MSBNx: A Component-Centric Toolkit for Modeling and Inference with Bayesian Networks

Carl M. Kadie
David Hovel
Eric Horvitz

28 July 2001

Technical Report
MSR-TR-2001-67

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

MSBNx: A Component-Centric Toolkit for Modeling and Inference with Bayesian Networks

Carl M. Kadie David Hovel Eric Horvitz
Adaptive Systems and Interaction
Microsoft Research
One Microsoft Way
Redmond, WA 98052-6399
{carlk, davidhov, horvitz}@microsoft.com

Abstract

We review the functionality of a modular, component-based tool kit for Bayesian network development and inference. Beyond its operation as a standalone modeling and inference environment, MSBNx facilitates the development of standalone applications by providing a set of run-time components that provide Bayesian reasoning services when integrated into other programs. MSBNx inferential operations provide both inference about states of inference and about the value of information for unobservered evidence. The services and modeling environment supports both diagnostic and troubleshooting mingles observations and repair operations. MSBNx facilitates the development and use of new *add-in* components. The modeling environment provides a means for assessing distinctions and beliefs, and special interfaces and tools for representing the asymmetric nature of probability distributions.

1. Introduction

MSBNx is a component-based Windows application for creating, assessing, and evaluating Bayesian networks. MSBNx can provide diagnosis as well as troubleshooting inference that considers both observations and repair operations. Components of MSBNx can be integrated into programs allow them to leverage inference and decision making under uncertainty. When doing diagnosis and troubleshooting, MSBNx can recommend what evidence to gather next based on value of information (VOI) computations. Beyond using the components that MSBNx provides, developers and researchers can create their own new *add-in* components and use them within MSBNx.

For information on downloading MSBNx and restrictions on its use, see http://research.microsoft.com/adapt/MSBNx/.

The application's installation module includes complete help files and sample networks. Bayesian networks are encoded in an XML-based file format. The application and its components run on any Windows system version from Windows 98 to Windows XP. MSBNx tries to make it easy for you to specify your probabilities for a Bayesian network. With the Standard Assessment Tool, you can specify full and causally independent probability distributions. With the Asymmetric Assessment Tool, you can avoid specifying redundant probabilities. If you have sufficient data and use machine learning tools to create Bayesian networks, you can use MSBNx to edit and evaluate the results.

MSBNx is fully component based. Its most important component is MSBN3, an ActiveX DLL. MSBN3 offers an extensive COM-based API for editing and evaluating Bayesian networks. You'll find MSBN3 especially easy to use from COM-friendly languages such as Visual Basic and JScript. A comprehensive

manual, available at http://research.microsoft.com/adapt/MSBNx/ documents MSBN3. MSBNx also includes graphical components, for example, both the Standard Assessment and Asymmetric Assessment tools are ActiveX controls and can be used in other applications. Also, you can extend the editing and evaluation abilities of MSBNx by creating add-ins. For example, MSBNx ships with an add-in for editing and evaluating Hidden Markov Models.

This report starts with an introduction to tasks of diagnosis and troubleshooting and a review of the concepts and vocabulary of Bayesian Networks. It then outlines the features of MSBNx. Next, it describes key aspects of the toolkit, including the inference and modeling environment. Finally, the report illustrates with examples how other programs can use MSBNx's inference and modeling services.

1.1 Background on Diagnosis and Troubleshooting

This section provides background information about on the tasks of diagnosis and troubleshooting with Bayesian networks [Breese & Heckerman, 1996; Heckerman, Horvitz, & Nathwani, 1992; Heckerman & Breese, 1994a; Heckerman & Breese, 1994b; Horvitz, Breese, & Henrion, 1988].

Consider this example of troubleshooting from real life. When your car doesn't start, what should you do?

A) Check if your lights work

B) Replace your fuel pump and see if that helps

Choice A) seems wiser. Why? Because checking your lights tells you much about your car's battery, a likely cause of your problem. But that's not the whole reason to check your lights first. Just as important, you check your lights first because such a check is easy and inexpensive.

MSBNx is a tool for doing this kind of cost-benefit reasoning for diagnosis and troubleshooting.

Causes and Uncertainty: You start using MSBNx by creating a model of your system. First, you create a diagram showing what causes what. For example, in our car model, the battery power causes the lights to on. Battery power also causes the engine to turn over which, in turn, causes the car to start.

Second, you tell about the uncertainty in the system, giving your probabilities for various situations. Here are some examples:

- Before you tried to start your car, how likely would you find a bad fuel pump? In our car model we set this to 2%.
- If battery power is high, how likely are the lights to work? We set this to 99.9%

Third and finally, you can add cost information. In the car model, we set the cost of checking the lights at $1. We say that fuel pumps can't be tested, only replaced, which costs $100.

Your model of causes and effects is called Bayesian Network (or sometimes Bayes' Net or Belief Network).

Run It Backwards: Once you've created a model, you can use it for diagnosis and troubleshooting. MSBNx evaluates your causes-to-effects model by running it backwards from effects to causes. Given known effects, for example, evidence of the lights working, it computes the probability of causes such as a bad battery. It also computes the utility of further tests, for example, checking the gas gauge. Although, MSBNx recommends which test to do next, you can do tests in any order. As you enter new evidence, MSBNx efficiently recomputes the probability of possible causes and the recommended order of further tests.

1.2 Basics of Bayesian Inference and Bayesian Networks

This section provides background information about Bayesian Networks [Jensen, 1996; Pearl, 1988, Poole, 1998].

1.2.1 Motivation

Logic, both in mathematics and in common speech, relies on clear notions of truth and falsity. Information that is either true or false is known as Boolean logic. For example, consider a statement such as "Unless I turn the lights on, the room will be dark." It leaves no room for uncertainty.

However, what about daylight coming in the windows? Everyday life presents us with many situations in which the accumulation of evidence leads to a conclusion. For example, a small amount of oil underneath a car in your driveway may be no great cause for concern. You may remember that you recently changed the car's oil, spilling some on the driveway in the process. If so, the oil stain evidence may be unimportant; if not, you may have your car checked for an oil leak, or wait to see if another oil stain shows up underneath your car when you park it in a different area.

Bayesian probability theory is a branch of mathematical probability theory that allows one to model uncertainty about the world and outcomes of interest by combining common-sense knowledge and observational evidence.

1.2.2 Bayesian networks

A Bayesian network is:

- a set of variables,
- a graphical structure connecting the variables, and
- a set of conditional distributions.

A Bayesian network is commonly represented as a graph, which is a set of vertices and edges. The vertices, or nodes, represent the variables and the edges, or arcs, represent the conditional dependencies in the model. The absence of an arc between two variables indicates conditional independence; that is, there are no situations in which the probabilities of one of the variables depends directly upon the state of the other.

Construction of a Bayesian network follows a common set of guidelines:

- Include all variables that are important in modeling your system.
- Use causal knowledge to guide the connections made in the graph
- Use your prior knowledge to specify the conditional distributions.

Causal knowledge in this context means linking variables in the model in such a way that arcs lead from causes to effects.

1.2.3 What is a Discrete Variable?

A variable is an element of a probability model that can take on a set of different values that are mutually exclusive and exhaustive. Consider a medical experiment in which men and women of different ages are studied. Some of the relevant variables would be the sex of the participant, the age of the participant and the experimental result. The variable sex has only two possible values: male or female. The variable age, on the other hand, can take on many values.

While probability theory as a whole can handle variables of both types, MSBNx can only accept variables with a limited number of possible values. We call these *discrete variables*, and we call each of the set of possible values a state.

How would you handle the age variable in MSBNx? Most likely, you would create a variable where each state represents a range of years. For example:

- Child (0-18)
- Adult (18-45)
- Middle Aged (45-65)
- Senior (65+)

This process is known as discretization.

1.2.4 Relationships in a Bayesian Model

In probability theory, there is no *a priori* way of knowing which variables influence other variables. In general, the complete or joint probability distribution must be known to correctly perform inference. For a real-world model, the joint distribution is usually very large and cannot be directly stored on a computer.

One of the primary roles of a Bayesian model is to allow the model creator to use commonsense and real-world knowledge to eliminate needless complexity in the model. For example, a model builder would be likely to know that the time of day would not normally directly influence a car's oil leak. Any such influence would be based on other, more direct factors, such as temperature and driving conditions.

The method used to remove meaningless relationships in a Bayesian model is to explicitly declare the meaningful ones. After establishing all the variables in a model, you must deliberately associate

variables that cause changes in the system to those variables that they influence. Only those specified influences are considered.

These influences are represented by conditioning arcs between nodes. Each arc should represent a causal relationship between a temporal antecedent (known as the parent) and is later outcome (known as the child).

1.2.5 What is Inference?

Inference, or model evaluation, is the process of updating probabilities of outcomes based upon the relationships in the model and the evidence known about the situation at hand.

When a Bayesian model is actually used, the end user applies evidence about recent events or observations. This information is applied to the model by "instantiating" or "clamping" a variable to a state that is consistent with the observation. Then the mathematical mechanics are performed to update the probabilities of all the other variables that are connected to the variable representing the new evidence.

After inference, the updated probabilities reflect the new levels of belief in (or probabilities of) all possible outcomes coded in the model. These beliefs are mediated by the original assessment of belief performed by the author of the model.

The beliefs originally encoded in the model are known as prior probabilities, because they are entered before any evidence is known about the situation. The beliefs computed after evidence is entered are known as posterior probabilities, because they reflect the levels of belief computed in light of the new evidence.

1.2.6 Parents and Children: Introduction to Diagrams

To recap, every variable in the real world situation is represented by a Bayesian variable. Each such variable describes a set of states that represent all possible distinct situations for the variable.

Once the set of variables and their states are known, the next step is to define the causal relationships among them. For any variable, this means asking the questions:

- What other variables (if any) directly influence this variable?
- What other variables (if any) are directly influenced by this variable?

In a standard Bayesian network, each variable is represented by a colored ellipse; this graphical representation is called a node.

Each causal influence relationship is described by a line (or arc) connecting the influencing variable to the influenced variable. The influence arc has a terminating arrowhead pointing to the influenced variable.

The common terminology then is as follows:

- A node is a Bayesian variable.
- An arc connects a parent (influencing) node to a child (influenced) node.

1.2.7 Bayesian network Creation

To create a Bayesian network, then, the following steps are necessary.

1. Create a set of variables representing the distinct elements of the situation being modeled.
2. For each such variable, define the set of outcomes or states that each can have. This set is referred to in the mathematical literature as "mutually exclusive and exhaustive," meaning that it must cover all possibilities for the variable, and that no important distinctions are shared between states.
3. Establish the causal dependency relationships between the variables. This involves creating arcs (lines with arrowheads) leading from the parent variable to the child variable.
4. Assess the prior probabilities. This means supplying the model with numeric probabilities for each variable in light of the number of parents the variable was given in Step 3.

1.2.8 Running Inference

After the model has been developed, MSBNx allows you to model or mimic real-world situations by entering or removing evidence on the model. As you do so, there are various methods for displaying the resulting probabilities.

1.2.9 Recommendations and Troubleshooting

When inference is performed on a model, there are various mathematical schemes for discovering which pieces of evidence would be the most important to discover. The algorithms or procedures produce a list of variables that have not been given evidence. Simply put, they are ordered by the degree to which the application of evidence would simplify the situation.

2. MSBNx Features

This section outlines the most interesting features of MSBNx. The details of the features can be found in later sections of this report and on the MSBNx web pages (http://research.microsoft.com/adapt/MSBNx/). MSBNx offers these features:

- Graphical Editing of Bayesian networks
  - Use MSBNx to graphically relate causes to effects.
  - Express a system's uncertainty by giving your probabilities of various situations.
- Exact Probability Calculations

  MSBNx uses clique-tree propagation methods to calculate the probabilities exactly.
- Decision-Theoretic Diagnosis, Troubleshooting, and Recommendations

    - Using your cost information or defaults, MSBNx dynamically recommends troubleshooting steps.
    - It bases its recommendations on a cost-benefit analysis.
    - If you provide no cost information, a Value of Information (VOI) measure determines the recommendation order.
- XML Format

  MSBNx loads and stores files in an XML-based format.
- COM API
    - The MSBN3 ActiveX DLL provides an COM-based API for editing and evaluating Bayesian networks.
    - MSBN3's event-based design makes it especially easy to use from COM-friendly languages such as Visual Basic and JScript.
- Comprehensive Help Pages
    - Help pages for both the MSBNx editor and the MSBN3 API are on-line
    - The manuals are also installed as Windows Help documents.
    - While editing with MSBNx or while programming with MSBN3, pressing F1 will bring up help for the topic on which you are currently working.
    - The MSBN3 API Help includes overviews of loading models, running inference, and other common programming tasks.
- Other ActiveX controls

  MSBNx includes several graphic components that can be used directly by other programs written in languages that support ActiveX, such as Visual Basic, Jscript, and C++.
- Add-Ins

  You can extend the editing and evaluation abilities of MSBNx by creating add-ins. These add-ins are ActiveX DLLs. They can be created in languages such as Visual Basic.
- Standard Assessment of Probabilities
    - Use the Standard Assessment Tool to assess the probabilities of various situations.
    - Reduce the number of probabilities you need to specified by assuming causal independence.
    - Control the formatting of the probability tables.
    - Optionally display probabilities as odds.
    - Use the Standard Assessment Tool as an ActiveX component in other programs.
- Asymmetric Assessment

  When different situations can be treated the same, you can use the Asymmetric Assessment Tool to reduce the number of probabilities you need to specified.
- Dynamic Properties

Either interactively or programmatically, you can define and attach property values to models and nodes. For example, you can define a property type called “DatabaseKey” with string values. Then on each node in a model, you can attach a string value to used for looking up more information related the node in a database. Model and nodes can have any number of properties. Property values can be strings, reals, arrays of string or reals, or enumerated types.

- Work with Machine-Learned Models
  The WinMine Toolkit (http://research.microsoft.com/~dmax/WinMine/Tooldoc.htm) uses machine learning and statistical methods to create Bayesian networks. MSBNx can load, edit, and evaluate the models created by WinMine.

- Hidden Markov Models
  - MSBNx includes an (undocumented) add-in for editing and evaluating Hidden Markov Models (HMMs).
  - Inference on HMMs is exact, but inefficient for large models**.**

3. The MSBNx Application

MSBNx is a Microsoft Windows software application. In MSBNx each model is represented as a graph or diagram. The random variables are shown as ellipses, called nodes, and the conditional dependencies are shown as arrows, or directed arcs, between variables. At the present time, MSBNx only supports discrete distributions for its model variables. MSBNx supports simultaneous viewing and evaluation of multiple models.

The main tasks within MSBNx are

- Model Creation -- creating and modifying models and their diagrams.
- Working with Model Diagrams
- Model Evaluation -- control the probabilistic inference process.
- Probability Assessment -- enter and maintain prior probabilities.

Let’s look at each of these in turn.

3.1 Model Creation

Models are created in MSBNx as new model documents. Once the empty document window appears, you can add variables and dependency arcs for your model. By default, variables (or nodes) are created without prior probability distributions. You can easily and automatically supply uniform distributions of different kinds at any time.

After the applications loads, you can click the "new document" button on the main application toolbar. An empty diagram document is created and given the default name.

To create the elliptical nodes that represent random variables, right click anywhere in the diagram to activate the diagram context menu. Choose "New Node..." from the menu. An edit box will appear on

the diagram with a default name for the new variable and you can enter a new name. When you strike the Enter key, the new node appears.

To create a conditioning arc between two nodes, do the following steps. Click the left mouse button anywhere on the canvas. This de-selects any selected nodes or other objects. Position the mouse button in the ellipse representing the parent variable. While holding down the control (CTRL) key, also hold down the left mouse button and drag the mouse into the node representing the child variable.

By default, MSBNx will create a discrete distribution for every node created. You can change this behavior using the diagram window context menu. You have the choice of automatically creating standard discrete distributions, causally independent distributions or no distribution.

When dependency arcs are added or removed, the dimensionality of the distribution of the child node is changed accordingly.

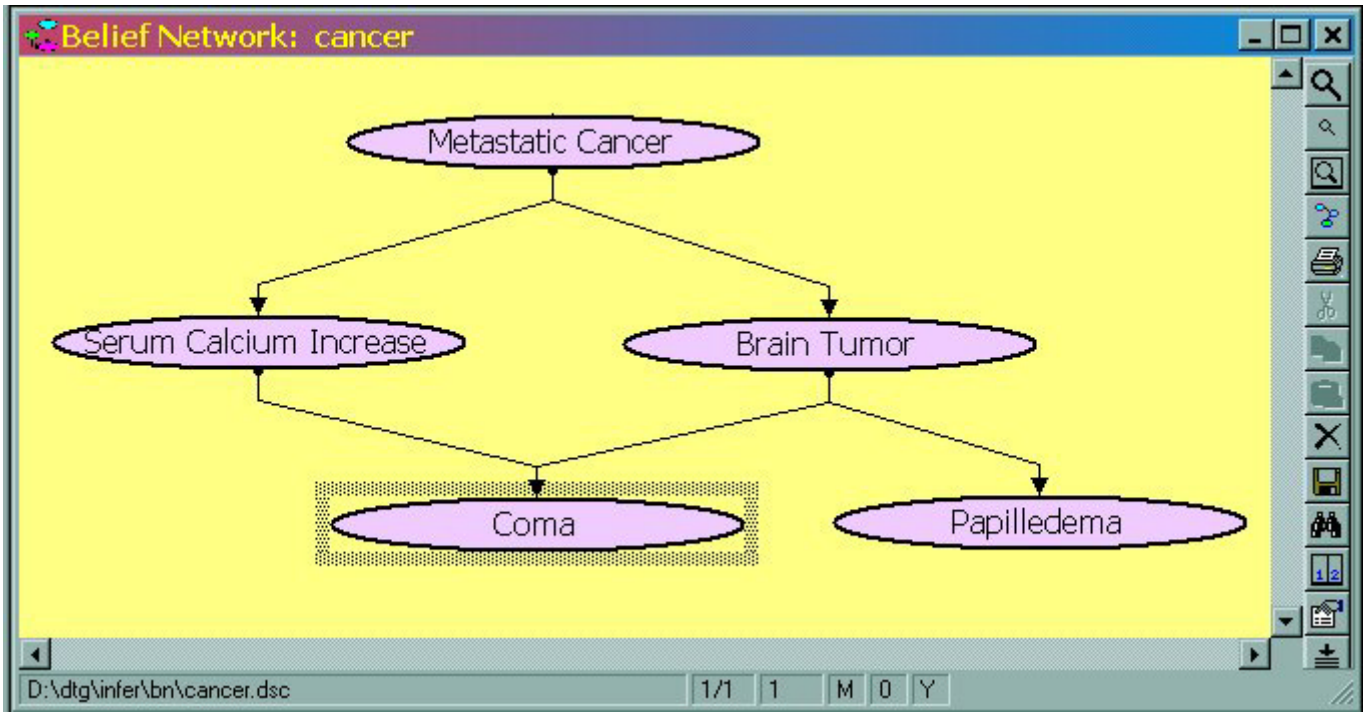
3.2 Working with Model Diagrams

The model diagram window is the primary document of MSBNx. Each window represents a Bayesian network. Each window can only view a single Bayesian network, but multiple windows can view the same Bayesian network.

You can control the diagram window by

- Direct manipulation of the objects within it, using standard operations such as dragging and selection.
- Choosing enabled buttons from the right-hand toolbar.
- Choosing menu items from context menus available on the diagram.

When maximized, a typical diagram window might appear as:



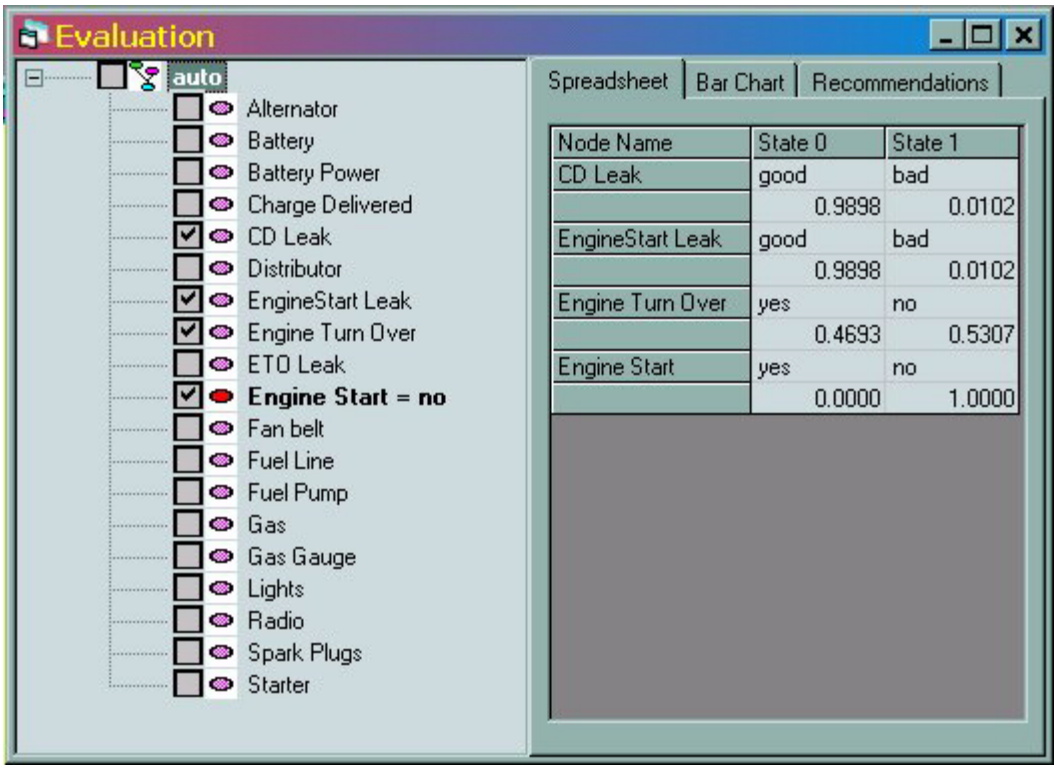
As with most modern graphical applications, you can simply left-click you mouse on an object.

Much of the functionality of MSBNx and its components is presented through context menus. To see a context menu for the diagram, right-click the mouse over any unoccupied space in the diagram. To see a context menu for a node (variable) or arc (edge), right-click on it.

Inference can be performed directly from a diagram window. To enter evidence for a variable, use its node's context menu to set its state. To view the posterior probabilities, enable bar chart displays for any or all of the nodes in the model.

3.3 Model Evaluation

The Model Evaluation window is the primary means of performing inference using a model. (An alternate method is to use bar charts on the main diagram window.)



Performing inference means manipulating evidence in a model and obtaining the posterior probabilities resulting from the changes in evidence.

There are two primary sections to the Evaluation window. The left-hand pane contains a tree-based view of the nodes in the model. The right-hand pane contains a tabbed display of the results of inference presented in several ways.

The basic Evaluation window has the results for a single model. The form can be extended to contain several pairs of evaluation panes (tree view and tabs), each of which can have a different appearance or represent a different model.

The primary purposes of the tree view are to allow you to choose which model variables are shown in the results pane and to enter or remove evidence from the model's nodes. Clicking in the black rectangle next to a node's name causes the posterior probabilities of that node to appear in the spreadsheet and bar chart views to the right of the tree. Right-click on the name of a node and a menu appears containing state names for the node, allowing you to change the evidence.

By default, the Tree View has a single root level, named after the model itself. In the default view, all the nodes of the model are attached directly to this level. You can also use node properties to create another level of organization in the tree view. See Evaluation Display Options for more information.

The right-hand section of an Evaluation window normally has a set of tabs across its top. (This can be controlled using the Evaluation Display Options.)

When the Spreadsheet tab is active, posterior probabilities are shown for the nodes that are checked in the tree view. Each pair of rows represents a node, and each column represents a state of that node.

When the Bar Chart tab is active, a vertical colored bar appears for each node checked in the tree view. The probability mass associated with each state is represented as a distinct color band in the bar.

During evaluation, models configured to support troubleshooting or diagnostic recommendations can provide an ordered list of most effective diagnostic steps. The Recommendations tab shows an ordered list of nodes (by name and description). Each set of lines in the list shows the information score (cost in dollars) of the node and each state of the node. The nodes are shown in order of least cost.

3.4 Probability Assessment

MSBNx supports these kinds of probability distributions:

- Discrete Sparse. This is the standard type of distribution. All possible probabilities are available, but there is no requirement that all values be specified. During evaluation, uniform distributions are automatically supplied for unassessed probabilities.

- Causally Independent. This type of distribution compresses the space of necessary probabilities by assuming that certain states of the parent nodes are mutually exclusive. CI distributions can be changed to Discrete Sparse distributions without loss.

MSBNx supports these kinds of probability assessment:

- Standard assessment is a table-based method. You can locate and edit a particular set of probabilities based upon the states of the parents of the variable.

- Causally Independent or CI assessment is a type of standard discrete assessment. It uses a special type of distribution based upon assumptions about conditional independence among the parents of a variable. Such distributions can be very convenient since they dramatically reduce the number of values that must be entered.

- Asymmetric assessment is a tree-based method. You create sets of probabilities, organized as a decision tree, by making explicit distinctions between states of parents of the variable. The tree serves to reduce the number of values that must be entered, since entire branches can easily be given common values. You cannot perform asymmetric assessment on variables with causally independent distributions.

The mechanics of each of these assessment types have one goal in common: to minimize the number of distinct probabilities required to correctly specify the distribution.

4. Creating Models for Troubleshooting and Diagnosis

A common use of Bayesian network models is to diagnosis system failures in a probabilistic framework.

This has several advantages over conventional rule-based or decision-tree methods, since Bayes nets support uncertain evidence in a theoretically correct fashion. In addition, prior distributions in Bayes nets

can be built that model logical functions such as **AND**, **OR** and **NOT** using what are known as *deterministic* nodes; that is, nodes whose distributions contain only zeroes and ones. Such nodes, therefore, act as logic gates.

A key question in decision theory is this: ***In the current evidence setting, what new evidence would most effectively lead to a clear diagnosis?*** Often known as the ***value of information***, information theory provides mathematical approaches to answering this question.

4.1 Types of Decision-Theoretic Diagnosis

MSBNx supports two algorithms that use information theory to order or rank variables in a Bayes net according to their information weight or influence.

In either scenario, variables or nodes in the model play certain roles. These roles are also known as ***labels***, and must be assigned correctly or the results cannot be interpreted.

Both methods produce as a result an ordered list of variables ranked by a *value of information* score. In a typically implementation, for example, this list would determine the order of questions being asked of a diagnostician or technician.

4.1.1 Diagnosis: Mutual Information

In such a model, variables are assigned one of two roles:

- **Hypothesis Node**. Also known as a *hidden variable*, this is typically a variable that cannot be directly observed. It is the target or purpose of the overall diagnosis.
- **Information Node**. An *observable variable* that influences the hypothesis node(s) in the model.

There may be other nodes in the model which are not labeled; although they influence inference in the normal way, they do not otherwise enter into the diagnostic process.

Utility-based **diagnosis** uses *mutual information* to compute the amount of weight or "lift" that evidence about the state of each information node would bring to each hypothesis variable. The resulting ranking of uncertain (undetermined) information nodes is used to expedite the diagnostic process.

4.1.2 Troubleshooting: Fix-or-Repair Planning

In addition to being assigned to roles, variables in a ***troubleshooting*** model are also given one or more ***costs***. The Bayesian network author may consider that these costs are measured in dollars (or other monetary currency), time (in minutes or seconds) or any other unit that is consistent with the problem formulation.

Troubleshooting uses an algorithm that iterates over all reasonable repair plans in an attempt to find the ones with the highest likelihood of success at the cheapest cost. The result is a list of nodes, ordered by cost. Establishing evidence about the top-ranked (cheapest) node is guaranteed (within the limits of the model) to lead to correct diagnosis in the shortest and cheapest number of steps.

4.2 Requirements for Diagnosis

To perform mutual information diagnosis in a model:

- At least one node in a diagnostic model must be identified as an *hypothesis* node
- At least two of its nodes must be identified as *information* nodes.

4.3 Requirements for Troubleshooting

4.3.1 Cost Factors

There are three types of costs that are important in a troubleshooting model.

- **Cost to Observe.** This is the cost of observing a symptom or sensor. For example, testing the battery of a car or running a blood test for gram-positive bacteria.
- **Cost to Fix.** This is the cost of fixing or replacing a component in a system. For example, replacing the power supply in a computer.

Service cost. This is a cost assigned to the network or model as a whole. In other words, it indicates the cost that would be expected if the diagnostic operation failed. For example, if a computer server in a network could not be repaired through diagnosis it would have to be replaced.

Each of the different possible roles of variables in troubleshooting networks may have either a ***cost to observe***, a ***cost to fix***, both or neither. The ***service cost*** of the model is treated as the *cost to fix* for the entire model as a whole. *Note: The service cost is required to perform troubleshooting.*

The roles of variables in a troubleshooting model and their costs are as follows

Name	Costs Allowed	Purpose
informational	observe	Used to define observable evidentiary variables
problem-defining	fix	Used to define primary symptoms of failure; that is, the element of the model that is the target of the diagnosis.
fixable and observable	observe and fix	Used to define observable and replaceable elements
fixable but unobservable	fix	Used to define elements that can only be replaced or repaired
unfixable	neither	Used to define elements that can neither be fixed or observed

other	neither	Used to define variables that play no direct part in the diagnostic process. These may be deterministic or "modal" variables that reshape the problem in a logical fashion.

4.3.2 Establishing "Problem" Nodes

The most vital part of a troubleshooting network is its problem nodes. These nodes must be declared in a particular manner: state zero (the first state declared) must be associated with the normal behavior of the component or element. All other states must be associated with the mutually exclusive and exhaustive set of states associated with failure modes of the component. Many problem nodes have only two states: *"Works*" and *"Doesn't Work*". If a problem node has more than two states, they must correlate to clearly distinct situations. Consider a computer printer with four states: *"Works"*, *"No Paper is Output" "Printing is Very Slow"*, and *"Printing is Garbled"*. In each case, observation allows its problem state to be distinguished. (There is, however, some ambiguity-- consider a case where printing is both garbled and slow.)

Multiple problem nodes may be defined, but only one is actually considered during any given troubleshooting session.

4.3.3 Using Troubleshooting

The mechanics of troubleshooting work as follows.

1. One of the problem nodes is set (instantiated) to one of its problem states (that is, a state other than state zero).
2. The *Troubleshooting Recommendations* algorithm is run, and a ranked list of nodes is returned, each with its predicted utility.
3. The highest (first) variable in the ranked list is the one with the lowest cost. The technician or diagnostician would then attempt to gather evidence about this variable.
4. The evidence found about the highest ranked variable is entered as evidence into the model. Alternatively, evidence can be entered for any other uninstantiated node in the collection.
5. Return to step 2.

4.4 Evaluation of Diagnostic and Troubleshooting Models

The evaluation window of MSBNx will attempt to determine the correct type of procedure to perform. The rules it uses are as follows.

- If there is a node labeled as problem-defining, then troubleshooting diagnosis is enabled.
- If there is a node labeled as hypothesis, then mutual information diagnosis is enabled.
- Otherwise, diagnosis is disabled.

If other criteria for diagnosis are not met, an error message will indicate the situation.

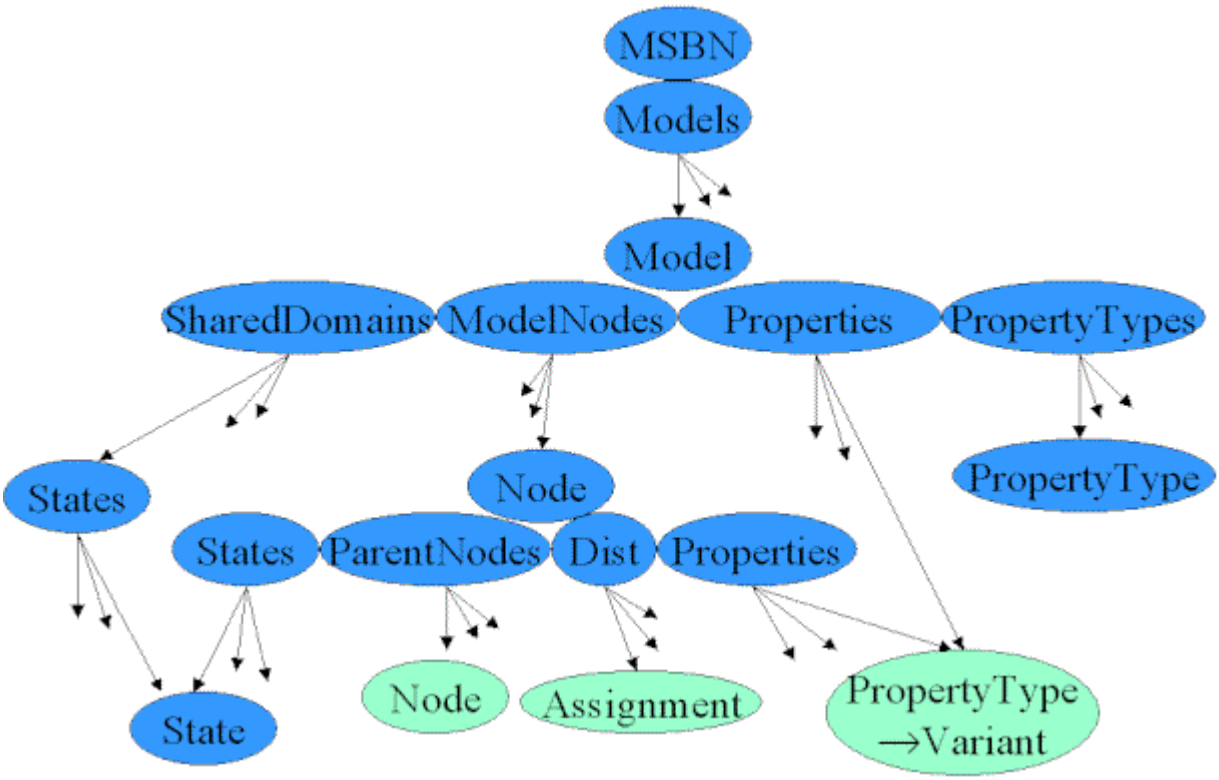
For additional information see the references.

5. The MSBN3 Component

MSBN3 is an ActiveX DLL that offers editing and inference services to other programs such as MSBNx. Because MSBN3 fires events whenever a model is edited, programs that use MSBN3 can be event-driven. This allows the programs to be more modular; indeed, multiple stand-alone programs, add-ins, and utilities to work on the same model without having to know about each other.

5.1 MSBN3 Object Model

The figure shows the MSBN3 object model (except those objects related directly to inference).



For example, the figure shows that:

- the MSBN object contains Models collection,
- the Models collection is made up of Model objects,
- a Model object contains a ModelNodes collection,
- a ModelNodes collection is made up of Node objects,
- a Node object contains a Properties collection,

- a Properties collection is a mapping from PropertyType objects to variant values.

5.2 Loading and Saving Models

This first task for any application that uses MSBN3 is to load or create a model. Other common operations are copying, renaming, removing, and saving.

This section illustrates the operations by annotating a simple Visual Basic program.

5.2.1 Creating a MSBN object and Loading a Model

Here we create a new MSBN object. Then we use the Models collection's Add method to load file "auto.dsc" and call it "Auto". Any error message from the loading go into file "errorfile.log".

```
' Creating a new MSBN object
Dim aMSBN As New MSBN3Lib.MSBN

'Loading a model
Dim model1 As MSBN3Lib.Model
Set model1 = aMSBN.Models.Add("Auto", FileName:="auto.dsc",
ErrorFilename:="errorfile.log")
```

5.2.2 Creating an Empty Model

An empty model also created with Add, but without the *FileName* argument.

```
'Creating an empty model
Dim model2 As MSBN3Lib.Model
Set model2 = aMSBN.Models.Add("Model 2")
```

5.2.3 Renaming and Loading a Model

To rename a model, change the value of its Name property.

```
'Rename it
model2.Name = "cat"
```

The Load method on a model will replace the current contents of the model with the contents of a file. If a *FileName* is given, that file will be used. If no *FileName* is given, the last filename associated with the file (given by the Model's FileName property) is used. The Load method can include the name of an error file for messages generated during the load.

```
'Load
model2.Load FileName:="cat.dsc", ErrorFilename:="errorfile.log"
```

5.2.4 Saving a Model

The Save method on a model will save the contents of the model to a file. A *FileName* and *FileFormat* can specified. If either is not specified then their last values (found in the properties FileName and FileFormat) are used.

```
'Save the auto it to a different place and format
model1.Save FileName:="c:\temp\Auto0.xbn", FileFormat:=fileformat_Xml
```

5.2.5 Loading the Same Model Twice

The Models collection can not have two models with the same name, so here we don't specify a name. This causes a new unique name to be generated and allows us to load the auto model twice.

```
'Load in another copy of the auto model
aMSBN.Models.Add FileName:="auto.dsc", ErrorFilename:="errorfile.log"
```

5.2.6 Copying a Model

The Copy method applied to a model, returns a copy of that model. The copy is also added to the Models collection. If no name is given, the copy will have a generated name.

```
'Copy the cat model
model2.Copy
```

5.2.7 Listing All Models

Like any collection, the Models collection can be enumerated (see Collections and Maps: an Overview in the MSBN3 manual). Its Description property can create a string listing the name of all its models. For example:

```
'Print the name of the all models
Debug.Print aMSBN.Models.Description
```

produces "Auto,cat,Model(1),cat(2)".

5.2.8 Removing Models

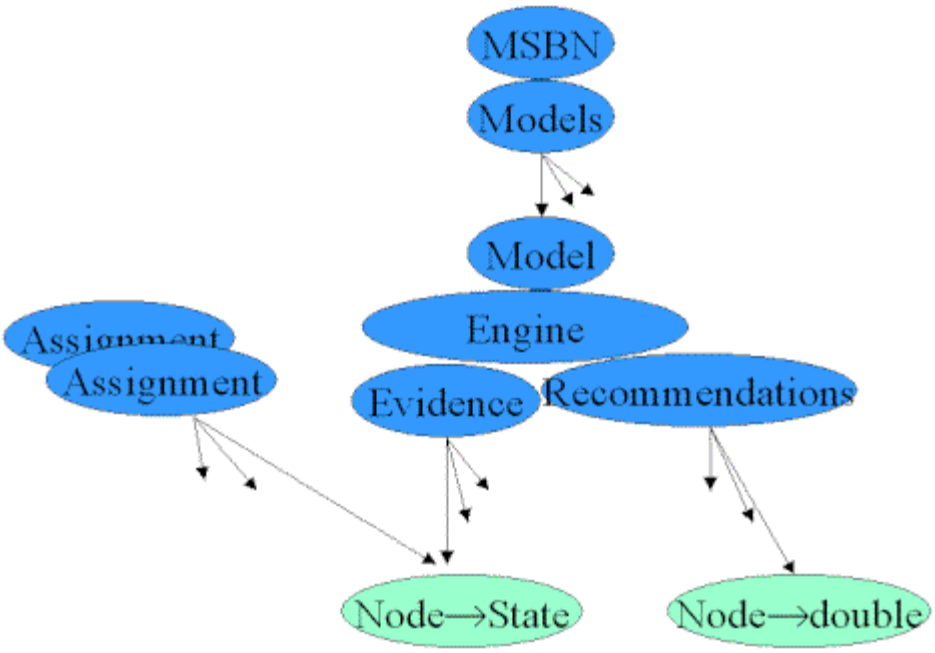
As with any collection, elements of the Models collection can be removed with the Remove method. See Collections and Maps: an Overview in the MSBN3 manual for more information.

```
'Remove the cat model
aMSBN.Models.Remove "cat"
```

5.3 Inference

MSBN3's primary task is inference, in other words, when told some things about the world it estimates the probability of other things in the world. It can also recommend what additional information would be most valuable.

This object model shows the objects related to inference.



For example it shows that:

- a Model object contains an Engine object,
- an Engine object contains an Evidence collection,
- an Evidence collection is a mapping from Node objects to State

This section illustrates inference by annotating a simple Visual Basic program.

5.3.1 Loading a Model and Accessing Its Inference Engine

Here we load a model for troubleshooting car problems. A model's inference engine is available from its Engine property.

```
Dim aMSBN As New MSBN3Lib.MSBN

'Load the Auto model
Dim modelAuto As MSBN3Lib.Model

Set modelAuto = aMSBN.Models.Add("Auto", FileName:="auto.dsc",
ErrorFilename:="loaderror.log")

'Call the model's inference engine "inferAuto"
Dim inferAuto As MSBN3Lib.Engine
Set inferAuto = modelAuto.Engine
```

5.3.2 Finding a Probability

This model has a node named "EngineStart" that has a state named "yes". To find the probability that the car's engine will start, we apply the inference engine's Belief function to the node and state of interest.

```
' Check a probability
Debug.Print "The probability that the car will start is "
```

```
Debug.Print inferAuto.Belief("EngineStart", "yes")
Debug.Print
```

Output:

```
The probability that the car will start is
0.899733245651341
```

To assert a fact, we add a node and state to the Evidence collection using the Add method:

```
' Add some evidence and check some probabilities
Debug.Print "Tell it that the car, in fact, doesn't start."
inferAuto.Evidence.Add "EngineStart", "no"
```

Now, if we ask for the probability that the car will start it will say 0. We can also ask about other probabilities.

```
Debug.Print "The probability that the car will start is "
Debug.Print inferAuto.Belief("EngineStart", "yes")
Debug.Print "The probability that the battery is good is "
Debug.Print inferAuto.Belief("Battery", "good")
Debug.Print
```

Output:

```
Tell it that the car, in fact, doesn't start.
The probability that the car will start is
 0
The probability that the battery is good is
 0.990498642853832
```

We'll tell it that the lights don't work and ask about the probability that the battery is good.

```
' Add some more evidence and check some probabilities
Debug.Print "Tell it the lights don't work"
inferAuto.Evidence.Add "LightsShine", "don't work"
Debug.Print "The probability that the battery is good is "
Debug.Print inferAuto.Belief("Battery", "good")
Debug.Print
```

Output:

```
Tell it the lights don't work
The probability that the battery is good is
 0.97598617912302
```

Using the Set method on the Evidence collection, we can change the evidence and say that lights do work.

```
' Change some evidence and check some probabilities
Debug.Print "Tell it the lights do work"
inferAuto.Evidence.Set "LightsShine", "work"
Debug.Print "The probability that the battery is good is "
Debug.Print inferAuto.Belief("Battery", "good")
Debug.Print
```

Output:

```
Tell it the lights do work
The probability that the battery is good is
 1
```

By enumerating every node and every state, we can see every probability:

```
'For every node and every state, show the probability
Dim aNode As MSBN3Lib.Node
Dim aState As MSBN3Lib.state
Debug.Print "Nodes and probabilities"
For Each aNode In modelAuto.ModelNodes
    Debug.Print aNode.Name,
    For Each aState In aNode.States
        Debug.Print "P("; aState.Name; ")="; inferAuto.Belief(aNode, aState),
    Next aState
    Debug.Print
Next aNode
Debug.Print
```

Output:

```
Nodes and probabilities
Alternator P(good)= 0.999989576605324 P(bad)= 1.0423394676112E-05
Battery P(good)= 1 P(bad)= 0
BatteryPower P(good)= 1 P(low)= 0 P(none)= 0
CD P(yes)= 0.999957618165044 P(no)= 4.23818349563663E-05
CDLeak P(good)= 0.999998957660463 P(bad)= 1.0423395366561E-06
Distributor P(good)= 0.841989863011888 P(bad)= 0.158010136988112
ESLeak P(good)= 0.984198985254525 P(bad)= 1.58010147454754E-02
ETO P(yes)= 0.83456582766937 P(no)= 0.16543417233063
ETOLeak P(good)= 0.984946844194984 P(bad)= 1.50531558050155E-02
EngineStart P(yes)= 0 P(no)= 1
FanBelt P(ok)= 0.999968729816554 P(loose)= 3.12701834458835E-05
FuelLine P(good)= 0.841989863011888 P(bad)= 0.158010136988112
FuelPump P(good)= 0.699067412143921 P(bad)= 0.300932587856079
Gas P(not empty)= 0.991603194714833 P(empty)= 8.39680528516724E-03
GasGauge P(not empty)= 0.990611591485833 P(empty)= 9.3884085141671E-03
LightsShine P(work)= 1 P(don't work)= 0
RadioPlays P(works)= 0.998999999965425 P(doesn't work)= 1.00000003457535E-03
SparkPlugs P(good)= 0.774046752305156 P(bad)= 0.225953247694844
Starter P(good)= 0.849468451921102 P(bad)= 0.150531548078898
```

5.3.3 Recommendations

Use the Recommendations collection to find how useful additional information about a node would be. The first item in Recommendations.Keys is the name of the most valuable node.

```
'Show a recommendation
Debug.Print "Knowing the state of "; inferAuto.Recommendations.Keys(0); "would be
most useful."
Debug.Print
```

Output:

```
Knowing the state of ETOwould be most useful.
```

Enumerating the Recommendations collection (and its KeyObjects collection) list all the useful nodes and their utility (in order of utility).

```
'List all recommendations
Debug.Print "Node", "Utility"
For Each aNode In inferAuto.Recommendations.KeyObjects
    Debug.Print aNode.Name, inferAuto.Recommendations(aNode)
Next aNode
```

Output:

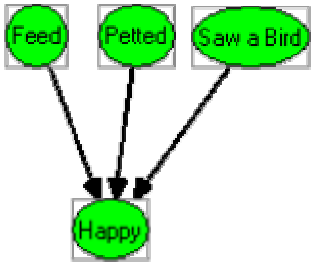
```
Node Utility
ETO -166.471758124647
GasGauge -203.17297015686
SparkPlugs -205.971769894314
Distributor -206.95861573907
Gas -216.904853682089
FanBelt -220.954647059405
Alternator -220.965951259178
FuelPump -234.409330429375
ESLeak -247.667290007677
ETOLeak -248.025747382471
FuelLine -252.400377437429
CDLeak -255.971139079536
Battery -285.971769894314
Starter -335.685042698628
```

5.4 Distributions

Before MSBN3 can do inference, it must know the conditional probability for every node given its parents. This probability distribution is specified has the Dist property of every node. If a node does not have a distribution, it's Dist property will be *Nothing* and no inference will be possible. This overview tells how these conditional distributions are created, read, and edited.

5.4.1 Distributions as Tables

If all nodes are discrete (as in the case currently in MSBN3), a conditional distribution between a node and its parents can be thought of as a table. For example, suppose we have this model of a cat:



in which a cat's happiness is caused by being feed, being petted, and seeing a bird. The conditional distribution of Happy given Feed, Petted, and SawBird can be thought of as a table. Suppose it has these values:

Feed	Petted	SawBird		Happy	
				Yes	No

Yes	Yes	Yes	0.971909	0.0280912
Yes	Yes	No	0.875726	0.124275
Yes	No	Yes	0.877786	0.122214
Yes	No	No	0.59855	0.40145
No	Yes	Yes	0.5	0.5
No	Yes	No	0.0887789	0.911221
No	No	Yes	0.248626	0.751374
No	No	No	0.0187786	0.981221

The table has one row for each possible assignment of parent nodes. It also has one column of parameter values for each state in the Happy node.

5.4.2 Accessing Parameter Values

The parameter values of a distribution can be accessed via the Dist object by specifying a row and column. For example, if ***nodeHappy*** is the node in the model with name "Happy", then this expression:

nodeHappy.Dist(0, "Yes")

will return value 0.971909. Parameters can also be set this way. For example, this expression changes the parameters of the last row:

nodeHappy.Dist(7, "Yes") = 0.01
nodeHappy.Dist(7, "No") = 0.99

As the examples show, rows can be specified with an integer index. Additionally, they can be specified with a parent assignment. This code creates a parent assignment and uses it to change the parameters.

```
Dim anAssign As MSBN3Lib.Assignment
Set anAssign = modelCat.CreateAssignment
anAssign.Add "Feed", "No"
anAssign.Add "Petted", "Yes"
anAssign.Add "SawBird", "Yes"
' anAssign is now: Feed->No,Petted->Yes,SawBird->Yes
nodeHappy.Dist(anAssign, "Yes") = 0.55
nodeHappy.Dist(anAssign, "No") = 0.45
```

The parent assignment must include a setting for every parent, but the parents can be in any order and other nodes can be included, too. (Other nodes will just be ignored.)

5.4.3 *Table Dimensions*

The Dist object's Count property tells the number of rows in the table.

nodeHappy.Dist.Count

The number of columns is available from these equivalent expressions:

nodeHappy.States.Count
nodeHappy.Dist.Node.States.Count

5.4.4 *Parent Assignments for each Row*

To find the parent assignment of a given row, use the Dist object's KeyObjects property. It takes a index integer and returns the Assignment collection for the row. (Like other MSBN3 collections, Dist.KeyObjects also accepts objects, but unlike other collections, it does not accept strings.) This expression retrieves the parent assignment for the row with index 2 (the third row) and then turns that Assignment collection into a string

nodeHappy.Dist.KeyObjects(2).Description

The returned value is the string "Feed->Yes,Petted->No,SawBird->Yes". Similarly, this expression retrieves the same parent assignment and then finds the state to which the Petted node is set:

nodeHappy.Dist.KeyObjects(2)("Petted").Name

It returns "No". The documentation for the Assignment collection has details on access and enumeration of Assignment collections.

5.4.5 *Enumerating the Rows*

The rows of a distribution can be enumerated with Dist object's KeyObjects collection. For example, this code fragment prints parent assignments and parameter values.

```
For Each anAssign In nodeHappy.Dist.KeyObjects
    Debug.Print anAssign.Description,
    For Each aState In nodeHappy.States
        Debug.Print nodeHappy.Dist(anAssign, aState),
    Next aState
    Debug.Print
Next anAssign
```

Output:

```
Feed->Yes,Petted->Yes,SawBird->Yes 0.971908986568451 0.028091199696064
Feed->Yes,Petted->Yes,SawBird->No 0.875725984573364 0.124274998903275
Feed->Yes,Petted->No,SawBird->Yes 0.877785980701447 0.122213996946812
Feed->Yes,Petted->No,SawBird->No 0.598550021648407 0.401450008153915
Feed->No,Petted->Yes,SawBird->Yes 0.55 0.45
Feed->No,Petted->Yes,SawBird->No 8.87788981199265E-02
0.911221027374268
```

```
Feed->No,Petted->No,SawBird->Yes 0.248625993728638 0.751374006271362
Feed->No,Petted->No,SawBird->No 0.01 0.99
```

5.4.6 Default Parameters

Suppose we want to model the happiness of a second cat with this distribution:

Feed	Petted	SawBird		HappyCat2	
				Yes	No
Yes	Yes	Yes		0.99	0.01
Yes	Yes	No		0.01	0.99
Yes	No	Yes		0.01	0.99
Yes	No	No		0.01	0.99
No	Yes	Yes		0.01	0.99
No	Yes	No		0.01	0.99
No	No	Yes		0.01	0.99
No	No	No		0.01	0.99

This cat is only likely be happy when everything is going its way. Do we really have to specify identical parameters for 7 of the 8 rows? The answer is no. We can instead, specify default parameters and let 7 of the 8 rows use that default.

This code fragment shows the creation of a new node with parents and states based on *nodeHappy*. The AddDist method is used to add a distribution to the node. The default parameter values of the distribution are then set with Default. These at first apply to all rows, but then are over ridden for row #0. We use UsingDefault to confirm that row #0 is not using the default parameters but that row #1 is.

```
' Create a node for a new cat's happiness.
Dim nodeHappyCat2 As MSBN3Lib.Node
Set nodeHappyCat2 = modelCat.ModelNodes.Add("HappyCat2", "HappyCat2", _
States:=nodeHappy.States, ParentNodes:=nodeHappy.ParentNodes)
nodeHappyCat2.AddDist

' Specify the default parameters
nodeHappyCat2.Dist.Default("Yes") = 0.01
nodeHappyCat2.Dist.Default("No") = 0.99

' Override the default on row #0
nodeHappyCat2.Dist(0, "Yes") = 0.99
```

```
nodeHappyCat2.Dist(0, "No") = 0.01

'Confirm that row #0 is not using the default and that row #1 is:
Debug.Assert Not nodeHappyCat2.Dist.UsingDefault(0)
Debug.Assert nodeHappyCat2.Dist.UsingDefault(1)
```

The UsingDefault property can also be set *true* or *false* to force a row to use the defaults or not.

5.4.7 Creating Causally Independent Distributions

Suppose we wish to model the happiness of a third cat. For this cat we want to assume that the causes of its happiness are independent. Here is what our table looks like:

Feed	Petted	SawBird	HappyCat3	
			Yes	No
Yes	Yes	Yes	0.99	0.01
Yes	Yes	No	0.50	0.50
Yes	No	Yes	0.40	0.60
No	Yes	Yes	0.05	0.95

Notice that it has fewer rows than before. It has one row for the "all normal" case (this is called the leak case) and one row for each abnormal state of any parent.

The distribution is called "causally independent" or CI. MSBN3 requires that all parents of a CI distribution have their Normal state before their Abnormal states. For example the Feed nodes states must be ordered "Yes", "No", not "No","Yes".

This code fragment shows the creation of a node with a causally independent distribution, setting its parameters, and then printing the result.

```
' Create a node for a new cat's happiness.
Dim nodeHappyCat3 As MSBN3Lib.Node
Set nodeHappyCat3 = modelCat.ModelNodes.Add("HappyCat3", "HappyCat3", _
States:=nodeHappy.States, ParentNodes:=nodeHappy.ParentNodes)
nodeHappyCat3.Add deCondCI

' Specify the parameters for the four rows
nodeHappyCat3.Dist(0, "Yes") = 0.99
nodeHappyCat3.Dist(0, "No") = 0.01
nodeHappyCat3.Dist(1, "Yes") = 0.5
nodeHappyCat3.Dist(1, "No") = 0.5
nodeHappyCat3.Dist(2, "Yes") = 0.4
nodeHappyCat3.Dist(2, "No") = 0.6
nodeHappyCat3.Dist(3, "Yes") = 0.05
```

```
nodeHappyCat3.Dist(3, "No") = 0.95

Debug.Print
Debug.Print "Happy Cat 3"
For Each anAssign In nodeHappyCat3.Dist.KeyObjects
    Debug.Print anAssign.Description,
    For Each aState In nodeHappyCat3.States
        Debug.Print nodeHappyCat3.Dist(anAssign, aState),
    Next aState
    Debug.Print
Next anAssign
```

Output:

```
Happy Cat 3
Feed->Yes,Petted->Yes,SawBird->Yes 0.99 0.01
Feed->No,Petted->Yes,SawBird->Yes 0.5 0.5
Feed->Yes,Petted->No,SawBird->Yes 0.4 0.6
Feed->Yes,Petted->Yes,SawBird->No 0.05 0.95
```

The new node has a CI distribution because of the Add method that created it included this: "DistType:=deCondCI". The type of distribution can also be determined (and changed) using the Dist object's Type property. The default distribution type is deCondSparse, the type we used above.

5.4.8 From Parameters to Probabilities

Suppose we want to know probability that Cat 3 will be happy when all the parent nodes are set to "No". The table contains no row for this condition. We could use the general inference (see Inference: an Overview in the MSBN3 manual), but there is a simpler way. We can use the Dist object's Prob property. For example, this code fragment:

```
Set anAssign = modelCat.CreateAssignment
anAssign!Feed = "No"
anAssign!Petted = "No"
anAssign!SawBird = "No"
Debug.Print nodeHappyCat3.Dist.Prob(anAssign, "Yes")
```

will output 0.0099.

Indeed, even when a parent assignment is in the table, the Prob property should be used. For example:

```
Set anAssign = modelCat.CreateAssignment
anAssign!Feed = "No"
anAssign!Petted = "Yes"
anAssign!SawBird = "Yes"
Debug.Print nodeHappyCat3.Dist.Prob(anAssign, "Yes")
```

has output 0.495 which less than the 0.5 parameter in the table. Why does the probability and the parameter differ? Both account for cat being unhappy because it is not feed. Only the probability, however, accounts for cat possibly being unhappy even when all is well. (This possibility is represented by row #0, the leak case).

5.5 Properties

With MSBN3's property mechanism you can attach strings and numbers to models and nodes.

This section illustrates the property mechanism by annotating a simple Visual Basic program.

5.5.1 Prerequisite Information

The property mechanism is based on MSBN3 collections and maps. See the documentation on Collections and Maps: an Overview for important background information.

5.5.2 Starting the Example

To start this example, we create a model called ***aModel*** with one node called ***aNode***.

```
    ' ========================================================================
    ' Create a model with one node
    ' ========================================================================

    ' Create a model
    Dim aMSBN As New MSBN3Lib.MSBN
    Dim aModel As MSBN3Lib.Model
    Set aModel = aMSBN.Models.Add

    'Create a node
    Dim aNode As MSBN3Lib.Node
    Set aNode = aModel.ModelNodes.Add("EngineStart")
    aNode.States.Add "Working"
    aNode.States.Add "Broken"
```

5.5.3 Defining Property Types

Before a property value can be attached to a model or node, its type must be defined. The definition tells what values the property can have. The five possibilities are:

- Real
- String
- Real Array
- String Array
- Enumerated

Property types are defined by adding PropertyType objects to the model's PropertyTypes collection. This is done with the PropertyTypes collection's Add method. This code shows the definition of 5 property types, one for each possibility.

```
    ' ========================================================================
    ' Define 5 property types, call the first ptCost
    ' ========================================================================

    ' Define a real-valued property type
    Dim ptCost As MSBN3Lib.PropertyType
    Set ptCost = aModel.PropertyTypes.Add("Cost", "What does it cost?",
PROPTYPETYPE:=pttReal)

    ' Define a string-valued property type
```

```
    aModel.PropertyTypes.Add "Category", "What is its category?",
PROPTYPETYPE:=pttString

    ' Define a real-array-valued property type
    aModel.PropertyTypes.Add "ScoreList", "What are the scores?",
PROPTYPETYPE:=pttReal, IsArray:=True

    ' Define a string-array-valued property type
    aModel.PropertyTypes.Add "NameList", "What are the names?",
PROPTYPETYPE:=pttString, IsArray:=True

    ' Define a enumerated property type
    aModel.PropertyTypes.Add "Visibility", "How visible is it?",
PROPTYPETYPE:=pttEnumerated, _
            EnumValues:="Visible, Semivisible, Invisible"
```

As with other maps and collections, we can use PropertyTypes's Description property to get a quick summary of its contents.

```
    Debug.Print aModel.PropertyTypes.Description
```

Output:

```
Cost,Category,ScoreList,NameList,Visibility
```

5.5.4 Attaching Properties to Models and Nodes

With some property types defined, property values can be attached to the model and its nodes. This example shows 3 property values being added to the model.

```
    ' ====================================================================
    ' Add 3 properties on the model and 3 on the node
    ' ====================================================================

    'Adding three properties to the model
    aModel.Properties.Set ptCost, 80#
    aModel.Properties.Set "Category", "Science and Nature"
    aModel.Properties.Set "ScoreList", Array(2, 3, 5)
```

Property values are added using the model or node object's Properties collection. As the example shows, the Set method can used to add an association between PropertyType objects and values. The property type can be specified either via an object (e.g. *ptCost*) or a string (e.g. "Category") or an integer index into the PropertyTypes collection. The value specified is a OLE variant. In the case of array values, an OLE smart array is used.

A summary of the model's current property settings can be found with Description.

```
    Debug.Print aModel.Properties.Description
```

Output:

```
Cost->80,Category->Science and Nature,ScoreList->Array<REAL>(...)
```

The Set method can also be used to add properties to nodes.

```
    'Adding three properties to the node
    aNode.Properties.Set 0, 40
```

```
aNode.Properties.Set "NameList", Array("Bobcat", "Sparky")
aNode.Properties.Set "Visibility", "Semivisible"
Debug.Print aNode.Properties.Description
```

Output:

```
Cost->40,NameList->Array<STRING>(...),Visibility->1
```

The value of enumerated property types, like the one named "Visibility", can either be an integer or a string. If a string, it must be in the **EnumValues** list given when the property type was defined.

5.5.5 Accessing a property value

A property value can be accessed by indexing the Properties collection. (For details see: Properties). The value returned by ***aModel.Properties("ScoreList")*** is an array. In the example below, an element of this array is accessed with "***(1)***".

```
' ========================================================================
' Access a property value
' ========================================================================
Debug.Print aModel.Properties(ptCost)
Debug.Print aModel.Properties("Category")
Debug.Print aModel.Properties("ScoreList")(1)
```

Output:

```
 80
Science and Nature
 3
```

5.5.6 Changing a property value

A property value can be also be changed using the Properties collection's Set method.

```
' ========================================================================
' Change a property value
' ========================================================================
aNode.Properties.Set ptCost, 70#
aNode.Properties.Set "Visibility", 0
Debug.Print aNode.Properties.Description
```

Output:

```
Cost->70,NameList->Array<STRING>(...),Visibility->0
```

5.5.7 Testing for a property

To test if a model or node has a property, use the the Properties collection's ExistingKey method. Here the node does have the "NameList" property but the model nodes not.

```
' ========================================================================
' Assert that node has the "NameList" property but that the model does not
' ========================================================================
Debug.Assert aNode.Properties.ExistingKey("NameList")
Debug.Assert Not aModel.Properties.ExistingKey("NameList")
```

5.5.8 Removing a property

To remove a property from a model or node has a property, just set its value to *Nothing*.

```
' ========================================================================
' Remove the "Name List" property from the node
' ========================================================================
aNode.Properties.Set ptCost, Nothing
Debug.Print aNode.Properties.Description
```

Output:

```
NameList->Array<STRING>(...),Visibility->0
```

5.6 Events

The goal of the MSBN3 events interface is to allow objects (ActiveX controls, VB forms, etc.) to precisely monitor MSBN3 editing with very little additional code. Toward this end, the events correspond to well-defined primitives. For example, there is no "Load Model" event. Rather, loading a model generates a series of primitive events like "ModelNodesAdd(*aNode*)" and "NodeParentNodesAdd(*nodeChild*, *nodeParent*)". Moreover, when you receive the "ModelNodesAdd" event, you are promised that node does have a name and description, but does not have any parents or properties. Likewise, when you receive an event that a node is to be removed, you are promised that it does not have any parents or properties.

The goal of the MSBN3 events interface is to allow objects (ActiveX controls, VB forms, etc.) to precisely monitor MSBN3 editing with very little additional code. Toward this end, the events correspond to well-defined primitives. For example, there is no "Load Model" event. Rather, loading a model generates a series of primitive events like "ModelNodesAdd(*aNode*)" and "NodeParentNodesAdd(*nodeChild*, *nodeParent*)". Moreover, when you receive the "ModelNodesAdd" event, you are promised that node does have a name and description, but does not have any parents or properties. Likewise, when you receive an event that a node is to be removed, you are promised that it does not have any parents or properties.

These promises are detailed in the event documentation.

With few exceptions, you will receive an event after the editing action has occurred. If an action will destroy an object, then the event is fired before the action so that the object can be queried. Examples of events that fire before actions are "ModelNodesRemove(*aNode*)" and "ModelsRemove(*aModel*)".

The events make heavy use of passing objects. In all cases (because of the requirements of Visual Basic) these are of the generic object type "IDispatch *".

6. Summary

We presented key components and services of MSBNx, a component-based tool kit for Bayesian network development and inference. We described how the toolkit can be used for diagnostic and troubleshooting support. We described also the use of MSBNx components from other programs.

Acknowledgments

Feedback, ideas, and methods represented in MSBNx code and functionality were derived from numerous people on the Decision Theory and Adaptive Systems (DTAS) team, and in its descendant teams, the Adaptive Systems & Interaction (ASI), Machine Learning & Applied Statistics (MLAS), and the Data Mining and Exploration (DMX) groups. Contributors include Jack Breese, David Heckerman, Eric Horvitz, David Hovel, Carl Kadie, Chris Meek, and Koos Rommelse. Finally, we have continued to receive feedback from the population of users of MSBN32, our previous tool. We expect to continue to listen to comments about the MSBNx tools.

References

J. S. Breese and D.E. Heckerman. Topics in Decision-Theoretic Troubleshooting: Repair and Experiment. Microsoft Research Technical Report TR-96-06, 1996.

D. E. Heckerman, E. J. Horvitz, and B. N. Nathwani. Toward Normative Expert Systems: Part I The Pathfinder Project. *Methods of Information in Medicine*, 31:90-105, 1992.

D. E. Heckerman and J. S. Breese, et al., Troubleshooting Under Uncertainty. Microsoft Research Technical Report TR-94-07, 1994.

D. E. Heckerman and J. S. Breese. Causal Independence for Probability Assessment and Inference Using Bayesian Networks. Microsoft Research Technical Report TR-94-08, 1994.

E.J. Horvitz, J.S. Breese, and M. Henrion, Decision Theory in Expert Systems and Artificial Intelligence, *Journal of Approximate Reasoning, Special Issue on Uncertainty in Artificial Intelligence*, 2:247-302, 1988.

F.V. Jensen. *An Introduction to Bayesian Networks*, Springer-Verlag, New York NY, 1996. ISBN 0-387-91502-8.

J. Pearl. *Probabilistic Reasoning in Intelligent* Systems., Morgan Kaufmann, San Mateo CA, 1988. ISBN 0-934613-73-7.

D. Poole, *et al. Computational Intelligence*. Oxford University Press, New York NY, 1998. ISBN 0-19-510270-3.