddArc} method and destroyed by \texttt{DSL\_network::RemoveArc}.

#### 5.4.1 Network

To create a network, simply use a default constructor. You can create \texttt{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. Many programs will call the \texttt{ReadFile} or \texttt{ReadString} methods to initialize the content of the network after creating it. The \texttt{UpdateBeliefs} method invokes the inference algorithm on the network and sets the node values. The \texttt{CalcProbEvidence} method computes the probability of evidence currently set in the network.

#### 5.4.2 Nodes

Nodes are represented by the \texttt{DSL\_node} object. Your program should always use the \texttt{DSL\_network::AddNode} method to create a node within the network and never use \texttt{DSL\_node} constructor directly. To delete the node, use \texttt{DSL\_network::DeleteNode}; never try to use operator delete.
Within the network, the nodes are uniquely identified by their handle. The node handle is a non-negative integer, which is preserved when network is copied using copy constructor or operator=. Most of DSL_network methods dealing with nodes uses node handles as input arguments. In general, the node handles may change when you write the network to file and read it later. This means you should not rely on handles as persistent identifiers.

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

```c
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.

In addition to handles, 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 (it may be changed later). The identifiers in SMILE start are case-sensitive, start with a letter and contain letters, digits and underscores. Node’s identifier can be converted to node handle with a call to DSL_network::FindNode method.

```c
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, 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.

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.
5.4.3 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 the arc, call `DSL_network::RemoveArc`.

The graph defined by nodes and arcs in `DSL_network` is a directed acyclic graph (DAG) at all times. If your call to `AddArc` would result in a cycle in the graph, the method returns an error code. To check if adding an arc would introduce a cycle, you can call `DSL_network::RemainsAcyclic` method.

To inspect the graph structure, use `DSL_network::GetParents` and `GetChildren` methods:

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

5.5 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.

5.5.1 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 specifies the prior probability of every state of the node.
Using SMILE

The conditional probability tables are stored as vectors of doubles that are a flattened version 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. 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 $2 \times 3 \times 3 = 18$. The annotation arrows in the image (not part of the actual GeNIe window) show the ordering of the entries in the linear buffer which DSL_Dmatrix uses internally. The first (or rather, the zero-th, because all indexes in SMILE are zero-based) element, the one with the value of 0.7 and yellow background, represents $P(\text{Forecast}=\text{Good} \mid \text{Success of the venture}=\text{Success} \& \text{State of the economy}=\text{Up})$. It is followed by the probabilities 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, and so on.

![Image of GeNIe's node properties window](image)

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 entry representing node for which CPT is defined moving fastest.)

Note that DSL_Dmatrix is not used exclusively for CPTs. Other uses of this class in SMILE include expected utility tables and marginal probability distributions.
5.5.2 Node definition

The definition of the node specifies how it interacts with other nodes in the network. The node definition is written as part of the network by `DSL_network::WriteFile` and `WriteString`. For general chance node the definition consists of conditional probability table (CPT) and list of state names. To access it use `DSL_node::Definition` method, which returns a pointer to the instance of the class derived from the `DSL_nodeDefinition`.

```c++
int nodeHandle = ...;
DSL_node *node = net.GetNode(nodeHandle);
DSL_nodeDefinition *def = node->Definition();
```

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

```c++
int nodeHandle = net.AddNode(DSL_CPT, "myNodeId");
DSL_nodeDefinition *def = net.GetNode(nodeHandle)->Definition();
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 use `static_cast` (or use old-style C cast) and obtain an access to type-specific functionality of the object, SMILE provides general purpose virtual methods defined in `DSL_nodeDefinition` and overridden in derived classes, which makes casting unnecessary most of the time. Two of these methods are `DSL_nodeDefinition::GetMatrix` and `GetOutcomesNames`, which give access to node’s parameters and state names. Note that some of the node types do not support all of the operations. For example, the decision node does not have any numeric parameters, therefore its definition object of `DSL_list` class returns NULL from its `GetMatrix` method.

5.5.3 Node value

The value of the node contains the values (typically, the marginal probability distribution or the 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::Value` method, is an instance of the class derived from `DSL_nodeValue`. 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 node definition) part of the input to the inference algorithm. As with the definition part of node, most of the time there is no need to
cast the pointer returned by `DSL_node::Value` to specific class, because the base class provides the set of general purpose virtual functions overriden by derived value classes.

```cpp
int evidenceNodeHandle = ...;
DSL_nodeValue *evVal = net.GetNode(evidenceNodeHandle)->Value();
evVal.SetEvidence(1); 1 is the 0-based outcome index
net.UpdateBeliefs();
int beliefNodeHandle = ...;
DSL_nodeValue *beVal = net.GetNode(beliefNodeHandle)->Value();
if (beVal->IsValueValid())
{
    const DSL_Dmatrix *m = beVal->GetMatrix();
    // read the matrix contents
}
```

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 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 the virtual evidence using `DSL_nodeValue::SetVirtualEvidence` method.

### 5.5.4 Other node attributes

Node’s attributes 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.

- **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.

```cpp
int nodeHandle = ...;
DSL_node *node = net.GetNode(nodeHandle);
DSL_nodeInfo &info = node->Info();
printf("Node name: %s\n", info.Header().GetName());
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);
```

Note that for convenience `DSL_node` allows direct read access to its identifier by `DSL_node::GetId` method, which calls `Info().Header().GetId()`.
5.6 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 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 type based on the filename extension. The file type can be explicitly specified in the call if needed.

```c++
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 read failure the program listed above will display the specific error message on the standard output (due to earlier RedirectToFile(stdout) call).

5.7 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 Bayesian Networks is clustering over network preprocessed with relevance. To switch between Bayesian Network algorithms, call DSL_network::SetDefaultBNAlgorithm and pass the 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.
To obtain probability of evidence currently set in the network, call 
DSL_network::CallProbEvidence.

5.8 User properties

To integrate data specific to your application with SMILE, you can use SMILE’s user 
properties. The 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, the user properties are stored in the XDSL files.

```cpp
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: 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.

5.9 Cases

SMILE includes management of cases, which allow 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 going 
to be available.

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

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

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

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

5.10 Sensitivity analysis

DSL_sensitivity class contains an implementation of a sensitivity analysis algorithm.

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. Note that the derivatives are dependent on the 
evidence currently set in the network. These derivatives 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 deviation in x may lead to a 
large difference in the posteriors of the targets. If the derivative is small, then even large 
deviations in the parameter 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:

\[ y = \frac{Ax + B}{Cx + D} \]

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:

\[ y' = \frac{(AD - BC)}{(Cx + D)^2} \]

The denominator is positive, therefore 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:

\[ y_1 = \frac{B}{D}, \quad y_2 = \frac{(A + B)}{(C+D)} \]
The sign of $AD-BC$ determines which value is minimum and which is maximum.

As an illustration, let's 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](https://repo.bayesfusion.com). The choice of the target node, evidence and CPT parameter is motivated only by the large value of the derivative:

$$x = P(\text{bilirubin}=a19_7 \mid \text{Hyperbilirubinemia}=\text{absent}, \text{PBC}=\text{absent}, \text{Cirrhosis}=\text{absent}, \text{gallstones}=\text{absent}, \text{ChHepatitis}=\text{absent})$$

$$y = P(\text{PBC}=\text{present} \mid \text{ast}=149_40, \text{irregular_liver}=\text{present}, \text{bilirubin}=a19_7, \text{proteins}=a5_2)$$

Note that the right side of the vertical bar in the definition of $x$ determines the position of the parameter within the CPT of the $\text{bilirubin}$ node, while the right 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 $y(x)$, blue line is a tangent 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, set 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*.

```csharp
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's 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)->Definition()->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 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. Some nodes, which do not affect target node's posteriors, are
dropped early by the relevance. The matrix objects returned for these nodes by
GetSensitivity and GetCoefficients will be empty (DSL_Dmatrix::GetSize will
return zero). Unlike the example above, your code should include the check for the matrix
size and assume that target is not sensitive to changes in node under study if the sensitivity
matrix is empty.

In the usual case, the programs calculating sensitivity do not focus on the 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 a 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.

5.11 Diagnosis

(to be done)

5.12 Dynamic Bayesian networks

(to be done)

5.13 Canonical nodes

(to be done)

5.14 Continuous models

(to be done)

5.15 Hybrid models

(to be done)

5.16 Datasets

The data used by SMILE for learning and network validation is stored in the objects of the `DSL_dataset` class. The dataset is a table-like structure. Its columns are called `variables` and rows are called `records`. 
The contents of the dataset can be loaded from a text file. Alternatively, your program can initialize the structure of the dataset, then add records containing actual data.

5.16.1 Text file I/O

You can load the contents of the dataset from a text file by calling `DSL_dataset::ReadFile`. For the illustration purposes, let's assume we have the 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 dataset will be structured like this:

- variable `VarA` is continuous. The missing value was replaced by `sqrt(-1)`.
- variable `VarB` is discrete. The missing value was replaced by `-1`.
- variable `VarC` is also discrete, and has the string labels associated with its integer values. The missing value 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 the discrete columns should be replaced by 999:
DSL_dataset ds;
DSL_datasetParseParams params;
params.columIdsPresent = false;
params.missingValueToken = "N/A";
params.missingInt = 999;
int res = ds.ReadFile("datafile.txt", &params);

To write the contents of the dataset 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 example 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.columIdsPresent = true;
params.missingValueToken = "(none)"
params.separator = ',';
int res = ds.WriteFile("datafile.csv", &params);

5.16.2 Discrete and continuous variables

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

DSL_dataset ds;
    ds.AddFloatVar("Var1");
    ds.AddIntVar("Var2");
    ds.AddIntVar("Var3");
    vector<string> stateNames(3);
    stateNames[0] = "StateX";
    stateNames[1] = "StateY";
    stateNames[2] = "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 hardcoded index, as we know what variables were added to the dataset just prior to DSL_dataset::SetStateNames.

You can set the number of records in the dataset upfront with a call to DSL_dataset::SetNumberOfRecords, or call AddEmptyRecord for each record you're appending to the dataset.

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

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're not used for data entry.

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

The following code snippet displays content of any dataset. 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.

```cpp
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");
    }
}
```

5.16.3 Generating data from network

Given that a Bayesian network is a representation of the joint probability distribution over its variables, you can generate a dataset based on this distribution. Use the `DSL_dataGenerator` class for this purpose. The `DSL_dataGenerator::GenerateData` method has three overloads, which write the data to the following outputs:
• DSL_dataset object

• text file specified by a filename

• any object derived from the pure abstract class DSL_dataGeneratorOutput

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

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

### 5.16.4 Discretization

To discretize the dataset variable, use the DSL_dataset::Discretize method. To convert variable to the discrete variable with 10 states use the following code:

```cpp
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.

### 5.17 Learning

Learning in SMILE can perform two tasks:

• structure learning: create a new network from a dataset
• parameter learning: refine parameters (CPTs) in an existing network

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

### 5.17.1 Learning network structure

The following classes can be used to learn DSL_network from 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 learning algorithm and call its Learn method:

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

If algorithm succeeds, DSL_OKAY is returned and the DSL_network passed as an argument to Learn is the output of the actual learning. Note that every variable in the dataset 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 with EM.

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:

```cpp
DSL_bs baySearch;
baySearch.nrIteration = 10;
baySearch.maxParents = 4;
int res = baySearch.Learn(ds, net);
```
The settings for the learning algorithms are described in detail in the Reference section.

SMILE also contains the DSL_pc class, which implements the PC structure learning algorithm (algorithm name is an acronym derived from its inventors' names). This algorithm also uses DSL_dataset as data source, but instead of DSL_network 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 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 the arc between X and Y and forbids the arc between Z and Y. It is assumed that dataset contains the variables with the identifiers used in the calls to DSL_dataset::FindVariable.

```cpp
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);
```

5.17.2 Learning network parameters

To learn the parameters in the 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 dataset variables and network nodes. If the variables and nodes have identical identifiers, you can use the DSL_dataset::MatchNetwork method:

```cpp
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)
```
Using SMILE

{ 
    DSL_em em; 
    res = em.Learn(ds, net, matching); 
}

If your network and data cannot be automatically matched with MatchNetwork, you can build the 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 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 the 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); 

5.17.3 Validation

To evaluate the predictive quality of your network you can use the DSL_validator class.

The DSL_validator constructor requires a references to DSL_dataset and DSL_network objects to be specified. To properly match the network and data the constructor also requires the 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.

For each record in the dataset during the validation, 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 a predicted outcome. The prediction is compared with an outcome in the dataset variable associated with the class node. The number of matches and calculated posteriors are used to obtain the accuracy, confusion matrix, ROC and calibration curves.

Validation can be either performed without parameter learning using DSL_validator::Test method, or with parameter learning using DSL_validator::KFold and LeaveOneOut methods. K-fold crossvalidation divides the dataset into K parts of equal size, trains the
network on K-1 parts, and tests it on the last, Kth 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 5 folds using one class node.

```cpp
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);
    printf("Accuracy=%f\n", acc);
}
```

See the `DSL_validator` reference for more details.
Reference Manual
6 Reference Manual

For additional information, please refer to the header files included in your SMILE distribution - each class reference in this section begin with the header filename. The public members of SMILE classes are generally placed first within the class declarations.

Note that you don't need to include the headers one-by-one in your program; the main header smile.h does that.

6.1 DSL_network

Header file: network.h

```
DSL_network();
DSL_network(const DSL_network &src);
~DSL_network();
```

Default constructor, copy constructor and destructor are implemented.

```
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:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSL_XDSL_FORMAT</td>
<td>xdsl</td>
<td>Native SMILE format</td>
</tr>
<tr>
<td>DSL_DSL_FORMAT</td>
<td>dsl</td>
<td>Old, deprecated SMILE format</td>
</tr>
<tr>
<td>DSL_HUGIN_FORMAT</td>
<td>net</td>
<td>Hugin</td>
</tr>
<tr>
<td>DSL_NETICA_FORMAT</td>
<td>dne</td>
<td>Netica</td>
</tr>
</tbody>
</table>
Returns DSL_OKAY on success or negative error code otherwise. If the reason for failure is syntax error in the file, or the file is not found, the error message is logged to DSL_errorH().

```c
int WriteFile(const char *filename, int fileType = 0,
        void *reserved = NULL);
```

Writes the network contents to the file system.

```c
int ReadString(const char *xdslString, void *reserved = NULL);
```

Read the network content from xdslString, only XDSL format is supported.

```c
int WriteString(std::string &xdslOutputString,
        void *reserved = NULL);
```

Writes the network content to xdslOutputString as XDSL.

```c
int UpdateBeliefs();
```

Executes inference algorithm to update node values. Returns DSL_OKAY on success or negative integer code on error.

```c
void SetDefaultBNAlgorithm(int algorithm);
```

Sets the algorithm used for inference in Bayesian networks. Available algorithm identifiers are:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Exact?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSL_ALG_BN_LAURITZEN</td>
<td>Yes</td>
<td>Clustering (default algorithm)</td>
</tr>
</tbody>
</table>
```c
void SetDefaultIDAlgorithm(int algorithm);
```

Sets the algorithm used for inference in influence diagrams. Available algorithm identifiers are:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSL_ALG_ID_COOPERSOLVING</td>
<td>Policy evaluation</td>
</tr>
<tr>
<td>DSL_ALG_ID_SHACHTER</td>
<td>Best policy search</td>
</tr>
</tbody>
</table>

```c
int InvalidateAllBeliefs();
```

Invalidates values in all of network’s nodes. Returns DSL_OKAY on success or negative integer code on error.

```c
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 modified jointree 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 ClearAllEvidence();
Clears all evidence in the network.

DSL_node* GetNode(int handle);
Returns a pointer to the node identifier by handle, NULL otherwise.

int FindNode(const char *nodeId) const;
Returns a handle for the node with identifier equal to nodeId, negative integer 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 by calling GetFirstNode followed by GetNextNode in the loop. If the network is empty, returns DSL_OUT_OF_RANGE.

int GetNumberOfNodes() const;
Returns the number of nodes in the network.

int AddNode(int nodeType, const char *nodeId);
Creates new node and returns its handle. If nodeId is NULL, the unique identifier will be created by the network. Supported node types are:

| DSL_CPT       | Conditional probability table |
If input parameters are invalid, the function returns negative status code.

```c
int DeleteNode(int handle);
```
Deletes the node specified by handle. Returns DSL_OKAY or negative status code.

```c
int DeleteAllNodes();
```
Deletes all nodes in the network.

```c
int AddArc(int parentHandle, int childHandle,
            dsl_arcType layer = dsl_normalArc);
```
Adds an arc between nodes specified by parentHandle and childHandle. Returns DSL_OKAY on success or negative status code on error. If adding an arc would result in cycle in the network, the status code is DSL_CYCLE_DETECTED.

```c
int RemoveArc(int parentHandle, int childHandle,
              dsl_arcType layer = dsl_normalArc);
```
Removes an arc. Returns DSL_OKAY on success or negative status code on error.

```c
const DSL_intArray& GetParents(int nodeHandle,
                               dsl_arcType layer = dsl_normalArc) const;
```
Returns a reference to the DSL_intArray containing handles of parent nodes. If nodeHandle is invalid the return value is undefined.

\[
\text{const DSL_intArray& GetChildren(int nodeHandle,}
\text{ dsl_arcType layer = dsl_normalArc) const;}
\]

Returns a reference to the DSL_intArray containing handles of children nodes. If nodeHandle is invalid the return value is undefined.

\[
\text{int ReverseArc(int parentHandle, int childHandle);}\]

Reverses an arc, preserving the joint probability distribution in the network. Returns DSL_OKAY on success, negative status code otherwise.

\[
\text{int IsArcNecessary(int parentHandle, int childHandle,}
\text{ double epsilon, bool &necessary) const;}
\]

Checks if the arc between parentHandle and childHandle is necessary. The arc is considered to be necessary when child’s conditional 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. The return value of the function is DSL_OKAY on success or negative status code otherwise.

\[
\text{int RemoveAllArcs();}
\]

Removes all arcs in the network.

\[
\text{int MarginalizeNode(int nodeHandle, DSL_progress *progress = NULL);}\]

Removes node specified by nodeHandle, but preserves the joint probability distribution over the remaining nodes. Returns DSL_OKAY on success or negative status code otherwise. The marginalization may take 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. If cancelled, the function returns DSL_INTERRUPTED.

\[
\text{int MakeUniform();}
\]

Uniformizes all node definitions in the network.

\[
\text{int GetNumberOfTargets() const;}
\]
Returns the number of target nodes in the network.

```c
int IsTarget(int nodeHandle);
```
For valid handle, returns nonzero if node was set as target, zero otherwise. If handle is invalid, it returns a negative status code.

```c
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).

```c
int ClearAllTargets();
```
Resets the target flags on all nodes.

```c
DSL_simpleCase* AddCase(const std::string & name);
```
Adds new case with specified name. Returns the pointer to the newly created case.

```c
DSL_simpleCase* GetCase(int index) const;
```
Returns the pointer to the case specified by the zero-based case index. If index is negative or greater or equal to the number of cases, returns NULL.

```c
DSL_simpleCase* GetCase(const std::string & name) const;
```
Returns the pointer to the case specified by name. If the case is not found, returns NULL.

```c
int DeleteCase(int index);
```
Deletes the case specified by the index. Returns DSL_OKAY on success, negative status code otherwise.

```c
void DeleteAllCases();
```
Removes all cases from the network.

```c
int GetNumberOfCases() const;
```
Returns the number of cases defined for the network

```c
void EnableSyncCases(bool sync);
```
Enables or disables the 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.

```c
bool IsEnableSyncCases() const;
```
Returns true if case synchronization is enabled, false otherwise.

## 6.2 DSL_node

Header file: `node.h`

DSL_node objects are created and destroyed by their parent DSL_network.

```c
const char* GetId() const;
```
Returns a node identifier.

```c
int SetId(const char *newId);
```
Sets a node identifier. Returns DSL_OKAY on success or negative status code on failure.

```c
DSL_nodeInfo &Info();
```
Returns a reference to DSL_nodeInfo structure containing node attributes not directly related to calculations.
DSL_network *Network();

Returns a pointer to node’s network.

int Handle();

Returns a node handle.

DSL_nodeDefinition *Definition();

Returns a pointer to node’s definition.

DSL_nodeValue *Value();

Returns a pointer to node’s value.

### 6.3 DSL_nodeDefinition

Header file: nodedef.h

DSL_nodeDefinition objects are created and destroyed by their parent DSL_node.

DSL_network* Network();

int Handle() const;

virtual int GetType() const = 0;

virtual const char *GetTypeName() const = 0;
virtual int AddOutcome(const char *outcomeId);

virtual int InsertOutcome(int outcomeIndex, const char *outcomeId);

virtual int RemoveOutcome(int outcomeIndex);

virtual int GetNumberOfOutcomes();

virtual int RenameOutcome(int outcomeIndex, char *newId);

virtual int RenameOutcomes(DSL_stringArray &outcomeIds);

virtual DSL_idArray* GetOutcomesNames();

virtual int SetNumberOfOutcomes(int outcomeCount);

virtual int SetNumberOfOutcomes(DSL_stringArray &outcomeIds);
virtual int SetDefinition(DSL_doubleArray &def);

virtual int SetDefinition(DSL_intArray &def);

virtual int GetDefinition(DSL_doubleArray **def);

virtual int GetDefinition(DSL_Dmatrix **def);

DSL_Dmatrix* GetMatrix();

6.4 DSL_nodeValue

Header file: nodeval.h

DSL_nodeValue objects are created and destroyed by their parent DSL_node.

DSL_network *Network();

int Handle();

virtual int GetType() const = 0;
DSL_intArray& GetIndexingParents();

int IsValueValid();

virtual int SetValue(DSL_Dmatrix &val);

virtual int GetValue(DSL_Dmatrix **val);

DSL_Dmatrix* GetMatrix();

int IsEvidence() const;

int IsRealEvidence() const;

int IsPropagatedEvidence() const;

int IsVirtualEvidence() const;

virtual int GetEvidence() const;
virtual int GetEvidence(double &evidence) const;

virtual int SetEvidence(double evidence);

virtual int SetEvidence(int evidence);

virtual int ClearEvidence();

virtual int ClearPropagatedEvidence();

virtual int GetVirtualEvidence(std::vector<double> &evidence) const;

virtual int SetVirtualEvidence(const std::vector<double> &evidence);

## 6.5 DSL_Dmatrix

Header file: dmatrix.h

DSL_Dmatrix();

The default constructor, creates an empty matrix without any dimensions.

DSL_Dmatrix(const DSL_intArray &dimensions);
The constructor which creates the matrix with specified dimensions.

```c
DSL_Dmatrix(const DSL_Dmatrix &mtx);
```
Copy constructor.

```c
int operator=(const DSL_Dmatrix &mtx);
```
Assignment operator.

```c
void Swap(DSL_Dmatrix &mtx);
```
Swaps the two matrices.

```c
double &operator[](int index);
double operator[](int index) const;
double &Subscript(int index);
```
Access to the matrix elements with linear index.

```c
double &operator[](const DSL_intArray &coords);
double operator[](const DSL_intArray &coords) const;
double &Subscript(DSL_intArray &coords);
```
Access to the matrix elements using multidimensional coordinates.

```c
double &operator[](const int * const * indirectCoords);
```
Access to the matrix elements using indirect multidimensional coordinates.

```c
int IndexToCoordinates(int index, DSL_intArray &coords) const;
```
Converts the linear index to multidimensional coordinates. Returns DSL_OKAY on success, or an error code on failure.

```c
int CoordinatesToIndex(DSL_intArray &coords) const;
int CoordinatesToIndex(const int * const * indirectCoords) const;
```
Converts the multidimensional coordinates to linear index. Returns a converted index, or a negative error code on failure.

```c
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's no next element).

```c
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's no previous element).

```c
DSL_doubleArray& GetItems();
const DSL_doubleArray& GetItems() const;
```

Returns the reference the linear buffer containing the elements of the matrix.

```c
const DSL_intArray& GetDimensions() const;
```

Returns the dimensions of the matrix.

```c
const DSL_intArray& GetPreProduct() const;
```

Returns the reference to the preproduct array, which is useful for converting between linear and multidimensional coordinates.

```c
int GetNumberOfDimensions() const;
```

Returns the number of dimensions of the matrix.

```c
int GetLastDimension() const;
```

Returns the index of the last dimension of the matrix (not the size of the last dimension)
int GetSizeOfDimension(int dimIdx) const;
Returns the size of the specified matrix dimension.

int GetSize() const;
Returns the total size of the matrix (which is a product of the 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 an error is returned. 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 an error is returned. 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 an error is returned. 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 \((a,b,c,\ldots,z=N)\) by the Nth element of vector v.

int FillWith(double x);
Fills the matrix with a specified scalar value.
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.

6.6 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 dataset from the text file. Returns DSL_OKAY on success or an error code on failure. If errOut is not NULL, the 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 dataset variables are determined as:

- if the data column in the file contains non-numeric entries, the corresponding dataset variable is string discrete

- if the data column in the file contains only numeric entries and there's at least one fractional value, the corresponding dataset variable is numeric continuous

- otherwise the dataset 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:

```c
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;
};
```

```c
int WriteFile(const std::string &filename,
             const DSL_datasetWriteParams *params = NULL,
             std::string *errOut = NULL) const;
```

Writes the contents of the dataset to the text file. Returns DSL_OKAY on success or an error code on failure. If errOut is not NULL, the 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:

```c
struct DSL_datasetWriteParams
{
    DSL_datasetWriteParams() :
        missingValueToken("*") ,
        columnIdsPresent(true) ,
        useStateIndices(false) ,
```

```c
```
SMILE: Structural Modeling, Inference, and Learning Engine

Reference Manual

int MatchNetwork(const DSL_network &net,
                 std::vector<DSL_datasetMatch> &matching,
                 std::string &errMsg);

Attempts to match the contents of the dataset to the structure of the network specified as
first argument (typically before parameter learning or network validation). May change the
integer indices in the dataset 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 network and data, at least one node
and dataset variable have to have identical identifier, and

- either both node and dataset variable are continuous, or

- both node and dataset variable are discrete, and all values in the dataset variable can be
  mapped onto node outcomes

When the network and the dataset cannot be matched, the error code is returned and
additional human-readable information about the error is written to errMsg parameter.

int AddIntVar(const std::string id = std::string(),
              int missingValue = DSL_MISSING_INT);

Adds discrete integer variable to the dataset. 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 dataset. Returns DSL_OKAY on success or error
code on failure.
Multiple variables with empty identifiers are allowed.

```cpp
int RemoveVar(int var);
```
Removes a variable from the dataset. Returns DSL_OKAY on success or error code on failure.

```cpp
void AddEmptyRecord();
```
Appends a record with all values missing.

```cpp
void SetNumberOfRecords(int numRecords);
```
Sets the number of records in the dataset. If the new number is greater than current, new records will have all values missing.

```cpp
int RemoveRecord(int rec);
```
Removes the specified record from the dataset. Returns DSL_OKAY on success or error code on failure.

```cpp
int FindVariable(const std::string &id) const;
```
Returns the index of the variable with the specified identifier, or a negative error code on failure.

```cpp
int GetNumberOfVariables() const;
```
Returns the number of variables in the dataset.

```cpp
int GetNumberOfRecords() const;
```
Returns the number of records in the dataset.

```cpp
int GetInt(int var, int rec) const;
```
Returns the integer data value in the specified variable and record.

```cpp
float GetFloat(int var, int rec) const;
```
Returns the floating data value in the specified variable and record.

\[
\text{void SetInt(int var, int rec, int value);} \\
\text{Sets the integer data value in the specified variable and record.}
\]

\[
\text{void SetFloat(int var, int rec, float value);} \\
\text{Sets the floating data value in the specified variable and record.}
\]

\[
\text{void SetMissing(int var, int rec);} \\
\text{Marks the data element in the specified variable and record as missing.}
\]

\[
\text{bool IsMissing(int var, int rec) const;} \\
\text{Returns true if the data element in the specified variable and record is missing.}
\]

\[
\text{int GetMissingInt(int var) const;} \\
\text{Returns the integer value representing missing data in the specified discrete variable.}
\]

\[
\text{float GetMissingFloat(int var) const;} \\
\text{Returns the floatvalue representing missing data in the specified continuous variable.}
\]

\[
\text{bool IsDiscrete(int var) const;} \\
\text{Returns true if specified variable is discrete.}
\]

\[
\text{enum DiscretizeAlgorithm \{ Hierarchical, UniformWidth, UniformCount \};} \\
\text{int Discretize(int var, DiscretizeAlgorithm alg, int intervals,} \\
\text{ const std::string &statePrefix, std::vector<double> &edges);} \\
\text{int Discretize(int var, DiscretizeAlgorithm alg, int intervals,} \\
\text{ const std::string &statePrefix);} \\
\text{Discretizes a dataset variable. Returns DSL_OKAY on success or error code on failure. The first} \\
\text{overload also returns the values of discretization interval edges.}
\]
6.7 **DSL_dataGenerator**

Header file: `datagenerator.h`

```c
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.

```c
int GenerateData(DSL_dataset &ds);
```

Generate data and store the results in the `DSL_dataset`

```c
int GenerateData(const char *filename,
                 const DSL_datasetWriteParams *params = NULL);
```

Generate data and write the results to the text file. To fine tune the output format pass the pointer to the `DSL_datasetWriteParams` object.

```c
int GenerateData(DSL_dataGeneratorOutput &out);
```

Generate data and write the results to the 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.

```c
void SetNumberOfRecords(int numrec);
int GetNumberOfRecords() const;
```

Set/get the number of records to generate

```c
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.

```c
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.

6.8 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 the dataset, the network and the 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 dataset 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 the K-fold crossvalidagation using the dataset 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 datasets 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 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 LeaveOneOut(DSL_em &em, DSL_progress *progress = 0);
```

Performs the Leave-one-out crossvalidation using the dataset 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 dataset with the index specified by the recordIndex.

Returns DSL_OKAY on success or an error code on failure.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
int GetPredictedNode(int recordIndex) const;
int GetPredictedNodeIndex(int recordIndex) const;
```

Gets the predicted node handle or index for the specified record index. The node prediction is based on the probabilities of most likely outcomes in class nodes. Returns DSL_OKAY on success or an error code on failure.

```cpp
int GetFoldIndex(int recordIndex) const;
```

Gets the fold to which the specified dataset record belongs. Returns DSL_OKAY on success or an error code on failure.

```cpp
void GetResultDataset(DSL_dataset &output) const;
```

Fills the output dataset with the content of the input dataset (specified in the constructor) and calculated class node probabilities and predicted outcomes.

```cpp
int CreateROC(int classNodeHandle, int outcomeIndex,
              std::vector<std::pair<double, double> > &curve,
```
std::vector<double> &thresholds) 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.

6.9  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 in the specified network using data from the dataset using the EM algorithm. Returns DSL_OKAY on success or an error code on failure.

Network nodes and dataset 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 \texttt{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 \texttt{progress} can be used to stop the learning by returning false from \texttt{DSL\_progress::Tick} method, which is called periodically within the main loop of the learning algorithm. In such case, the \texttt{Learn} method returns DSL\_INTERRUPTED.

\begin{verbatim}
void SetRandomizeParameters(bool r);
bool GetRandomizeParameters() const;
\end{verbatim}
Set/get the value of 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.

\begin{verbatim}
void SetUniformizeParameters(bool u);
bool GetUniformizeParameters() const;
\end{verbatim}
Set/get the value of 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.

\begin{verbatim}
void SetSeed(int seed);
int GetSeed() const;
\end{verbatim}
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.

\begin{verbatim}
void SetEquivalentSampleSize(float eqs);
float GetEquivalentSampleSize() const;
\end{verbatim}
Set/get the equivalent sample size. The equivalent sample size, also known as confidence, which 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 the new cases, which gives a mechanism for gentle refinement of 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.

Defaults to 1. Call \texttt{SetRandomizeParameters(false)} and \texttt{SetUniformizeParameters(false)} if you want to use larger values as equivalent sample sizes.
6.10  DSL_bs

Header file: bs.h

```c
DSL_bs();
```

The default constructor.

```c
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 network structure using Bayesian Search algorithm, then learns the parameters with EM using the specified dataset. Each variable in the dataset 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 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.

The optional argument `eval` may be used to provide an alternative structure evaluator.

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.

```c
int nrIteration;
```

Number of iterations to perform in the main structure learning loop. Each iteration starts with random structure and is refined until convergence. Defaults to 20.

```c
int maxParents;
```

Maximum number of parents in the learned network. Defaults to 5.

```c
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 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 starting 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 over all edges. It comes into the formula in the following way:

\[
\log \text{Posterior score} = \log \text{marginal likelihood (i.e., the BDeu)} + \mid \text{parents} \mid \log (\text{pll}) + (\mid \text{nodes} \mid - \mid \text{parents} \mid - 1) \log (1-\text{pll})
\]

Defaults to 0.001.

DSL_bkgndKnowledge bkk;
Background knowledge used to constrain the network structures created by structure learning algorithm. Empty by default.

6.11 DSL_pc

Header file: pc.h

DSL_pc();
The default constructor.

int Learn(const DSL_dataset &ds, DSL_pattern &pat,
Based on the specified dataset, creates a graph using PC algorithm and stores the graph edges in the specified DSL_pattern. Each variable in the dataset 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 case, the Learn method returns DSL_INTERRUPTED.

---

**int maxAdjacency;**

Maximum number of neighbours of a node. 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;**

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.

---

**6.12 DSL_tan**

Header file: tan.h

---

**DSL_tan();**

The default constructor.
int Learn(DSL_dataset &ds, DSL_network &net, 
        DSL_progress *progress = NULL) const;

Creates network structure using Tree Augmented Naive Bayes (TAN) algorithm, then learns the parameters with EM using the specified dataset. Each variable in the dataset 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 case, the Learn method returns DSL_INTERRUPTED.

std::string classvar;
Identifier of the class variable.

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 initialized the random generator. Defaults to zero, which causes the value based on system clock to be used as seed.

6.13 DSL_abn

Header file: abn.h

DSL_abn();
The default constructor.

int Learn(DSL_dataset &ds, DSL_network &net,
**DSL_progress *progress = NULL) const;**

Creates network structure using Augmented Naive Bayes (ABN) algorithm, then learns the parameters with EM using the specified dataset. Each variable in the dataset 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 case, the Learn method returns DSL_INTERRUPTED.

```c++
std::string classvar;
```

Identifier of the class variable.

```c++
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.

```c++
int maxParents;
int maxSearchTime;
int nrIteration;
double linkProbability;
double priorLinkProbability;
int priorSampleSize;
int seed;
```

See the *DSL_bs* reference.

### 6.14 DSL_nb

Header file: *nb.h*

```c++
DSL_nb();
```

The default constructor.
int Learn(DSL_dataset &ds, DSL_network &net, 
        DSL_progress *progress = NULL) const;

Creates the naive Bayes network, then learns the parameters with EM using the specified dataset. Returns DSL_OKAY on success or an error code on failure.

The structure of a 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.

### 6.15 DSL_bkgndKnowledge

Header file: bkgndknowledge.h

typedef std::vector<std::pair<int, int> > IntPairVector;

Type defined for convenience.

IntPairVector forcedArcs;

Arcs which are required to be present in the learned network structure

IntPairVector forbiddenArcs;

Arcs which 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.

6.16 DSL_pattern

Header file: pattern.h

enum EdgeType {None, Undirected, Directed};

Edge type identifiers

void SetSize(int size);
int GetSize() const;

Sets/gets 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's 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's 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;
SMILE: Structural Modeling, Inference, and Learning Engine

Reference Manual

Returns true if the pattern is a directed acyclic graph - there are no cycles and all edges are directed.

bool ToDAG();
Attempts the conversion of pattern to a directed acyclic graph. Returns true if successful.

bool ToNetwork(const DSL_dataset &ds, DSL_network &net);
Attempts the conversion of pattern to a directed acyclic graph. If successful, creates DSL_network with node identifiers and outcomes based on the specified dataset. It is assumed that the dataset was used previously to obtain this pattern, therefore the order of variables in the dataset 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's 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.

6.17  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, which must be overriden 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 can't 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.
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.

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

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

To run all tutorials you can use the main.cpp file listed below. This approach requires compiling and linking main.cpp AND all of the tutorialN.cpp files.

7.1 main.cpp

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

#include <cstdio>

// smile_license.h is your personal license key file
#include "smile_license.h"

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

int main()
{
    static int (* const f[])() =
    {
        Tutorial1, Tutorial2, Tutorial3, Tutorial4, Tutorial5
    };
    for (int i = 0; i < sizeof(f) / sizeof(f[0]); i++)
    {
        int r = f[i]();
        if (r)
        {
            return r;
        }
        printf("----------------
");
    }
    printf("\nALL TUTORIALS COMPLETE.\n");
    return 0;
```
7.2 Tutorial 1: Creating a Bayesian Network

Consider a slight twist on the problem described in the Hello SMILE! 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 predictions. 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.

```c
DSL_errorH().RedirectToFile(stdout);
DSL_network net;

const char *ECONOMY_OUTCOMES[] = {"Up","Flat","Down",NULL};
int e = CreateCptNode(net, "Economy", "State of the economy", 
    ECONOMY_OUTCOMES, 160, 40);

const char *SUCCESS_OUTCOMES[] = {"Success","Failure",NULL};
int s = CreateCptNode(net, "Success", "Success of the venture", 
    SUCCESS_OUTCOMES, 60, 40);

const char *FORECAST_OUTCOMES[] = {"Good","Moderate","Poor",NULL};
int f = CreateCptNode(net, "Forecast", "Expert forecast", 
    FORECAST_OUTCOMES, 110, 140);
```
Before connecting the nodes with arcs, let’s have a look at the CreateCptNode function.

```cpp
static int CreateCptNode(DSL_network &net,
                        const char *id, const char *name,
                        const char *outcomes[], int xPos, int yPos)
{
    int handle = net.AddNode(DSL_CPT, id);
    DSL_node *node = net.GetNode(handle);

    DSL_idArray ida;
    for (const char **p = outcomes; *p != NULL; p++)
    {
        ida.Add(*p);
    }
    DSL_nodeDefinition *def = node->Definition();
    def->SetNumberOfOutcomes(ida);
    node->Info().Header().SetName(name);
    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 specified identifier, name, outcomes and position on the screen. The handle returned by `AddNode` is converted to `DSL_node*` pointer with a call to `GetNode`. CPT nodes are created with two outcomes named `State0` and `State1`. To change the number of node outcomes and rename them, we will call the `DSL_nodeDefinition::SetNumberOfOutcomes` (we could also use `DSL_nodeDefinition::AddOutcome` or `DSL_nodeDefinition::InsertOutcome`). The single parameter passed to `SetNumberOfOutcomes` is a reference to `DSL_idArray` object containing the outcome identifier. After outcomes, we set node’s name and position. Function returns the handle of the node it created.

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

```cpp
    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 section in this manual for the description of the CPT memory layout. For each of three nodes in our network we obtain a pointer to node definition object. To change the numbers (probabilities) in the CPT we will call `DSL_nodeDefinition::SetDefinition` method and pass a reference to a `DSL_doubleArray` as its single argument. When constructing `DSL_doubleArray` object we initialize its size to be equal to the CPT size. Next, we fill the array with probabilities. The code shown below for the `Success` node. Two other nodes are initialized in the same way.

```cpp
    DSL_nodeDefinition *successDef = net.GetNode(s)->Definition();
```
DSL_doubleArray sp(successDef->GetMatrix()->GetSize());
sp[0] = 0.3; // P(Success=S|Economy=U)
sp[1] = 0.7; // P(Success=F|Economy=U)
sp[2] = 0.2; // P(Success=S|Economy=F)
sp[3] = 0.8; // P(Success=F|Economy=F)
sp[4] = 0.1; // P(Success=S|Economy=D)
sp[5] = 0.9; // P(Success=F|Economy=D)
res = successDef->SetDefinition(sp);
if (DSL_OKAY != res)
{
    return res;
}
res = successDef->SetDefinition(sp);

With CPTs initialized our network is complete. We write its contents to the tutorial1.xdsl file. Tutorial 2 will load this file and perform the inference. The split between tutorials is artificial, your program can use networks right after its creation without the need to write/read from the file system.

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

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, 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;

    const char *ECONOMY_OUTCOMES[] = { "Up", "Flat", "Down", NULL }; int e = CreateCptNode(net, "Economy", "State of the economy", ECONOMY_OUTCOMES, 160, 40);

    const char *SUCCESS_OUTCOMES[] = { "Success", "Failure", NULL }; int s = CreateCptNode(net, "Success", "Success of the venture", SUCCESS_OUTCOMES, 60, 40);

    const char *FORECAST_OUTCOMES[] = { "Good", "Moderate", "Poor", NULL }; int f = CreateCptNode(net, "Forecast", "Expert forecast", FORECAST_OUTCOMES, 110, 140);
net.AddArc(e, s);
net.AddArc(s, f);
net.AddArc(e, f);

DSL_nodeDefinition *economyDef = net.GetNode(e)->Definition();
DSL_doubleArray ep(economyDef->GetMatrix()->GetSize);
ep[0] = 0.2; // P(Economy=U)
ep[1] = 0.7; // P(Economy=F)
ep[2] = 0.1; // P(Economy=D)
int res = economyDef->SetDefinition(ep);
if (DSL_OKAY != res)
{
    return res;
}

DSL_nodeDefinition *successDef = net.GetNode(s)->Definition();
DSL_doubleArray sp(successDef->GetMatrix()->GetSize);
sp[0] = 0.3; // P(Success=S|Economy=U)
sp[1] = 0.7; // P(Success=F|Economy=U)
sp[2] = 0.2; // P(Success=S|Economy=F)
sp[3] = 0.8; // P(Success=F|Economy=F)
sp[4] = 0.1; // P(Success=S|Economy=D)
sp[5] = 0.9; // P(Success=F|Economy=D)
res = successDef->SetDefinition(sp);
if (DSL_OKAY != res)
{
    return res;
}

DSL_nodeDefinition *forecastDef = net.GetNode(f)->Definition();
DSL_doubleArray fp(forecastDef->GetMatrix()->GetSize);
fp[0] = 0.70; // P(Forecast=G|Success=S,Economy=U)
fp[1] = 0.29; // P(Forecast=M|Success=S,Economy=U)
fp[2] = 0.01; // P(Forecast=P|Success=S,Economy=U)
fp[3] = 0.65; // P(Forecast=G|Success=S,Economy=F)
fp[4] = 0.30; // P(Forecast=M|Success=S,Economy=F)
fp[5] = 0.05; // P(Forecast=P|Success=S,Economy=F)
fp[6] = 0.60; // P(Forecast=G|Success=S,Economy=D)
fp[7] = 0.30; // P(Forecast=M|Success=S,Economy=D)
fp[8] = 0.10; // P(Forecast=P|Success=S,Economy=D)
fp[9] = 0.15; // P(Forecast=G|Success=F,Economy=U)
fp[10] = 0.30; // P(Forecast=M|Success=F,Economy=U)
fp[11] = 0.55; // P(Forecast=P|Success=F,Economy=U)
fp[12] = 0.10; // P(Forecast=G|Success=F,Economy=F)
fp[13] = 0.30; // P(Forecast=M|Success=F,Economy=F)
fp[14] = 0.60; // P(Forecast=P|Success=F,Economy=F)
fp[15] = 0.05; // P(Forecast=G|Success=F,Economy=D)
fp[16] = 0.25; // P(Forecast=G|Success=F,Economy=D)
fp[17] = 0.70; // P(Forecast=G|Success=F,Economy=D)
res = forecastDef->SetDefinition(fp);
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,
    const char *outcomes[], int xPos, int yPos)
{
    int handle = net.AddNode(DSL_CPT, id);
    DSL_node *node = net.GetNode(handle);

    DSL_idArray ida;
    for (const char **p = outcomes; *p != NULL; p++)
    {
        ida.Add(*p);
    }
    DSL_nodeDefinition *def = node->Definition();
    def->SetNumberOfOutcomes(ida);

    node->Info().Header().SetName(name);

    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 we have previously created. We will perform multiple calls to Bayesian inference algorithm through the DSL_network::UpdateBeliefs, starting with network without any evidence. After each UpdateBeliefs call the posterior probabilities of nodes will be displayed.

The Tutorial12 function itself is very simple and delegates work to helper functions declared at the top of the file and defined below Tutorial12.
The model is loaded with the DSL_network::ReadFile. We exit the program if the status code ReadFile returned is not DSL_OKAY.

```cpp
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.

```cpp
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 the nodes, DSL_network::GetFirstNode and GetNextNode are used. In the body of the loop we call another locally defined helper function, PrintPosteriors.

```cpp
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 node has evidence set; 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.

```cpp
static void PrintPosteriors(DSL_network &net, int handle)
{
    DSL_node *node = net.GetNode(handle);
    const char *nodeId = node->GetId();
    const DSL_idArray &outcomeIds
        = *node->Definition()->GetOutcomesNames();
    DSL_nodeValue *val = node->Value();
    if (val->IsEvidence())
    {
        printf("%s has evidence set (%s)\n", nodeId, outcomeIds[val->GetEvidence()]);
    }
    else
    {
        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:

```cpp
printf("\nSetting Forecast=Good.\n");
```
Let us examine the `ChangeEvidenceAndUpdate` helper.

```c
static int ChangeEvidenceAndUpdate(DSL_network &net,
    const char *nodeId, const char *outcomeId)
{
    int res = SetEvidenceById(net, nodeId, outcomeId);
    if (DSL_OKAY != res)
    {
        return res;
    }
    res = net.UpdateBeliefs();
    if (DSL_OKAY != res)
    {
        return res;
    }
    PrintAllPosteriors(net);
    return DSL_OKAY;
}
```

It is just a series of calls to `SetEvidenceById`, `DSL_network::UpdateBeliefs` and `PrintAllPosteriors`. Let us check the `SetEvidenceById` helper now.

```c
static int SetEvidenceById(DSL_network &net,
    const char *nodeId, const char *outcomeId)
{
    int handle = net.FindNode(nodeId);
    if (handle < 0)
    {
        return handle;
    }
    DSL_node *node = net.GetNode(handle);
    if (NULL == outcomeId)
    {
        return node->Value()->ClearEvidence();
    }
    else
    {
        DSL_nodeDefinition *def = node->Definition();
        int idx = def->GetOutcomesNames()->FindPosition(outcomeId);
        if (idx < 0)
        {
            return idx;
        }
        return node->Value()->SetEvidence(idx);
    }
}
```
The function converts human-readable node identifier passed as its 2nd argument to a node handle with a call to DSL_network::FindNode. If the 3rd argument (the outcome identifier) is NULL, evidence for the node is cleared. Otherwise the identifier is converted to its index; the index is used to set the evidence with DSL_nodeValue::SetEvidence.

7.3.1 tutorial2.cpp

```c++
#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...
");
    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 int SetEvidenceById(
    DSL_network &net, const char *nodeId, const char *outcomeId)
{
    int handle = net.FindNode(nodeId);
    if (handle < 0)
    {
        return handle;
    }

    DSL_node *node = net.GetNode(handle);
    if (NULL == outcomeId)
    {
        return node->Value()->ClearEvidence();
    }
    else
    {
        int idx =
            node->Definition()->GetOutcomesNames()->FindPosition(outcomeId);
        if (idx < 0)
        {
            return idx;
        }
        return node->Value()->SetEvidence(idx);
    }
}

static void PrintPosteriors(DSL_network &net, int handle)
{
    DSL_node *node = net.GetNode(handle);
    const char *nodeId = node->GetId();
    const DSL_idArray &outcomeIds = *node->Definition()->GetOutcomesNames();
    DSL_nodeValue *val = node->Value();
    if (val->IsEvidence())
    {
        printf("%s has evidence set (%s)\n", nodeId, outcomeIds[val->GetEvidence()]);
    }
    else
    {
        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))
static int ChangeEvidenceAndUpdate(DSL_network &net, const char *nodeId, const char *outcomeId)
{
    int res = SetEvidenceById(net, nodeId, outcomeId);
    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 the locally defined helper function PrintNodeInfo. This is where real work is done. The first node attribute displayed is its name, stored in node header object.

    DSL_node *node = net.GetNode(nodeHandle);
    printf("Node: %s\n", node->Info().Header().GetName());

Identifiers of the outcomes are next, retrieved from node definition

    printf("  Outcomes: ");
    const DSL_idArray &outcomes = *node->Definition()->GetOutcomesNames();
    for (int i = 0; i < outcomes.NumItems(); i++)
    {
        printf(" %s", outcomes[i]);
    }

The parents of the node follow.

    const DSL_intArray &parents = net.GetParents(nodeHandle);
    if (parents.NumItems() > 0)
    {
        printf("  Parents: ");
        for (int i = 0; i < parents.NumItems(); i++)
        {
            printf("  %s", net.GetNode(parents[i])->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 node’s definition with the DSL_nodeDefinition::GetMatrix. Note that all the nodes in the network created in Tutorial 1[85] are DSL_CPT. The if statement is there to keep the function general enough to be copied and pasted into other program using SMILE.

```cpp
DSL_nodeDefinition *def = node->Definition();
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 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:

```cpp
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->Definition()->GetOutcomesNames();
            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 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 CoordinatesToIndex. While not significant for this tutorial, the coordinate-based loop performance over large CPTs is better.
when NextCoordinates approach is used (NextCoordinates is more economical than IndexToCoordinates). Of course, the plain linear index will be even faster.

The node outcome is the rightmost entry in the coordinates. The parents’ outcome indexes 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

\[
\begin{align*}
P(\text{Good} \mid \text{Success}=\text{Success}, \text{Economy}=\text{Up}) &= 0.7 \\
P(\text{Moderate} \mid \text{Success}=\text{Success}, \text{Economy}=\text{Up}) &= 0.29 \\
P(\text{Poor} \mid \text{Success}=\text{Success}, \text{Economy}=\text{Up}) &= 0.01 \\
P(\text{Good} \mid \text{Success}=\text{Success}, \text{Economy}=\text{Flat}) &= 0.65 \\
P(\text{Moderate} \mid \text{Success}=\text{Success}, \text{Economy}=\text{Flat}) &= 0.3 \\
P(\text{Poor} \mid \text{Success}=\text{Success}, \text{Economy}=\text{Down}) &= 0.3 \\
P(\text{Good} \mid \text{Success}=\text{Success}, \text{Economy}=\text{Down}) &= 0.1 \\
P(\text{Poor} \mid \text{Success}=\text{Failure}, \text{Economy}=\text{Up}) &= 0.15 \\
P(\text{Moderate} \mid \text{Success}=\text{Failure}, \text{Economy}=\text{Up}) &= 0.3 \\
P(\text{Poor} \mid \text{Success}=\text{Failure}, \text{Economy}=\text{Up}) &= 0.55 \\
P(\text{Good} \mid \text{Success}=\text{Failure}, \text{Economy}=\text{Flat}) &= 0.1 \\
P(\text{Moderate} \mid \text{Success}=\text{Failure}, \text{Economy}=\text{Flat}) &= 0.3 \\
P(\text{Poor} \mid \text{Success}=\text{Failure}, \text{Economy}=\text{Down}) &= 0.6 \\
P(\text{Good} \mid \text{Success}=\text{Failure}, \text{Economy}=\text{Down}) &= 0.05 \\
P(\text{Moderate} \mid \text{Success}=\text{Failure}, \text{Economy}=\text{Down}) &= 0.25 \\
P(\text{Poor} \mid \text{Success}=\text{Failure}, \text{Economy}=\text{Down}) &= 0.7
\end{align*}
\]

### 7.4.1 tutorial3.cpp

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

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

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

    DSL_errorH().RedirectToFile(stdout);

    // load the network created by Tutorial1
    DSL_network net;
    int res = net.ReadFile("tutorial1.xdsl");
    if (DSL_OKAY != res)
    {
        printf("tutorial3.cpp");
```

```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.
```
"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, preceded 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->Definition()->GetOutcomesNames();
                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->Info().Header().GetName());
    printf(" Outcomes:\n");
    const DSL_idArray &outcomes = *node->Definition()->GetOutcomesNames();
    for (int i = 0; i < outcomes.NumItems(); i++)
    {
        printf(" %s", outcomes[i]);
    }
    printf("\n");

    const DSL_intArray &parents = net.GetParents(nodeHandle);
    if (parents.NumItems() > 0)
    {
        printf(" Parents:\n");
        for (int i = 0; i < parents.NumItems(); i++)
        {
            printf(" %s", net.GetNode(parents[i])->GetId());
        }
        printf("\n");
    }

    const DSL_intArray &children = net.GetChildren(nodeHandle);
    if (children.NumItems() > 0)
    {
        printf(" Children:\n");
        for (int i = 0; i < parents.NumItems(); i++)
        {
            printf(" %s", net.GetNode(parents[i])->GetId());
        }
        printf("\n");
    }

    DSL_nodeDefinition *def = node->Definition();
    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 the Influence Diagram

We will further expand the model created in Tutorial 1 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 or 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 program starts by reading the file, just like Tutorial 2 and Tutorial 3. We convert the identifier 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 slightly modified version of the CreateCptNode from Tutorial 1. 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 for the value of its outcomes parameter and if it is NULL, the call to DSL_nodeDefinition::SetNumberOfOutcomes is skipped.

```c
static int CreateNode(DSL_network &net, int nodeType, const char *id, const char *name, const char *outcomes[], int xPos, int yPos)
{
    int handle = net.AddNode(nodeType, id);
    DSL_node *node = net.GetNode(handle);
    if (NULL != outcomes)
        /... the rest of the function is unchanged
}
```

Back in Tutorial 4 function, we add nodes and arcs:

```c
const char *INVEST_DECISIONS[] = { "Invest", "DoNotInvest", NULL }; int i = CreateNode(net, DSL_LIST, "Invest", "Investment decision", INVEST_DECISIONS, 160, 240);

int g = CreateNode(net, DSL_TABLE, "Gain", "Financial gain", NULL, 60, 200);
```
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 last dimension being always set to one (as there are no outcomes). Node Gain has two parents with two outcomes each and size of its definition is $2 \times 2 \times 1 = 4$. The program specifies four numbers for the utilities.

```c
DSL_nodeDefinition *gainDef = net.GetNode(g)->Definition();
DSL_doubleArray gu(gainDef->GetMatrix()->GetSize());
res = gainDef->SetDefinition(gu);
```

The influence diagram is now complete. We write its contents to file and exit the function. Tutorial 5 will load the file and perform the inference.

### 7.5.1 tutorial4.cpp

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

static int CreateNode(
    DSL_network &net, int nodeType, const char *id, const char *name,
    const char *outcomes[], int xPos, int yPos);

int Tutorial4()
{
    printf("Starting Tutorial4...
");
    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;
    }
```
const char *INVEST_DECISIONS[] = { "Invest", "DoNotInvest", NULL };  
int i = CreateNode(net, DSL_LIST, "Invest", "Investment decision",  
INVEST_DECISIONS, 160, 240);  

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

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

DSL_nodeDefinition *gainDef = net.GetNode(g)->Definition();  
DSL_doubleArray gu(gainDef->GetMatrix()->GetSize());  
res = gainDef->SetDefinition(gu);  
if (DSL_OKAY != res)  
{T  
  return res;  
}  

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,  
const char *outcomes[], int xPos, int yPos)  
{  
  int handle = net.AddNode(nodeType, id);  
  DSL_node *node = net.GetNode(handle);  

  if (NULL != outcomes)  
  {  
    DSL_nodeDefinition *def = node->Definition();  
    DSL_idArray ida;  
    for (const char **p = outcomes; *p != NULL; p++)  
    {  
      ida.Add(*p);  
    }  
    def->SetNumberOfOutcomes(ida);  
  }  

  node->Info().Header().SetName(name);  

  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 that we have created in Tutorial 4. 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 UpdateBeliefs for the first time, the model has no evidence. A local helper function, PrintFinancialGain, is called to print out the utilities.

```c
static void PrintFinancialGain(DSL_network &net, int gainHandle)
{
    DSL_node *node = net.GetNode(gainHandle);
    const char *nodeName = node->Info().Header().GetName();
    printf("%s:\n", nodeName);
    DSL_nodeValue *val = node->Value();
    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 (utilities) 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 slightly modified version of the function from Tutorial 3. The changes are needed to properly display utility matrix with its last dimension set to 1, as utility nodes have no outcomes.

The utilities without evidence suggest that we should not invest:

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

Next, we model the analyst’s forecast to be good by calling local helper function SetEvidenceById. It made its first appearance in Tutorial 3, we are using it unchanged here.

```c
SetEvidenceById(net, "Forecast", "Good");
if (DSL_OKAY != net.UpdateBeliefs())
{
    return res;
}
PrintFinancialGain(net, gain);
```

The utilities have changed; with good forecast the optimal decision is to invest:

- Financial gain:
  - Utility(Invest=Invest)=4455.78
Now we observe the state of the economy and conclude that it is growing.

```c
SetEvidenceById(net, "Economy", "Up");
if (DSL_OKAY != net.UpdateBeliefs())
{
    return res;
}
PrintFinancialGain(net, gain);
```

Growing economy makes our chances even better:

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

This concludes Tutorial 5.

### 7.6.1 tutorial5.cpp

```c
// 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 int SetEvidenceById(DSL_network &net, const char *nodeId, const char *outcomeId);
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\n");
        return res;
    }

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

    return res;
}
```
printf("No evidence set.\n");
if (DSL_OKAY != net.UpdateBeliefs())
{
    return res;
}
PrintFinancialGain(net, gain);

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

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

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

// PrintMatrix displays each probability entry in the matrix in the separate
// line, preceded 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->Definition()->GetOutcomesNames();
    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);
  const char *nodeName = node->Info().Header().GetName();
  printf("%s:\n", nodeName);
  DSL_nodeValue *val = node->Value();
  const DSL_Dmatrix &mtx = *val->GetMatrix();
  const DSL_intArray &parents = val->GetIndexingParents();
  PrintMatrix(net, mtx, "Utility", NULL, parents);
}

static int SetEvidenceById(DSL_network &net, const char *nodeId, const char *outcomeId)
{
  int handle = net.FindNode(nodeId);
  if (handle < 0)
  {
    return handle;
  }
  DSL_node *node = net.GetNode(handle);
  if (NULL == outcomeId)
  {
    return node->Value()->ClearEvidence();
  }
  else
  {
    int idx =
      node->Definition()->GetOutcomesNames()->FindPosition(outcomeId);
    if (idx < 0)
{   return idx;
}

return node->Value()->SetEvidence(idx);
}
Acknowledgments
Acknowledgments

8 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
Copyright (c) 2014, Kenneth MacKay
All rights reserved.
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Expat
Copyright (c) 1998-2000 Thai Open Source Software Center Ltd and Clark Cooper
Copyright (c) 2001-2017 Expat maintainers

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
This page is intentionally left blank.
Index

- A -
  acknowledgments  108

- C -
  compiler options  12, 13

- D -
  deployment license  10
  development license  10

- G -
  gcc  13

- L -
  LeaveOneOut  42
  license  10

- V -
  Visual C++  12