JavaBayes Version 0.346
Bayesian Networks in Java
User Manual

Fabio Gagliardi Cozman
fgcozman@usp.br
http://www.usp.br/~fgcozman/home.html
Escola Politécnica
University of São Paulo

January 31, 2001

Preface

Bayesian networks have been used as a fundamental tool for the representation and manipulation of beliefs in Artificial Intelligence. There have been implementations of Bayesian networks in a variety of formats and languages.

JavaBayes is a system that handles Bayesian networks: it calculates marginal probabilities and expectations, produces explanations, performs robustness analysis, and allows the user to import, create, modify and export networks.

JavaBayes is the first full implementation of Bayesian networks in Java. A Java implementation has several advantages. First, Java is the best bet yet on a truly portable language; a package written in Java can be exported and run in Unix, Macintosh and Windows platforms without too much hair-splitting. Second, Java has been adopted by browsers in the Internet; a program or package written in Java can be intimately coupled with World Wide Web pages and can reach a gigantic audience. Third, and perhaps most important, a Java package can work as a tool for people that are interested in using reasoning in network-based applications. Suppose you had to put together a web page and you wanted to use some simple tool to reason about uncertainty in the domain of interest. A compact implementation of Bayesian networks in Java would be handy for such a task. Finally, Java is a good object-oriented language; Java has a set of widgets that allow researchers to quickly prototype interfaces, and Java has functionality for multi-threaded processing, something that can be very useful for future parallelization of inference algorithms.

I hope this *JavaBayes* project is useful to others. It is far from a fully tested product; there are many possible improvements. I hope others will be interested in helping me test, modify and improve this. *JavaBayes* is distributed under the GNU License; I have had fun coding it and hope others will have fun interacting with it.

Many people have contributed decisively to *JavaBayes*.

First, thanks to my former advisor, Eric Krotkov, for the encouragement, the suggestions, and for giving me time to think about new ideas.

JavaBayes uses the inference algorithm presented by Rina Dechter in the Twelfth Annual Conference on Uncertainty in Artificial Intelligence [6]. I thank Prof. Dechter for pointing me to a Scheme implementation of this algorithm, which was nicely coded by Nicolas Thiéry. It has been brought to my attention that this algorithm, which is very similar to the so-called peeling algorithm, has also been published by Zhang and Poole under the name *variable elimination*. The original algorithm has been enhanced to obtain calculation of all marginal probabilities in a network simultaneously (similar to the commonly used joint tree algorithm [9]).

The graphical user interface is based on the original work by Sreekanth Nagarajan and Bruce D'Ambrosio at Oregon State University. Even though the current interface does not contain their code, the appearance of the network editor is based on their system (their system used the front-end interface to make calls to a server-based inference engine). Thanks much to Sreekanth Nagarajan and Bruce D'Ambrosio for making their interface available.

I also appreciate the encouraging comments and suggestions about this project sent by a number of people in the Internet. Thanks in particular to Chao-Lin Liu and Michael Wellman by proposing the name *JavaBayes* and for crucial help with the first version of the user interface; Chao-Lin Liu gave several other suggestions and wrote the code that detects cycles in a network. Hadar Ziv suggested a *zipped* version for PC-based users. Nir Friedman gave important suggestions concerning the Interchange Format. Akihiro Shinmori detected and corrected problems with the XML-based format. Wei Zhou detected and provided fixes for several problems with the generalized variable elimination algorithm. Alex Bronstein and his group at HP labs offered several suggestions and words of support.

Robert E. Bruce corrected a crucial problem with the code and Alexander Churbanov found some bugs in the system. I'm also grateful for various suggestions and bug fixes/warnings by Alan Mehlenbacher, Bozhena Bidyuk, Robert Wilensky, Simon Keizer, Michael Becke, Jason Townsend.

Good luck with the system; I hope it works well and provides useful assistance and guidance.

Cheers,

Fabio Cozman

Chapter 1

Introduction

The *JavaBayes* system is a set of tools for the creation and manipulation of Bayesian networks. The system is composed of a graphical editor, a core inference engine and a set of parsers. The graphical editor allows you to create and modify Bayesian networks in a friendly interface. The parsers allow you to import Bayesian networks in a variety of formats. The engine is responsible for manipulating the data structures that represent Bayesian networks. The engine can produce:

- the marginal probability for any variable in a Bayesian network.
- the expectations for univariate functions (for example, the expected value of a variable).
- configurations with maximum a posteriori probability.

Typically, the user assigns values to some variables in a network and asks the posterior marginal probability or expectation of some other variables. The set of variables that have assigned values is called the *evidence*. Marginal probabilities and expectations can be calculated conditional on any number of observations inserted into the network.

Another typical situation is that the user specifies some evidence and asks which are the values of non-evidence variables that lead to the maximum possible posterior probability for the evidence. A configuration with such optimal characteristics has been called an *explanation* for the available evidence. When an optimal configuration is produced, the variables in the network are *estimated* in the sense that their "best" values are found, where "best" is measured in terms of posterior probability. It is possible to specify a group of variables in the network to be estimated, or to estimate all variables in the network at once.

JavaBayes can produce marginal distributions and expectations using two different algorithms: variable elimination and bucket tree elimination. In the first case, inferences are generated from scratch for each query; in the second case, a data-structure (bucket tree) is generated once and several queries can be generated directly from the bucket tree. Variable elimination consumes less memory, but it may take longer if several queries are made to the same network with the same collection of observations.

A capability of *JavaBayes*, which sets it apart from other inference engines, is the ability to conduct robustness analysis on top of inferences. Bayesian robustness analysis is an on-going research topic, where sets of distributions are associated to variables: the size of these sets indicates the "uncertainty" in the modeling process. *JavaBayes* can use models with sets of distributions to calculate intervals of posterior distributions or intervals of expectations. The larger these intervals, the less robust are the inferences with respect to the model.

JavaBayes is distributed under the GNU License; if you want to distribute *JavaBayes* to someone, you have to package the whole distribution *including* the GNU license. If you need to include the Bayesian network engine in some application, you must also make a request; the engine might be available to you under the Lesser General Public License.

The *JavaBayes* distribution is available in the Internet; the various Java packages that compose the system are provided in source and bytecode forms. The Java packages can be used in other applications or applets (provided that the GNU license is respected) as a tool for probabilistic reasoning. The complete system, with graphical interface, can be used to construct and experiment with Bayesian networks, as a teaching and/or development tool.

Note that *JavaBayes* is distributed as bytecodes that are executable in the standard Java Virtual Machine as specified by Sun Microsystems Inc. Modification of the source code and generation of bytecode from modified source code is allowed under the restrictions specified by the GPL (the GNU license), but compilation of the source files to generate other types of executables or non-standard bytecode is not covered by the license and is *not* allowed. This is emphasized so that *JavaBayes* is always distributed as a portable, architecture-neutral system; if you are interested in generating non-portable executables from *JavaBayes*, then you must contact me. Generation/commercialization of such executables requires a specific license which must be negotiated. Again, this is emphasized to avoid the confusion that would occur if several unauthorized types of compiled code were to be generated from the source.

This manual is an on-going effort to document the *JavaBayes* system. There are directions for dowloading the system from the Internet, a brief description of how to run the system, and a brief description of how to compile the system. From there, some examples are presented, followed by a step-by-step description of the system. This manual also discusses the several data formats that are understood and generated by the system, and finishes with several miscellaneous items. You can also find a description of the inference algorithm used by *JavaBayes* in the system's web site.

There are some other projects that use Java with Bayesian networks:

- The Bayes applet produced by Dawid Poole and his group.
- The user interface of the new Hugin system is written in Java.
- The Bayesian Net Simulator implemented by Yoichi Motomura.
- The discussion of Bayesian networks using Java applets by Joel Martin.

Chapter 2

Downloading *JavaBayes*

To get *JavaBayes*, you have two options:

1. You can download the gzip/tar file JavaBayes-0.346.tar.gz. You have to use the `gunzip` and `tar` utilities to obtain all the files.

2. You can download the zip file JavaBayes-0.346.zip. You have to use one of the many utilities that read the zip format.

You can also download a compressed Postscript version of this manual. Finally, you can download a compressed javadoc-generated documentation for source code, generated and generously donated by Alan Mehlenbacher (note that I'm using the jar program for the javadoc files, as I would like to move to distribution through the jar program in the near future).

Note that *JavaBayes* is still coded using the Java 1.0.2 specification in the graphical user interface. But the current executable code has been produced with new development environments that are entirely based on Java 1.1. At this point, no check is made to guarantee that *JavaBayes* works with Java 1.0.2 browsers; the only valid test for this would be to actually run the system through Java 1.0.2 browsers. Also note that *JavaBayes* will be entirely converted to Java 1.1 at some point, so you should consider using tools that support Java 1.1.

Downloading and unpacking the *JavaBayes* distribution should produce several directories and files:

- A `README` directory, with miscellaneous information, such as: license, list of changes, list of bugs.
- A `Source` directory, containing the files with Java source code. This directory contains the JavaBayes.java file, which defines the main method for *JavaBayes*, and all the Java packages that are used by the system.
- A `Classes` directory, which simply contains the class files that result from the compilation of source files in the `Source` directory.
- An `Examples` directory, containing publicly available Bayesian networks in the formats understood by *JavaBayes*.

IMPORTANT: If you download *JavaBayes*, I ask you to notify me with a small email message. This software is experimental and will be evolving soon as I test it and kill bugs. I would like to know who has it so that I can send messages indicating patches and new versions. Even if you do not want to receive messages, send me a message indicating that you have the software but you do not want any messages. Thanks.

Chapter 3

Running *JavaBayes*

You can run *JavaBayes* in two different ways.

First, you can run it as an applet, inside a world-wide-web document using the APPLET tag in HTML. Note that in this mode, you are restricted by the Java sandbox model, so you cannot perform a number of operations. For example, you cannot load/save files from the local file system. Suppose you install the *JavaBayes* class files in the directory `Classes`; you would use the following piece of code in your HTML page to call *JavaBayes*:

```
<APPLET
CODEBASE="Classes/"
CODE="JavaBayes.class">
</APPLET>
```

Second, you can run *JavaBayes* as an application. After you download *JavaBayes*, the easiest thing is to go to the `Classes` directory and type

```
java JavaBayes
```

This should invoke the java runtime interpreter and tell it to load the class JavaBayes. You should see the *JavaBayes* windows pop out in a moment. If you have problems at this point, try checking your Java virtual machine and your classpath.

If you have your CLASSPATH variable set properly, you should be able to run *JavaBayes* anywhere just typing

```
java JavaBayes
```

Basically, the CLASSPATH tells the interpreter where to look for classes, and you can set it either as a system variable or as a parameter to the `java` interpreter. Usually the Java compiler will have a system classpath, and the compiler appends the contents of an environment variable CLASSPATH to the system classpath. You have to set up CLASSPATH so that the compiler can read files in the local directory and in the directory that contains the *JavaBayes* classes. Here is an example for the variable CLASSPATH:

```
CLASSPATH=.:/usr/users/your-name-here/Java/JavaBayes/Classes/
```

Chapter 4

Compiling *JavaBayes*

In case you change the *JavaBayes* source code for some reason, note that all compiled classes should be placed in the `Classes` directory. You must set a flag in the Java compiler that directs output to that directory (use the flag -d); otherwise you will end up with class files mixed with source files. For example, you can go to the `Source` directory and type:

```
javac -d ../Classes JavaBayes.java
```

One thing you *do not* need to do is to install JavaCC, the parser generator that creates the parser in JavaBayes. Even though the distribution contains source files for JavaCC, there is no need to install JavaCC unless you want to modify the grammars for the parsers. In case you want to check JavaCC, you can obtain information about it with Metamata, the company that distributes it.

There are some issues that you must keep in mind:

- Unfortunately, most current implementations of Java have some bugs. If you see some absurd crash, like a complete core dump of the whole Java Virtual Machine, please try another Java Virtual Machine. Remember that a Java system should never crash no matter which program it is running.
- Things are likely to break if you don't have your CLASSPATH variable set properly!

Chapter 5

Using *JavaBayes*

The easiest way to familiarize yourself with *JavaBayes* and Bayesian networks is to try some simple examples. In this section, two Bayesian networks are analyzed, the DogProblem network and the Alarm network. Both are available in the *JavaBayes* distribution, in the `Examples` directory (the DogProblem network is at `Examples/DogProgram`, and the Alarm network is at `Examples/Alarm`).

You should probably follow the discussion below while running the system; you can do that in one of two ways. You can download the system and run it as an application, or you can follow the examples inside your browser by running the *JavaBayes* applet. In the applet, you can use the full functionality of the system *except* that you have to comply with your browser's policy concerning applets. Running as an applet, *JavaBayes* does not let you perform load/save operations in the local file system.

5.1 The DogProblem network

Consider the following popular network in Figure 5.1, introduced by Charniak in his description of Bayesian networks [4].

The network describes a simple situation. Suppose you are going home, and you want to know what is the probability that the lights are on given the dog is barking and the dog does not have any bowel problem. If the family is out, often the lights are on. The dog is usually out in the yard when the family is out and when it has bowel troubles. And if the dog is in the yard, it probably barks. The Bayesian network for the example is given in Figure 5.1;

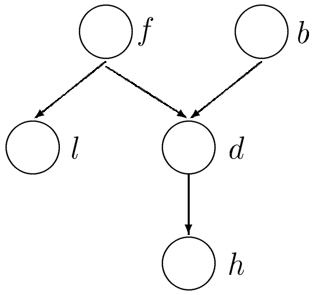

Figure 5.1: The DogProblem network

f	family-out
b	bowel-problem
l	lights-on
d	dog-out
h	hear-bark

Table 5.1: Abbreviations for the DogProblem variables.

to refer to variables, use the abbreviations in Table 5.1.

The several relationships in the example are captured by probability distributions on the nodes. So we have Table 5.2, that specifies the probability for the conditional events and their complements (the complement of a variable is indicated by the superscript c).

The structure defined by the graph in Figure 5.1 and Table 5.2 is a Bayesian network; it specifies a complete joint probability distribution over all variables in the problem. In short, a Bayesian network is a mechanism for the specification of joint probability distributions by graphical means.

More mathematically, a Bayesian network is composed of a directed acyclic graph and a collection of conditional probability distributions. Every node of the graph is associated with a variable X_i. The arcs in the graph indicate the collection of parents of any variable. For a variable X_i, denote the parents of X_i by $\mathrm{pa}(X_i)$. Every node is also associated with a conditional probability distribution: the probability of the variable X_i conditional on its parents $\mathrm{pa}(X_i)$. This probability is denoted by $p(X_i|\mathrm{pa}(X_i))$; for discrete variables this probability is represented by a table.

	Probability of true	Probability of false
$p(f)$	0.15	0.85
$p(b)$	0.01	0.99
$p(l\|f)$	0.60	0.40
$p(l\|f^c)$	0.05	0.95
$p(d\|f,b)$	0.99	0.01
$p(d\|f,b^c)$	0.90	0.10
$p(d\|f^c,b)$	0.97	0.03
$p(d\|f^c,b^c)$	0.30	0.70
$p(h\|d)$	0.70	0.30
$p(h\|d^c)$	0.01	0.99

Table 5.2: Probability values for the DogProblem network.

The graphical and probabilistic structure of a Bayesian network represents a single joint probability distribution. This distribution is obtained as follows:

$$p(X_1 \ldots X_n) = \prod_{i=1}^{n} p(X_i|\mathrm{pa}(X_i)).$$

Note that other interpretations of Bayesian networks exist in the literature [11]); looking at them as representations for joint distributions is just the easiest way to grasp their meaning.

5.1.1 Loading and saving the DogProblem network

There are three distinct ways to manipulate the DogProblem network in *JavaBayes*:

1. Load the network from the local disk. To load a network, go to the `File` menu in the *JavaBayes* console and choose the option `Open`. A dialog then appears, asking you to select the file to be loaded. You can load files in several formats; in the process of loading data, the system prints out some messages indicating how the parser is behaving. Messages from the parsing procedure are printed at standard output at this point. Note that the `Open` option only works if you are running *JavaBayes* as an application. If you try to load a network in an applet, nothing happens; this is due to the security restrictions that are enforced within applets.

2. Load the network from the World-Wide-Web. To load a network, go to the `File` menu in the *Javabayes* console and choose the option `Open URL`. A dialog appears, asking you to insert the URL (Uniform Resource Locator) that specifies the address of the file in the World-Wide-Web. Note that the complete URL must be given here; inserting just the name of the file does not suffice. The `Open URL` option works both for applications and applets (but applets have the restriction that the file to be loaded must come from the same host as the applet itself).

3. Create a network from scratch. This option is considered later.

Once a network is in the system, it can be saved to local disk. To save a network, go to the `File` menu in the *JavaBayes* console and choose either the option `Save` or the option `Save as`.

JavaBayes can *load* networks in different formats: the BIF format 0.1, the BIF format 0.15, the XMLBIF format 0.2 and the XMLBIF format 0.3. *JavaBayes* can also *save* networks in different formats: the BIF format 0.15, the XMLBIF format 0.3, and the BUGS format. The idea is that the BIF format 0.15 supercedes the BIF format 0.1, so there is no need to save anything in version 0.1. On the other hand, XMLBIF format 0.2 is a bit experimental (based on XML), and quite different from 0.15, so both BIF 0.15 and XMLBIF 0.3 are fully supported right now. The BUGS format is described in the documentation of the BUGS system, a complete interface and engine for Bayesian inferences through Gibbs sampling. The BUGS system offers the most general approach to the calculation of posterior probability values in probabilistic models; it can be used to process arbitrary networks generated in *JavaBayes*.

Once a network is loaded into *JavaBayes*, its graph is displayed in the editor window. After you work with the network, you can clear the current network in the editor, going to the `File` menu and choosing the `Clear` option. Note that if you load a network, any network that was loaded is cleared.

You can obtain information about the network and its variables using the `Edit` modes in the editor window. There are two types of interaction in *JavaBayes*: interaction with the editor window and interaction with the console window. The operations described above (opening, saving, clearing, etc) are all controlled by menus in the console window. Options in the console window handle files and control the overall behavior of the system. On the other hand, the editor window controls operations that focus on the creation and manipulation of a Bayesian network in the system. These operations are controlled by buttons on the top and bottom of the editor window.

5.1.2 Editing the DogProblem network

Once a network is loaded into *JavaBayes*, interaction with the network occurs in the editor window, and results appear in the console window. The editor window can be in a number of *modes*. The buttons in the editor window control which mode is active. Each mode of the editor window interprets mouse clicks in a particular way. The modes are: `Create`, `Move`, `Delete`, `Query`, `Observe`, `Edit Variable`, and `Edit Function`.

In several modes, you can use the mouse to draw a rectangular area around a number of nodes. To do that, click on a point that is not inside a node, and then drag the mouse. The nodes inside this rectangular area form a *group*; they can be moved and deleted together.

Note that the button `Edit Network` does not represent a mode, because it does not change the meaning of operations in the editor drawing area. The `Edit Network` button simply calls a dialog window that allows you to edit the characteristics of the network. Try the `Edit Network` with the DogProblem network. You have the option of changing the name of the network, inserting or modifying properties of the network. You can also specify global neighborhoods for the network (see the discussion of robustness analysis (Section 7)).

The `Edit Variable` and `Edit Function` activate different modes in the editor window. In the `Edit Variable` mode, any mouse click over a node produces a dialog that allows you to edit the characteristics of the node. In the `Edit Function` mode, any mouse click over a node produces a dialog that allows you to edit the probability function associated to the node variable.

Try editing the contents of some nodes in the DogProblem network. Start with the `Edit Variable` mode. A dialog appears when you click in a node. This dialog allows you to edit the name of the variable, the properties of the variable, the properties of the probability distribution associated with the node, and select the type of variable and the type of distribution associated with the node:

- A property is a string containing information that is deemed relevant to the variable. Any arbitrary string can be a property. Usually, the first word of the string is the property name and the remainder of the string is the actual property value, but this is not mandatory.

- A variable can be *explanatory* or not. The concept of explanation used in *JavaBayes* is simple, but it requires some understanding of Bayesian statistics. The idea is that some evidence is entered into the network, and the best explanation is the configuration of variables that maximizes the probability of the evidence. You may be interested in a configuration that includes all variables in the network, or you may be interested in a configuration that is limited to some variables. In the first case, you want a *complete explanation*. In the second case, you want an explanation for the *explanatory variables* only. The idea is that some variables are special; when you request an explanation, the system finds the configuration of explanatory variables that maximizes the probability of the evidence. Note that explanations are produced when the menu `Inference mode` (under the `Options` menu in the editor window) is properly used; details are provided later.

- A variable can be associated to a single conditional probability distribution or to a convex set of conditional probability distributions (a *credal set*). Models that employ convex sets of distributions are useful to represent imprecise or incomplete beliefs; they are commonly used to analyze the robustness of probabilistic models to perturbations and parameter variations [1]. You can indicate that a particular variable is associated with a credal set. Note that each vertex is a probability table; you can specify all probability densities in the `Edit Function` dialog.

It is possible to define and modify properties for the network (in the `Edit Network` dialog), and properties for the variable and the distribution in a network node (in the `Edit Variable` dialog).

After you familiarize yourself with the `Edit Variable` dialog, try editing the probability values that are associated with every variable in the network. There are two ways to edit probability values.

First, you can switch the editor window to `Edit Function` mode; in this mode, you obtain a dialog for the probability values of any node that gets a mouse click. When you are reading the tables that display conditional distributions, note that the parents are always laid in the horizontal; make sure you understand the meaning of the entries before entering values.

The second way to visualize or edit the probability values is to swith the editor window to `Edit Variable`, click on a node, and then click on the `Edit Function` button in the `Edit Variable` dialog.

Note an important point concerning the editing dialogs. When you edit aspects of the network in the editing dialogs, the changes do not affect the network until you press the Apply button. If you dismiss the dialog, the changes are lost. Changes are effective after you press Apply (and cannot be undone after that); after you press Apply, you can dismiss the dialog and the changes are retained.

Note the following: in the `Edit Variable` dialog, there is a button that calls the `Edit Function` dialog. Changes in the type of the variable are only effective after you press Apply; only changing the type of the variable and calling the `Edit Function` dialog will generate an `Edit Function` dialog that follows the currently applied type.

5.1.3 Modifying the DogProblem network

JavaBayes contains a number of modes that allow you to modify a network in a graphical manner. The `Create`, `Move` and `Delete` modes are quite easy to understand: you can create arcs and nodes, move nodes, delete arcs and nodes.

Once in `Create` mode, click on any point not occupied by a node, and a node will appear. Nodes receive default names and are associated to default (uniform) distributions when they are created; you have to use the `Edit Variable` and `Edit Function` dialogs to edit them. If you click on a node and drag the mouse, an arrow will be created, stemming from the node where you first clicked. If you drag the mouse until a second node, an arc will be created between the nodes. The second node is a child of the first node; the first node is a parent of the second node. The system does not allow you to create cyclic structures, nor it allows you to create an arrow to a node from itself.

You can delete nodes and arcs in `Delete` mode. To delete a node or an arc, just click on it. If a group of nodes has been created, then clicking on any node in the group causes the whole group to be deleted.

You can mode nodes in `Move` node. To move a node, just click on it and drag it. You cannot move arcs, as the position of an arc is solely defined by the position of the nodes the arc is associating. If a group of nodes has been created, then clicking and draggin any node in the group causes the whole group to be moved.

5.1.4 Querying the DogProblem network

JavaBayes allows you to process the information in a Bayesian network in a variety of ways. Consider the following operations on the DogProblem network.

Suppose you want to calculate posterior probabilities for some variables in the DogProblem network. For example, you can calculate $p(l|h, b^c)$, the posterior probability of l given h and b^c (light-on given hear-bark and not bowel-problem). The Artificial Intelligence community usually calls this the "belief" in l given the "evidence" h and b^c. Obtaining the belief given the evidence is usually called "belief updating". To do that, you have to set hear-bark as true and bowel-problem as false; that's the evidence.

Variables are set when the editor window is in `Observe` mode. To set variables for a variable, click on the variable. A dialog will appear, giving you the opportunity to insert observations. If you want to indicate that a variable was observed, click on the observed value. If you want to indicate that a variable was not observed, then make sure the checkbox on the top of the dialog indicates no observation. Note that if you use the checkbox to indicate that an observation was made (without actually clicking on any specific value), the system will automatically take the first value in the list as the observed value.

After you insert the evidence, you can query the variable light-on by switching the editor window to `Query` mode. Click the mouse on the variable light-on; *JavaBayes* produces:

```
Posterior distribution:
probability ( "light-on" ) {//1 variable(s) and 2 values
        table 0.236519 0.763481 ;
}
```

JavaBayes allows you to calculate several probabilistic quantities involved in problems like this. There are essentially four types of calculations that are possible in *JavaBayes*. You can query any variable for its posterior probability distribution. Or you can query any variable for its posterior expectation. A query is obtained by clicking on the variable of interest. Alternatively, you may obtain explanation for a set of variables or for all variables in the network. An explanation is a configuration of variables that maximizes the probability of the evidence; you get an explanation by clicking on any variable.

Regarding expectations, *JavaBayes* handles *only* univariate utility functions; a particular case is the expected value for a variable, and this is what you can obtain in the graphical interface. Suppose you declare a variable like this:

```
variable  "LowLLapse" { //4 values
    type discrete[4] {  "CloseToDryAd"  "Steep"  "ModerateOrLe"  "Stable" };
    property "position = (857, 440)" ;
}
```

In this case, the expectation is calculated assuming the following values:

CloseToDryAd	0
Steep	1
ModerateOrLe	2
Stable	3

If you want different values, then you could do something like this:

```
variable  "LowLLapse" { //4 values
    type discrete[4] {  "-4.20"  "1.0"  "14.22"  "100.0" };
    property "position = (857, 440)" ;
}
```

To obtain an explanation, *JavaBayes* finds the configuration for a set of "explanatory" variables such that the posterior probability of the evidence is maximized. Maximization of probabilities is usually called maximum a posteriori estimation; a particular case that has received great attention in the literature is the most probable explanation problem (called MPE), where all variables are explanatory variables. *JavaBayes* lets the user choose which one of these two problems is to be solved. If the user wants to estimate some variables, then the user must indicate which variables are explanatory in the `Edit Variable` dialog.

5.1.5 Some other miscellaneous operations in *JavaBayes*

You can decide what to show in the console window after an inference is performed. By default, *JavaBayes* displays a short message indicating the final values of interest in any query. *JavaBayes* can also display the whole network that is used in processing a query. It is also possible to check the full bucket tree generated in processing a query (the bucket tree is the basic data structure used internally in *JavaBayes*). The bucket tree works best as a debugging tool; for a large network, it may be an overwhelming amount of information. Note that to actually display the information you want, you have to perform a query in the network.

The console area does not scroll indefinitely; the size of the scrolling buffer is controlled by the Java distribution you have, not by *JavaBayes*. Because Java displays only a limited number of lines, information may be lost (and note that the maximum number of lines that is actually displayed is not accessible by the program). To clear the text in the *JavaBayes* console and send the contents of the console to a file, try the `Dump console` option in the File menu. This option asks for a filename, dumps all the contents of the console window into the indicated file, and clears the console window. This option is also appropriate if you are interested in logging your actions or inferences during a *JavaBayes* session.

5.2 The Alarm network

Now that the small DogNetwork has been explored, you can try several other networks that are available in the *JavaBayes* distribution. There are some other small networks, like the Asia and the Cancer networks, and some relatively large networks, like the Alarm and the Hailfinder networks.

The Alarm network is well-known as it is relatively old and has been used in a large number of studies about inferences and about learning. The network was created to model situations that arise in medicine [5]; it contains lots of symptons, illnesses, exams, and other medical terms. Even though you can experiment with all example networks, the Alarm network is particularly nice as it is relatively large, but not too much; it is relatively sparse, but not too much; and it is based on real situations that have real meaning.

First, go to the `File` menu and pick the `Open` item. Now select the file containing the Alarm network; you will see the network appear in the editor window. If you have a large computer screen, you may be able to see the entire network; if you cannot see the entire network in a single screen, you have to use the scrollbars to view the network in pieces.

Try inserting some evidence randomly in the network, and then query some variables (first use the `Observe` mode, then swith to `Query` mode). You will realize that inference with the Alarm network is slower than inference with the DogProblem network, but not substantially so. *JavaBayes* starts any query by selecting only those variables that are necessary for the calculation of posterior probabilities; in a relatively sparse network like the Alarm network, it is usually the case that only a fraction of the network is used in any given query.

Try different inference algorithms, and check if you can feel the difference in speed. In actually, both algorithms process single inferences quickly, so it is almost impossible to feel any difference between them.

To familiarize yourself with the network, check the `Edit network`, `Edit variable` and `Edit function` dialogs, and select some variables as explanatory. Then query the network for posterior probabilities, posterior expectations, best explanation for the complete network and best explanation for the explanatory variables. See if you can grasp the meaning of explanations. Note that nodes turn orange when they are used as explanatory variables. So if you ask for a complete explanation, all nodes (except evidence nodes) turn orange.

And if you ask for an explanation only for explanatory variables, then only the explanatory variables turn orange.

Try viewing the network and the buckets in the console window, by setting the corresponding options in the `Options` menu. After a few inferences, the console window will be full; try dumping its contents into a file.

At this point, you probably have understood almost everything in *JavaBayes*, and you are ready to use the system for your own networks.

Chapter 6

Loading and saving data in *JavaBayes*

Data can be locally loaded/saved when you use JavaBayes as an application. Note that applets cannot load/save data (they are forbidden by the browsers)!

Applications and applets can read Bayesian networks through the Internet; this opens the possibility that *JavaBayes* be used to help process and organize the huge amounts of data and knowledge in the Internet.

This section contains a detailed description of the formats that can be manipulated by *JavaBayes*. If you have no interest on this kind of information (if you are not reading/writing files for *JavaBayes*), you can skip this section entirely.

6.1 All the formats

There are three different formats, and all three are supported by *JavaBayes* in the sense that *JavaBayes* can read files written on them.

The Bayesian Interchange Format version 0.1 (BIF 0.1) is a simple format, that has been succesfully used to represent a variety of networks. But BIF 0.1 had certain problems, and has been replaced by BIF version 0.15. BIF 0.15 is a more mature format and should work for most applications.

XMLBIF 0.3 is an experimental format, based on the new XML specification. The best way to understand it is to read about BIF 0.15, then read something about XML, then read the description of XMLBIF 0.3.

Because BIF 0.15 supercedes BIF0.1, *JavaBayes* does not save files in BIF 0.1 anymore. You can choose between XMLBIF 0.3 and BIF 0.15 in the Options menu.

Note that no format supports Noisy functions (since *JavaBayes* does not support those functions yet). The BIF formats also use the general concept of a *property*; implementations of the BIF format can use specific properties. *JavaBayes* handles some properties, such as *observed*, *explanation* and *credal-set*, which are explained later on.

For files, any extension is possible, but the extension *bif* is recommended for BIF 0.15, and the extension *xml* is tentatively used for XMLBIF 0.3.

6.2 Representing probability values

It is important to understand how the *JavaBayes* formats handle the specification of probability values. All distributions are specified as arrays of real numbers, and the meaning of the numbers depends on the definition of the distribution. Note that the same representation is used in internal arrays to store and manipulate probability values.

The distribution $p(f)$ in the example above can be specified as follows:

$$0.15, 0.85$$

Let's consider a more complicated example. The function $p(d|f,b)$ is given by

$$0.99, 0.90, 0.97, 0.30, 0.01, 0.10, 0.03, 0.70$$

The logic is simple: proceed as if you were filling a table, where the indices of the table vary from the right to left (in the example above, it is like binary counting because all variables have only two values).

A more complicated example would be a function $p(A|B,C)$ where A has 3 values, B has 2 values and C has 4 values. The function is represented as:

$$p(A_1|B_1C_1)p(A_1|B_1C_2)p(A_1|B_1C_3)p(A_1|B_1C_4)$$

$$p(A_1|B_2C_1)p(A_1|B_2C_2)p(A_1|B_2C_3)p(A_1|B_2C_4)$$

$$p(A_2|B_1C_1)p(A_2|B_1C_2)p(A_2|B_1C_3)p(A_2|B_1C_4)$$

$$p(A_2|B_2C_1)p(A_2|B_2C_2)p(A_2|B_2C_3)p(A_2|B_2C_4)$$

$$p(A_3|B_1C_1)p(A_3|B_1C_2)p(A_3|B_1C_3)p(A_3|B_1C_4)$$
$$p(A_3|B_2C_1)p(A_3|B_2C_2)p(A_3|B_2C_3)p(A_3|B_2C_4).$$

IMPORTANT: Notice that there is some redundancy in the values, because all probability functions must add up to one. Right now the BayesianNetworks package does not attempt to fill blanks or ensure consistency; the user has to provide the data in the correct format (it has to have the correct number of values, has to add to one, etc).

6.3 BIF version 0.15

White spaces, tabs and newlines are ignored; the C/C++ style of comments is adopted. The "," character is also ignored when it occurs between tokens.

The basic unit of information is a *block*: a piece of text which starts with a keyword and ends with the end of an attribute list (to be explained later). Arbitrary characters are allowed between blocks. This allows the user to insert arbitrarily long comments outside the blocks. It also allows user-specific blocks and commands to be placed outside the standard blocks.

Other than blocks, the BIF 0.15 refers to three entities: words, non-negative integers and non-negative reals.

A *word* is a contiguous sequence of characters, with the restriction that the first character be a letter. Characters are letters plus numbers plus the underline symbol (_) plus the dash symbol (-).

A *non-negative number* is a sequence of numeric characters, containing a decimal point or an exponent or both.

6.3.1 Blocks

A block is a unit of information. The general format of a block is:

```
block-type block-name {
  attribute-name  attribute-value;
  attribute-name  attribute-value;
```

```
    attribute-name  attribute-value;
  }
```

with as many attributes as necessary. The closing semicolon is mandatory after each attribute.

There are three possible blocks: *network*, *variable* and *probability* blocks.

- A network block defines the name of the network and lists the properties. Example:

```
network "Robot-Planning" {
   property version 1.1;
   property author Nobody;
}
```

- Variable blocks define the variables in a network. Example:

```
variable Leg {
  type discrete[2] { long, short };
  property temporary yes;
}
```

- Probability blocks specify the (conditional) probability tables (CPTs) for these variables, and hence the topology of the network. The block indicates the variables of the probability distribution right after the keyword probability. Example:

```
probability ( "Leg" | "Arm" ) {
   table 0.1 0.9 0.9 0.1;
}
```

The blocks must be placed in the following order:

- A network declaration block (one, must be first).
- A series of variable declaration blocks and probability definition blocks, possibly intermixed.

6.3.2 Attributes

Several attributes are defined at this point: *property*, *type*, *table*, *default* and entry attributes (the entry attribute is not associated with any keyword).

The attribute property can appear in all types of blocks. A property is just a string of arbitrary text to be associated with a block. Examples of properties:

```
property "size 12";
property "name Trial number ten";
```

Any text is valid in the string following keyword *property*. The idea is to store information that is specific to a particular system or network in the properties. Any number of property attributes can appear in a block.

The type attribute is specific to variable blocks. The property type lists the values of a discrete variable:

```
type discrete[ number-of-values ] { list-of-values };
```

The number-of-values token is a non-negative integer which indicates how many different values this variable may assume (the size of the list-of-values). The list-of-values is a sequence of words, each one the name of a variable value.

There are attributes that are specific to probability blocks (these attributes are discussed in the next section):

- table lists a sequence of non-negative real numbers.
- default lists a sequence of non-negative real numbers.
- the entry attribute, which is not associated with any keyword.

6.3.3 The *JavaBayes* properties

JavaBayes uses a number of properties to load and save information about Bayesian networks:

- The *observed* property for a variable. Suppose you have the following block:

  ```
  variable "light-on" {//2 values
          type discrete[2] { "true" "false" };
          property  "position = (218, 195)" ;
  }
  ```

  and you want to indicate that variable light-on is observed with value *true* (i.e., light-on = true is the evidence). You do this with the observed property:

  ```
  variable "light-on" {//2 values
  property "observed true";
          type discrete[2] { "true" "false" };
          property  "position = (218, 195)" ;
  }
  ```

  You can set as many variables as you want as observed; the syntax is simple:

  ```
  property observed [ observed-value ];
  ```

- The *explanation* property for a variable. Suppose you have the following block:

  ```
  variable "light-on" {//2 values
          type discrete[2] { "true" "false" };
          property "position = (218, 195)" ;
  }
  ```

  and you want to indicate that variable light-on is to be estimated. You can set light-on as a *explanation variable*, i.e., a variable which will be estimated. The meaning of a explanatory variable is that you would like to know which value for the variable would produce the highest probability or expectation. It is not necessarily true that you can operate on the variable and change it at will; it is just that you want to know which value would be best in the face of evidence. You do set explanatory variables with the explanation property:

  ```
  variable light-on{//2 values
  property "explanation";
          type discrete[2] { "true" "false" };
          property  "position = (218, 195)" ;
  }
  ```

If you request *JavaBayes* to produce the "best" configuration for the explanation variables, *JavaBayes* will only process the variables that are marked through an explanation property. You can set as many variables as you want as explanation variables; the syntax is simple:

```
property "explanation";
```

There are also properties that are related to robustness analysis in *JavaBayes*. Since robustness analysis is still an ongoing research project, the support for it is minimal. If you want to use robustness analysis now, please send me email. The properties related to robustness analysis always start with the keyword *credal-set*; if you are defining your own properties, please do not use this keyword.

6.3.4 Probability Blocks

Probability blocks are used to define the actual network topology and conditional probability tables.

An example of a standard probability block is:

```
probability("GasGauge" | "Gas", "BatteryPower") {
           ("yes", "high") 0.999 0.001;
           ("yes", "low") 0.850 0.150;
           ("yes", "medium") 0.000 1.000;
           ("no", "high") 0.000 1.000;
           ("no", "low") 0.000 1.000;
           ("no", "medium") 0.000 1.000;
}
```

As explained before, the symbol ',' is ignored between tokens so it does not affect the list of variables given after the keyword probability. The variables however must be enclosed by parenthesis.

The example above uses the entry attribute, which is different from the other attributes in that it has no keyword. It simply starts with an opening parenthesis, and has a list of values for all the conditioning variables. After the closing parenthesis, a list of probability values for the first variable is given (the user must provide numbers that add to 1, but this is not mandatory).

The probability vectors can be listed in any order, since the names in parentheses uniquely identify the parent instantiation.

In addition to the entry attribute, the BIF 0.15 supports the concept of a default entry. So the above CPT could have been specified equivalently as:

```
probability("GasGauge" | "Gas", "BatteryPower") {
            default 0.000 1.000;
            ("yes", "low")  0.850 0.150;
            ("no", "medium") 0.000 1.000;
}
```

Note that each number is a separate token, so we can use "," between numbers.

Another way to define a probability distribution is through the table attribute. The body of such attribute is a sequence of non-negative real numbers, in the counting order of the declared variables (if all variables were binary, we would say binary counting with least significant digit in the right). So, for the example above, we could simply say:

```
probability("GasGauge" | "Gas", "BatteryPower") {
           table 0.999 0.850 0.0 0.0 0.0 0.0 0.001 0.15 1.0 1.0 1.0 1.0;
}
```

There are some subtle rules that regulate these declarations.

- If multiple default declarations exist, only the last one is valid.
- If multiple table declarations exist, only the last one is valid.
- A table can contain more elements than the necessary to specify a distribution; the excess elements are discarded.
- A table can contain less elements than the necessary to specify a distribution, which is then padded with zeros.
- Specified entries override conflicting default and table declarations.

6.3.5 Examples

Here are some of the available examples:

- dog-problem.bif, a very simple network based on the discussion at Charniak, E., Bayesian Networks without Tears, AI Magazine, 1991.
- elimbel2.bif, a simple network based on the second example in the Elimbel system.
- car-starts.bif, a somewhat large network contributed by Sreekanth Nagarajan, based on the automobile belief network that David Heckerman and Jack Breese presented in the March, 1995 issue of Communications of the ACM.
- alarm.bif, the famous *Alarm* network.

Here is the dog-problem.bif network:

```
// Bayesian Network in the Interchange Format
// Produced by BayesianNetworks package in JavaBayes
// Output created Sun Nov 02 17:49:49 GMT+00:00 1997
// Bayesian network
network "Dog-Problem" { //5 variables and 5 probability distributions
property "credal-set constant-density-bounded 1.1" ;
}
variable  "light-on" { //2 values
type discrete[2] {  "true"  "false" };
property "position = (218, 195)" ;
}
variable  "bowel-problem" { //2 values
type discrete[2] {  "true"  "false" };
property "position = (335, 99)" ;
}
variable  "dog-out" { //2 values
type discrete[2] {  "true"  "false" };
property "position = (300, 195)" ;
}
variable  "hear-bark" { //2 values
type discrete[2] {  "true"  "false" };
property "position = (296, 268)" ;
```

```
}
variable  "family-out" { //2 values
type discrete[2] {  "true"  "false" };
property "position = (257, 99)" ;
}
probability (  "light-on"  "family-out" ) { //2 variable(s) and 4 values
table 0.6 0.05 0.4 0.95 ;
}
probability (  "bowel-problem" ) { //1 variable(s) and 2 values
table 0.01 0.99 ;
}
probability (  "dog-out"  "bowel-problem"  "family-out" ) { //3 variable(s) and 8 values
table 0.99 0.97 0.9 0.3 0.01 0.03 0.1 0.7 ;
}
probability (  "hear-bark"  "dog-out" ) { //2 variable(s) and 4 values
table 0.7 0.01 0.3 0.99 ;
}
probability (  "family-out" ) { //1 variable(s) and 2 values
table 0.15 0.85 ;
}
```

6.4 BIF version 0.1

White spaces, tabs and newlines are ignored; the C/C++ style of comments is adopted. Two other characters are also ignored when they occur between tokens: "," and "—". These characters can be used to separate variables in the definition of a probability distribution.

The basic unit of information is a *block*: a piece of text which starts with a keyword and ends with the end of an attribute list (to be explained later). Arbitrary characters are allowed between blocks. This allows the user to insert arbitrarily long comments outside the blocks. It also allows user-specific blocks and commands to be placed outside the standard blocks.

Other than blocks, the BIF 0.1 refers to three entities: words, non-negative integers and non-negative reals.

A *word* is a contiguous sequence of characters, with the restriction that the first character be a letter. Characters are letters plus numbers plus the underline symbol (_) plus the dash symbol (-).

A *non-negative number* is a sequence of numeric characters, containing a decimal point or an exponent or both.

6.4.1 Blocks

A block is a unit of information. The general format of a block is:

```
block-type block-name {
  attribute-name  attribute-value;
  attribute-name  attribute-value;
  attribute-name  attribute-value;
}
```

with as many attributes as necessary. The closing semicolon is mandatory after each attribute.

There are three possible blocks: *network*, *variable* and *probability* blocks.

- A network block defines the name of the network and lists the properties. Example:

```
network Robot-Planning {
   property version 1.1;
   property author Nobody;
}
```

- Variable blocks define the variables in a network. Example:

```
variable Leg {
  type discrete[2] { long, short };
  property temporary yes;
}
```

- Probability blocks specify the (conditional) probability tables (CPTs) for these variables, and hence the topology of the network. The block indicates the variables of the probability distribution right after the keyword probability. Example:

```
probability ( Leg | Arm ) {
   table 0.1 0.9 0.9 0.1;
}
```

The blocks must be placed in the following order:

- A network declaration block (one, must be first).
- A series of variable declaration blocks and probability definition blocks, possibly intermixed.

6.4.2 Attributes

Several attributes are defined at this point: *property*, *type*, *table*, *default* and entry attributes (the entry attribute is not associated with any keyword).

The attribute property can appear in all types of blocks. A property is just a string of arbitrary text to be associated with a block. Examples of properties:

```
property size 12;
property name "Trial number ten";
```

Any text is valid between the keyword property and the ending semicolon. The idea is to store information that is specific to a particular system or network in the properties. Any number of property attributes can appear in a block.

The type attribute is specific to variable blocks. The property type lists the values of a discrete variable:

```
type discrete[ number-of-values ] { list-of-values };
```

The number-of-values token is a non-negative integer which indicates how many different values this variable may assume (the size of the list-of-values). The list-of-values is a sequence of words, each one the name of a variable value.

There are attributes that are specific to probability blocks (these attributes are discussed in the next section):

- table lists a sequence of non-negative real numbers.
- default lists a sequence of non-negative real numbers.
- the entry attribute, which is not associated with any keyword.

6.4.3 The *JavaBayes* properties

JavaBayes uses a number of properties to load and save information about Bayesian networks:

- The *observed* property for a variable. Suppose you have the following block:

```
variable light-on{//2 values
        type discrete[2] { true false };
        property  position = (218, 195) ;
}
```

  and you want to indicate that variable light-on is observed with value *true* (i.e., light-on = true is the evidence). You do this with the observed property:

```
variable light-on{//2 values
property observed true;
        type discrete[2] { true false };
        property  position = (218, 195) ;
}
```

  You can set as many variables as you want as observed; the syntax is simple:

```
property observed [ observed-value ];
```

- The *explanation* property for a variable. Suppose you have the following block:

```
variable light-on{//2 values
        type discrete[2] { true false };
        property  position = (218, 195) ;
}
```

and you want to indicate that variable light-on is to be estimated. You can set light-on as a *explanation variable*, i.e., a variable which will be estimated. The meaning of a explanatory variable is that you would like to know which value for the variable would produce the highest probability or expectation. It is not necessarily true that you can operate on the variable and change it at will; it is just that you want to know which value would be best in the face of evidence. You do set explanatory variables with the explanation property:

```
variable light-on{//2 values
property explanation;
        type discrete[2] { true false };
        property  position = (218, 195) ;
}
```

If you request *JavaBayes* to produce the "best" configuration for the explanation variables, *JavaBayes* will only process the variables that are marked through an explanation property. You can set as many variables as you want as explanation variables; the syntax is simple:

```
property explanation;
```

There are also properties that are related to robustness analysis in *JavaBayes*. Since robustness analysis is still an ongoing research project, the support for it is minimal. If you want to use robustness analysis now, please send me email. The properties related to robustness analysis always start with the keyword *credal-set*; if you are defining your own properties, please do not use this keyword.

6.4.4 Probability Blocks

Probability blocks are used to define the actual network topology and conditional probability tables.

An example of a standard probability block is:

```
probability(GasGauge | Gas, BatteryPower) {
           (yes, high) 0.999 0.001;
           (yes, low) 0.850 0.150;
           (yes, medium) 0.000 1.000;
```

```
            (no, high) 0.000 1.000;
            (no, low) 0.000 1.000;
            (no, medium) 0.000 1.000;
}
```

As explained before, the symbols "—" and "," are ignored between tokens so they do not affect the list of variables given after the keyword probability. The variables however must be enclosed by parenthesis.

The example above uses the entry attribute, which is different from the other attributes in that it has no keyword. It simply starts with an opening parenthesis, and has a list of values for all the conditioning variables. After the closing parenthesis, a list of probability values for the first variable is given (the user must provide numbers that add to 1, but this is not mandatory).

The probability vectors can be listed in any order, since the names in parentheses uniquely identify the parent instantiation.

In addition to the entry attribute, the BIF 0.1 supports the concept of a default entry. So the above CPT could have been specified equivalently as:

```
probability(GasGauge | Gas, BatteryPower) {
            default 0.000 1.000;
            (yes, low)  0.850 0.150;
            (no, medium) 0.000 1.000;
}
```

Note that each number is a separate token, so we can use "," and "—" between numbers; these symbols are ignored.

Another way to define a probability distribution is through the table attribute. The body of such attribute is a sequence of non-negative real numbers, in the counting order of the declared variables (if all variables were binary, we would say binary counting with least significant digit in the right). So, for the example above, we could simply say:

```
probability(GasGauge | Gas, BatteryPower) {
           table 0.999 0.850 0.0 0.0 0.0 0.0 0.001 0.15 1.0 1.0 1.0 1.0;
}
```

There are some subtle rules that regulate these declarations.

- If multiple default declarations exist, only the last one is valid.
- If multiple table declarations exist, only the last one is valid.
- A table can contain more elements than the necessary to specify a distribution; the excess elements are discarded.
- A table can contain less elements than the necessary to specify a distribution, which is then padded with zeros.
- Specified entries override conflicting default and table declarations.

6.4.5 Example

Here is the dog-problem.bif network in BIF0.1:

```
// Bayesian Network in the Interchange Format
// Produced by BayesianNetworks package in JavaBayes
// Output created Tue Feb 25 12:55:25  1997
// Bayesian network
network Internal-Network{ //5 variables and 5 probability distributions
}
variable light-on{//2 values
type discrete[2] { true false };
property  position = (218, 195) ;
}
variable bowel-problem{//2 values
type discrete[2] { true false };
property  position = (335, 99) ;
}
variable dog-out{//2 values
type discrete[2] { true false };
property  position = (300, 195) ;
}
variable hear-bark{//2 values
type discrete[2] { true false };
property  position = (296, 268) ;
}
variable family-out{//2 values
```

```
type discrete[2] { true false };
property  position = (257, 99) ;
}
probability ( light-on family-out ) { //2 variable(s) and 4 values
table 0.6 0.05 0.4 0.95 ;
}
probability ( bowel-problem ) { //1 variable(s) and 2 values
table 0.01 0.99 ;
}
probability ( dog-out bowel-problem family-out ) { //3 variable(s) and 8 values
table 0.99 0.97 0.9 0.3 0.01 0.03 0.1 0.7 ;
}
probability ( hear-bark dog-out ) { //2 variable(s) and 4 values
table 0.7 0.01 0.3 0.99 ;
}
probability ( family-out ) { //1 variable(s) and 2 values
table 0.15 0.85 ;
}
```

6.5 XMLBIF version 0.3

The XMLBIF format provides a different perspective for the storage and manipulation of Bayesian networks. Instead of focusing on a readable and simplified description of Bayesian networks, the XMLBIF format emphasizes ease of distribution through wide area networks. The XMLBIF format is defined through XML, a dialect of SGML that is used to specify formats. The advantage of XML is that it has industry-wide support, and many software developers plan to introduce parsers, search-engines, and browsers for XML. The power of XML is that it is a standard language for editing formats, and XMLBIF attempts to use XML to reduce to a minimum the burden of distributing graphical models to a large audience.

The XMLBIF format is actually quite similar to BIF 0.15, but it is stated in a manner that is XML-compliant. Note the similarity of XMLBIF to HTML; this happens because both HTML and XML are dialects of SGML.

White spaces, tabs and newlines are ignored. The XML style of comments and declarations is used to detect text that should be ignored: any character between <! and > is ignored. Note that XML comments should be enclosed by <! − − and −− >.

The XMLBIF format is defined by a set of XML-compliant tags. Other than XML tags, the XMLBIF 0.3 refers to three entities: words, non-negative integers and non-negative reals.

A *word* is a contiguous sequence of characters, with the restriction that the first character be a letter. Characters are letters plus numbers plus the underline symbol (_) plus the dash symbol (-).

A *non-negative number* is a sequence of numeric characters, containing a decimal point or an exponent or both.

Note that every XML file starts with the expression <?xml version="1.0"? >, indicating the XML version. Other attributes and directives can be contained within this tag; for example, the tag <?xml version="1.0" encoding="US-ASCII"? > specifies the file encoding. This initial tag is followed by any XML definitions and statements that define the DTD for the document (the DTD is always optional in XML).

6.5.1 Networks, variables and probabilities

The first tag of a XMLBIF 0.3 file is the <BIF> tag; the last tag is the closing < /BIF> tag. All the information about the model is contained between these tags. There are three basic units of information: *network*, *variable* and *probability densities.*

A network is defined by its name, followed by a list of properties (optional), followed by a list of variables and probability densities. For example, a network may be defined as:

```
<BIF VERSION="0.3">
<NETWORK>
<NAME>Dog-Problem</NAME>
<PROPERTY>date Sunday, 19 July, 1998</PROPERTY>
<PROPERTY>author John</PROPERTY>

variables and probabilities go here

</NETWORK>
</BIF>
```

The VERSION attribute in the BIF tag is mandatory.

Variables are defined by their names, types and properties:

```
<VARIABLE TYPE="chance">
<NAME>light-on</NAME>
<OUTCOME>true</OUTCOME>
<OUTCOME>false</OUTCOME>
<PROPERTY>position = (73, 165)</PROPERTY>
</VARIABLE>
```

Conditional probability densities can be specified in various ways inside the DEFINITION tag. One example is:

```
<DEFINITION>
<FOR>hear-bark</FOR>
<GIVEN>dog-out</GIVEN>
<TABLE>0.7 0.01 0.3 0.99 </TABLE>
</DEFINITION>
```

There is no mandatory order of variable and probability blocks.

A *property* is just a string of arbitrary text to be associated with a block. Examples of properties:

```
    <PROPERTY>size 12</PROPERTY>
    <PROPERTY>comment Trial number ten</PROPERTY>
```

Any text is valid in the string inside the PROPERTY opening and closing tags. The idea is to store information that is specific to a particular system or network in the properties. Any number of property attributes can appear in a block.

A variable is defined by a NAME tag (with the TYPE attribute), and its possible OUTCOMES:

```
<VARIABLE TYPE="chance">
<NAME>light-on</NAME>
<OUTCOME>true</OUTCOME>
<OUTCOME>false</OUTCOME>
<PROPERTY>position = (73, 165)</PROPERTY>
</VARIABLE>
```

Currently the content of a TYPE attribute must be the keyword either "chance" or "decision" or "utility".

The TABLE tag is specific to the DEFINITION block (note that a definition can be a probability distribution, a set of decision values or a set of utility values, depending on the TYPE attributes of the referred variable). DEFINITION blocks are used to define the actual network topology, by specifying conditional probability tables.

An example of a standard probability block is:

```
<DEFINITION>
<FOR>GasGauge</FOR>
<GIVEN>BatteryPower</GIVEN>
<GIVEN>GasInTank</GIVEN>
<TABLE>1.0 0.0 0.2 0.0 0.0 1.0 0.8 1.0 </TABLE>
</DEFINITION>
```

for a variable GasGauge that is defined with TYPE equal to "chance". The body of the TABLE tag is a sequence of non-negative real numbers, in the counting order of the declared variables (if all variables were binary, we would say binary counting with least significant digit in the right). If multiple table declarations exist, only the last one is valid.

6.5.2 The *JavaBayes* properties

JavaBayes uses a number of properties to load and save information about Bayesian networks:

- The *observed* property for a variable. Suppose you have the following block:

  ```
  <VARIABLE TYPE="chance">
  <NAME>light-on</NAME>
  <OUTCOME>true</OUTCOME>
  <OUTCOME>false</OUTCOME>
  <PROPERTY>position = (73, 165)</PROPERTY></VARIABLE>
  ```

  and you want to indicate that variable light-on is observed with value *true* (i.e., light-on = true is the evidence). You do this with the observed property:

```
<VARIABLE TYPE="chance">
<NAME>light-on</NAME>
<OUTCOME>true</OUTCOME>
<OUTCOME>false</OUTCOME>
<PROPERTY>observed true</PROPERTY>
<PROPERTY>position = (73, 165)</PROPERTY></VARIABLE>
```

You can set as many variables as you want as observed; the syntax is simple:

```
<PROPERTY>observed (observed-value)</PROPERTY>
```

- The *explanation* property for a variable. Suppose you want to indicate that variable light-on is to be estimated. You can set light-on as a *explanation variable*, i.e., a variable which will be estimated. The meaning of a explanatory variable is that you would like to know which value for the variable would produce the highest probability or expectation. It is not necessarily true that you can operate on the variable and change it at will; it is just that you want to know which value would be best in the face of evidence. You do set explanatory variables with the explanation property:

```
<VARIABLE TYPE="chance">
<NAME>light-on</NAME>
<OUTCOME>true</OUTCOME>
<OUTCOME>false</OUTCOME>
<PROPERTY>explanation</PROPERTY>
<PROPERTY>position = (73, 165)</PROPERTY></VARIABLE>
</VARIABLE>
```

If you request *JavaBayes* to produce the "best" configuration for the explanation variables, *JavaBayes* will only process the variables that are marked through an explanation property.

There are also properties that are related to robustness analysis in *JavaBayes*. Since robustness analysis is still an ongoing research project, the support for it is minimal. If you want to use robustness analysis now, please send me email. The properties related to robustness analysis always start with the keyword *credal-set*; if you are defining your own properties, please do not use this keyword.

6.5.3 Examples

Here are some of the available examples:

- dog-problem.xml, a very simple network based on the discussion at Charniak, E., Bayesian Networks without Tears, AI Magazine, 1991.
- elimbel2.xml, a simple network based on the second example in the Elimbel system.
- car-starts.xml, a somewhat large network contributed by Sreekanth Nagarajan, based on the automobile belief network that David Heckerman and Jack Breese presented in the March, 1995 issue of Communications of the ACM.
- alarm.bif, the famous *Alarm* network.

Here is the dog-problem.xml network:

```
<?xml version="1.0" encoding="US-ASCII"?>


<!--
Bayesian network in XMLBIF v0.3 (BayesNet Interchange Format)
Produced by JavaBayes (http://www.cs.cmu.edu/~javabayes/
Output created Wed Aug 12 21:16:40 GMT+01:00 1998
-->



<!-- DTD for the XMLBIF 0.3 format -->
<!DOCTYPE BIF [
<!ELEMENT BIF ( NETWORK )*>
      <!ATTLIST BIF VERSION CDATA #REQUIRED>
<!ELEMENT NETWORK ( NAME, ( PROPERTY | VARIABLE | DEFINITION )* )>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT VARIABLE ( NAME, ( OUTCOME |  PROPERTY )* ) >
      <!ATTLIST VARIABLE TYPE (chance|decision|utility) "chance">
<!ELEMENT OUTCOME (#PCDATA)>
<!ELEMENT DEFINITION ( FOR | GIVEN | TABLE | PROPERTY )* >
<!ELEMENT FOR (#PCDATA)>
```

```
<!ELEMENT GIVEN (#PCDATA)>
<!ELEMENT TABLE (#PCDATA)>
<!ELEMENT PROPERTY (#PCDATA)>
]>


<BIF VERSION="0.3">
<NETWORK>
<NAME>Dog-Problem</NAME>

<!-- Variables -->
<VARIABLE TYPE="chance">
<NAME>light-on</NAME>
<OUTCOME>true</OUTCOME>
<OUTCOME>false</OUTCOME>
<PROPERTY>position = (73, 165)</PROPERTY>
</VARIABLE>

<VARIABLE TYPE="chance">
<NAME>bowel-problem</NAME>
<OUTCOME>true</OUTCOME>
<OUTCOME>false</OUTCOME>
<PROPERTY>position = (190, 69)</PROPERTY>
</VARIABLE>

<VARIABLE TYPE="chance">
<NAME>dog-out</NAME>
<OUTCOME>true</OUTCOME>
<OUTCOME>false</OUTCOME>
<PROPERTY>position = (155, 165)</PROPERTY>
</VARIABLE>

<VARIABLE TYPE="chance">
<NAME>hear-bark</NAME>
<OUTCOME>true</OUTCOME>
<OUTCOME>false</OUTCOME>
<PROPERTY>position = (154, 241)</PROPERTY>
</VARIABLE>
```

```
<VARIABLE TYPE="chance">
<NAME>family-out</NAME>
<OUTCOME>true</OUTCOME>
<OUTCOME>false</OUTCOME>
<PROPERTY>position = (112, 69)</PROPERTY>
</VARIABLE>


<!-- Probability distributions -->
<DEFINITION>
<FOR>light-on</FOR>
<GIVEN>family-out</GIVEN>
<TABLE>0.6 0.05 0.4 0.95 </TABLE>
</DEFINITION>

<DEFINITION>
<FOR>bowel-problem</FOR>
<TABLE>0.01 0.99 </TABLE>
</DEFINITION>

<DEFINITION>
<FOR>dog-out</FOR>
<GIVEN>bowel-problem</GIVEN>
<GIVEN>family-out</GIVEN>
<TABLE>0.99 0.97 0.9 0.3 0.01 0.03 0.1 0.7 </TABLE>
</DEFINITION>

<DEFINITION>
<FOR>hear-bark</FOR>
<GIVEN>dog-out</GIVEN>
<TABLE>0.7 0.01 0.3 0.99 </TABLE>
</DEFINITION>

<DEFINITION>
<FOR>family-out</FOR>
<TABLE>0.15 0.85 </TABLE>
</DEFINITION>
```

```
</NETWORK>
</BIF>
```

Chapter 7

Robustness analysis in *JavaBayes*

The core inference engine in *JavaBayes* provides support for *robustness analysis* of Bayesian networks. Robustness analysis employs sets of distributions to model perturbations in the parameters of a probability distribution [1, 2, 14]. Robust Bayesian inference is the calculation of bounds on posterior values given such perturbations.

7.1 Motivation

In the real world we can rarely meet all the assumptions of a Bayesian model. First, we have to face imperfections in our beliefs, either because we have no time, resources, patience, or confidence to provide exact probability values. Second, we may deal with a group of disagreeing experts, each specifying a particular distribution. Third, we may be interested in abstracting away parts of a model and assessing the effects of this abstraction.

There is some empirical evidence that Bayesian networks are not too sensitive to parameters; this is due to the fact that many classic examples of Bayesian networks are sparse graphs, with probability values that are close to zero or one (for example, noisy functions have probability values that are only zero or one). When that happens, you're lucky because robustness is likely to be present. In other words, if changes in one variable do not affect many variables, and changes are not large relative to the magnitude of the numbers, then it is likely that these changes will not produce significant variations in inferences.

Situations where probability values are not very close to zero-one, or where the graph is heavily inter-connected, are situations where robustness may falter. Another situation is model building, where some parameter are not entirely specified, and the question is how much effort should be spent nailing down their values. A serious analysis of a network must consider the possibility of robustness problems, or at least assess how robust the model is. That's the aspect of inference that JavaBayes is trying to address.

Research on Bayesian networks has not fully explored the robustness analysis aspect of inference, due to the lack of algorithms for inferences with convex sets of distributions. *JavaBayes* is the first Bayesian network engine that provides facilities that explicitly account for perturbations in probabilistic models.

7.2 Algorithms for robust Bayesian analysis

Robust Bayesian inference in multivariate structures such as Bayesian networks is a complex algorithmic problem. Usually, the objective of robustness analysis is to obtain the interval that contains all values of a certain quantity of interest, given all possible perturbations in a probabilistic model. Models that attempt to combine Bayesian networks with probability intervals have faced great difficulties. Even though particular classes of probability intervals are amenable to analysis and some brute-force algorithms are possible, there has been no general model of probability intervals with the same style of efficient propagation used in Bayesian networks.

JavaBayes contains two classes of algorithms for robustness analysis:

- Local algorithms, where the perturbations in a Bayesian network are associated with the individual nodes of the network.
- Global algorithms, where perturbations are associated with the whole joint distribution represented by a Bayesian network.

If you have an application that requires use of robustness analysis, I would be grateful if you could send me email explaining what the application is and how you used the system.

The algorithms in *JavaBayes* employ some recent results to reduce the complexity of robustness analysis. The starting point is the theory of Quasi-Bayesian behavior, proposed in 1980 by Giron and Rios. This theory builds a complete decision making model based on convex sets of probability measures.

A complete discussion of all these issues and an exposition of algorithms can be found at http://www.cs.cmu.edu/~qbayes/Tutorial/.

7.3 Global neighborhoods

Consider first the algorithms for global robustness analysis, which are activated through the `Edit Network` dialog.

Suppose you set a network to represent a multivariate constant density ratio class. You can do this in the *Edit Network* dialog. If you save the network with the global neighborhood set (in the BIF 0.15 format), you should see the property:

```
network Example {
property credal-set constant-density-ratio 1.2;
}
```

When an inference is requested, the algorithm for global neighborhoods with density ratio classes will be called. The parameter that defines the class in the example is 1.2. If this parameter is smaller than zero, the parameter is automatically set to one (so that it has no effect); if it is smaller than one, then its inverse is used (the parameter has to be larger than one).

Take another example. Suppose a network is declared with the `credal-set epsilon-contaminated` property:

```
network Example {
property credal-set epsilon-contaminated 0.1;
}
```

then the algorithm for global neighborhoods with ϵ-contaminated classes will be called, using 0.1 as the definition parameter for the ϵ-contaminated class. If this parameter is smaller than zero or larger than one, inferences assume the parameter to be zero.

There are four possible global neighborhoods for a network:

```
network Example {
property credal-set constant-density-ratio 1.1;
}

network Example {
property credal-set epsilon-contaminated 0.1;
```

```
}

network Example {
property credal-set constant-density-bounded 1.1;
}

network Example {
property credal-set total-variation 0.1;
}
```

The parameter for the constant density bounded class behaves as the parameter for the constant density ratio class; the parameter for the total variation class behaves as the parameter for the ϵ-contaminated class.

If any of the `credal-set` properties above are present, the result is a pair of functions, the lower and the upper bounds for the posterior marginals.

Consider the example discussed in Section 5.1, taken from [4].

The problem represents several facts about a family with a dog; the dog barks under some circumstances, and the lights are on under some circumstances. Running this problem in *JavaBayes* as a standard Bayesian problem , you get:

```
Posterior distribution:
probability (  "light-on" ) { //1 variable(s) and 2 values
table 0.23651916875671802 0.763480831243282 ;
}
```

Now try to perform a robustness analysis by adding say an epsilon-contamination of 0.1. This roughly means that you expect the Bayesian network description to be correct 90 percent of the time, but in 10 percent of the cases you would expect any other joint distribution to be possible. Notice this is a somewhat radical model of uncertainty as you are allowing for 0.1 in probability mass to be concentrated in arbitrary sets or events. Add the following line in the network block:

```
property credal-set epsilon-contaminated 0.1;
```

and load the new network description into *JavaBayes*: You will get the result:

```
Posterior distribution:
 envelope (  "light-on" ) { //1 variable(s) and 2 values
table lower-envelope 0.21286725188104622 0.6871327481189539 ;
table upper-envelope 0.31286725188104625 0.7871327481189538 ;
}
```

These functions are the lower and upper bounds respectively.

7.3.1 Local neighborhoods

Local perturbations to a network can be inserted as sets of conditional densities associated with variables in the network. Each variable can be associated to a polytope in the space of densities.

To associate a variable with a set of densities, you have to insert the vertices of the set of densities. Go to the *Edit variable* window and mark some variable as a *Credal set with extreme points*. Insert the number of vertices of the set of distributions. Then edit these densities in the *Edit function* window.

You can also insert the vertices of a set of densities directly into the describing a network; *JavaBayes* simply asks you to determine which vertice you are referring to. In the example above, suppose you want to define an interval for the probability of family-out. You can write a file in the BIF0.15 format with the following declaration:

```
probability ( "family-out" ) { //1 variable(s) and 2 values
table 0.15 0.85 ;
table 0.25 0.75 ;
}
```

This defines an interval $0.15 \leq p(family-out) \leq 0.25$. You can also insert more vertices if that's appropriate, but note that for a binary variable that does not add any information.

If you insert the set of densities above, then you get:

```
Posterior distribution:
 envelope (  "light-on"  "<Transparent:family-out>" ) { //2 variable(s) and 2 values
table lower-envelope 0.23651916875671802 0.6792901716068643 ;
```

```
table upper-envelope 0.3207098283931358 0.763480831243282 ;
}
```

This indicates the lower and upper bounds for the probability of light-on given the evidence, and also indicates which sets of densities affect the result (in this case, the densities for family-out).

Bibliography

[1] J. O. Berger. *Statistical Decision Theory and Bayesian Analysis.* Springer-Verlag, 1985.

[2] J. O. Berger. Robust Bayesian analysis: Sensitivity to the prior. *Journal of Statistical Planning and Inference*, 25:303–328, 1990.

[3] J. S. Breese and K. W. Fertig. Decision making with interval influence diagrams. In P. P. Bonissone, M. Henrion, L. N. Kanal, and J. F. Lemmer, editors, *Uncertainty in Artificial Intelligence 6*, pages 467–478. Elsevier Science, North-Holland, 1991.

[4] E. Charniak. Bayesian networks without tears. *AI Magazine*, 1991.

[5] G. F. Cooper. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence*, 42:393–405, 1990.

[6] R. Dechter. Bucket elimination: a unifying framework for probabilistic inference. *Twelfth Annual Conference on Uncertainty in Artificial Intelligence*, 1996.

[7] M. DeGroot. *Optimal Statistical Decisions.* McGraw-Hill, New York, 1970.

[8] J. Earman. *Bayes or Bust?* The MIT Press, Cambridge, MA, 1992.

[9] F. V. Jensen. *An Introduction to Bayesian Networks.* Springer Verlag, New York, 1996.

[10] U. Kjaerulff. Triangulation of graphs — algorithms giving small total state space. Technical Report R-90-09, Department of Mathematics and Computer Science, Aalborg University, Denmark, March 1990.

[11] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kauffman, San Mateo, CA, 1988.

[12] J. Pearl. On probability intervals. *Int. Journal of Approximate Reasoning*, 2:211–216, 1988.

[13] L. A. Wasserman. Prior envelopes based on belief functions. *The Annals of Statistics*, 18(1):454–464, 1990.

[14] L. Wasserman. Recent methodological advances in robust Bayesian inference. In J. M. Bernardo, J. O. Berger, A. P. Dawid, and A. F. M. Smith, editors, *Bayesian Statistics 4*, pages 483–502. Oxford University Press, 1992.

[15] N. L. Zhang and D. Poole. Exploiting causal independence in Bayesian network inference. *Journal of Artificial Intelligence Research*, pages 301–328, 1996.