Bayesian Belief Network Propagation Engine In Java

Lukas Sklenar
University of Kent
ls5@kent.ac.uk

Abstract

This paper is dedicated to the propagation of data within Bayesian Belief Networks, and to the process regarding the development of a software engine, written in Java, to handle said propagation. It also provides the reader with a basic introduction to Bayesian Belief Network technology, as well as describing in some detail the challenges faced when propagating such complex structures.

1. Introduction

Recently, I have become interested in the potential use of intelligent prediction techniques on mobile and ubiquitous devices to predict what a user might want given his or her contextual information. In particular, I would like to focus on Bayesian Belief Networks as a possibility to do the above. Formally, a Bayesian Belief Network can be thought of as a compact representation of the joint probability distribution of a set of random variables[1]. When it is propagated, the BBN propagation algorithm provides an efficient way of calculating the corresponding distribution conditional on the observed values of the random variables. Essentially the network allows you to calculate the probabilities of inter-dependent events, allowing one to make predictions based on available data.

With this in mind, I have built a Bayesian Belief Network propagation engine written entirely in Java and in such a way as to be easily translatable into J2ME. This product could be used experimentally on mobile devices.

In Section 3 I will give a brief introduction to the underlying Bayesian technology, and in the later sections I will outline the propagation methods and describe the challenges encountered in my project.

2. Previous Work

There are a number of commercial Bayesian Belief Network (BBN) tools available today, most prominently Hugin and Netica. There are also some open source tools like JavaBayes and XBaies. While these tools are precise and effective, most of them do not fulfill the criteria I wanted in my tool. The reason for this is that most of them are meant to be desktop tools that deal with BBN creation (sometimes graphically) and propagation is usually only a small subset of what they can do. Moreover many of these tools try to go beyond just BBN propagation and try to incorporate many extras. (Novel ways of entering evidence or linking nodes, automated learning mechanisms, and so on.) This tends to make them rather bloated for the tasks I had in mind, in my opinion. A majority of them are also written in C, and thus not easily portable onto a mobile device.

3. Bayesian Belief Networks – Introduction

Bayesian Belief Networks are essentially models which reflect the states of some part of a world and describe how these states are related by probabilities. In simple words, BBNs let you model the probability of something happening given whether or not certain other related things happen.

A BBN defined in the form of a directed acyclic graph, where parts of a world (with corresponding states) are nodes in the graph, while relationships (commonly causal influences) are represented by directed edges.

Figure 1 is an example that is modelling an Alarm(A) system that can be caused to go off by either a Burglar(B) or an Earthquake(E). B and E have their own probabilities of occurring, and the directed edges from them to A mean that they can cause A to change state. Further, if the Alarm is set off, this causes either John or Mary to 'change state' and maybe call to see what is happening.

BBNs allow you to change the probability of the states in these nodes, and each change is treated as an event in the graph. BBNs then provide a mathematical way of measuring the effects of events on other nodes. Essentially this means that the prediction in one node is calculated precisely based on the predictions or states of the nodes neighbouring (and thus related to) it.

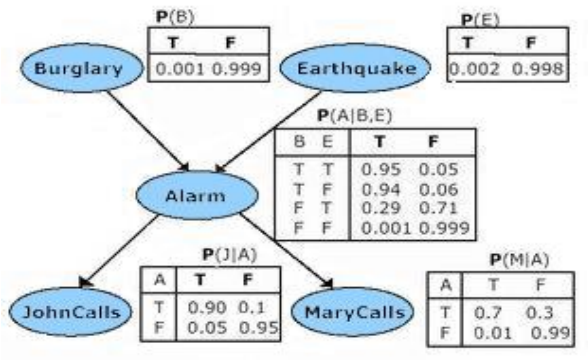

Figure 1 : A sample BBN with node probability tables[6]

The calculations are done by making use of node probability tables (NPTs), something that every node in a BBN must have, which define the probability distribution over the node's possible states. We can see this depicted in Figure 1, where the table for Alarm for example shows that if Burglary and Earthquake are both false, Alarm will have a probability of 0.001 of being true.

This lookup mechanism would appear simple and trivial to implement were it not for these important considerations:

- The first one is that you can also deal with unknowns. Being able to model the unknown is useful in any model, as that reflects many real world situations. In BBNs this is simply handled by using the NPT themselves, rather than waiting until real evidence is inserted. This means that if you do not know anything about the states of neither Earthquake nor Burglary you can still predict the probability of an Alarm by the following calculation[6]:

Table 1 : Calculating initial probabilities of a node

p(A) =	p(AEB) + p(A~EB) + p(AE~B) + p(A~E~B)
=	p(A | EB) * p(EB) + p(A | ~EB) * p(~EB) + p(A | E~B) * p(E~B) + p(A | ~E~B) * p(~E~B)
=	p(A | EB) * p(E) * p(B) + p(A | ~EB) * p(~E) * p(B) + p(A | E~B) * p(E) * p(~B) + p(A | ~E~B) * p(~E) * p(~B)
=	0.95 * 0.002 * 0.001 + 0.94 * 0.998 * 0.001 + 0.29 * 0.002 * 0.999 + 0.001 * 0.998 * 0.999
=	0.0025 (0.25%)

As you can see from the above case, only once we reach row 3 do all the probabilities simplify to ones that can be looked up directly in the NPTs.

- This method of calculation allows you to do another thing: it allows you to input observational data (called likelihoods in some BBN desktop applications[5]) into nodes. This means that you can say directly that there is a 20% chance of an Earthquake (and 80% of no quake). This form of uncertainty is also highly realistic, and hence potentially important in any prediction mechanism. However the fact that these 2 types of uncertainties can be modelled by a BBN means calculating probabilities is slightly trickier than just a table look up.

- The last and most important consideration is that BBN's can calculate backwards. This means that given an observation on a node they can work back to update the node's parents. To go back to our sample net, we can say, for example, that if we know that an alarm has happened, we also can guess that the probabilities of Burglar and Earthquake will get pushed up. This is intuitive to human beings (One might say: "Well, if the alarm went off, it must have been either an earthquake or a burglar, so they are now much more probable…"), but is hard for machines to model.

The way this is done explains the name of Bayesian Belief Networks, because it makes use of Bayes' Theorem :

$$P(X \mid Y) = \frac{P(X) \bullet P(Y \mid X)}{P(Y)}$$

So if we know that A is true (this is called entering a finding about A into the BBN), we can backwards calculate the probabilities of E or B being true, by using Bayes Theorem like so:

Table 2 : Posterior belief calculations

P(B | A) =	(p(A | B) * p(B)) / p(A)
=	((p(A | EB) * p(E) + p(A | ~EB) * p(~E)) * p(B)) / p(A)
=	((0.95 * 0.002 + 0.94 * 0.998) * 0.001) / 0.0025
=	0.376 (37.6 %)

P(E | A) =	(p(A | E) * p(E)) / p(A)
=	((p(A | EB) * p(B) + p(A | E~B) * p(~B)) * p(E)) / p(A)
=	((0.95 * 0.001 + 0.29 * 0.999) * 0.002) / 0.0025
=	0.233(23.3 %)

Note that as soon as a finding gets entered into A any sort of evidence entered into B or E in the future will not affect it anymore, unless we retract the finding from A. This essentially stops the BBN from changing any evidence we are sure about.

A side-effect of this is that any evidence input into either B or E will not affect any other node in the network, which it would have had A not been known. This concept in a BBN is called d-separation. The name comes from the fact that B and E are now both separated from any node in the network, and from each other.

This sort of posterior updating is a major strength as well complication of this technology. The complication becomes clear in Figure 2. If the a finding is entered node b, the fact that updating goes both up and down links means that at node d, for example, conflicting evidence might be coming into the node from its two neighbours. There is also a remote danger of creating potentially infinite cycles in much larger networks. (In this case there would be no infinite cycle since **b** is known, and hence would not be updated by arrows 7 and 8.) In any case, updating such a BBN in a consistent way is a challenge at best.

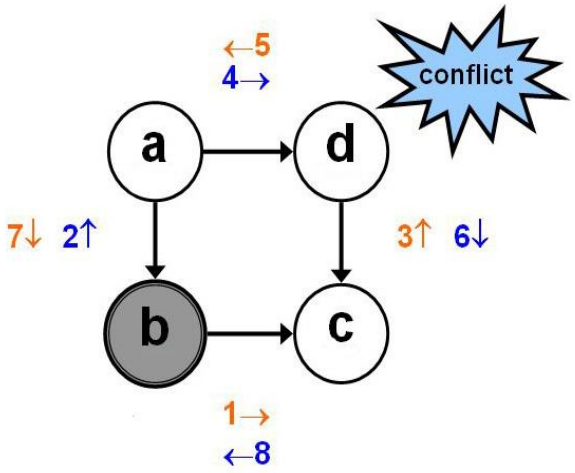

Figure 2 : Potential conflicts during propagation

In the next section, I will outline the series of steps over the network necessary to produce a clean, efficient, and fast solution to the propagation problem.

These details of how a Bayesian Belief Network is propagated and initialised are also available in the 'BBN' section on the CD which accompanies this report. That section also contains real examples that can be run in the desktop tool Netica. I would encourage you to read this and some of the papers and resources listed there in order to understand BBNs better.

4. Junction Tree Algorithm

This section provides insight into the complicated algorithm that lets us propagate a Bayesian Belief Network efficiently. In order to do this we must first construct what is called a junction tree (sometimes also called a cluster tree) over the network.

The junction tree is an intricate data structure that contains complex nodes called cliques. Each clique may be thought of as a subset of the nodes in the BBN. It also holds a table constructed by multiplying the NPTs of the nodes it contains. Further, cliques are linked together by undirectional labelled edges. Each such edge contains the intersection between the subsets of nodes that the cliques on either end of the edge contain. These edges are called separators. Why and how this sort of structure allows us to propagate Bayesian Belief networks is covered in the next few sections.

4.1 Why use a Junction Tree?

The propagation algorithm over a junction tree works by passing and receiving messages between its (clique) nodes utilizing the separators. Each node has to collect evidence from its neighbours only once, and once it has done that it sends out its own evidence when its neighbours request it. The end result of this is that the network gets propagated and consistent in a definitely-finite number of steps and no conflicts. This means that this sort of message passing algorithm is very efficient compared to the up-and-down updating mentioned in Section 3.

Further, once a junction tree has been successfully set up and propagated, we no longer require complex probability calculations to determine the probability of a node after an event, but only a simple summation over a clique. The only time we need the calculations is when we set up the tree. More on this in sections 4.3 and 4.5.

4.2 Constructing a Junction Tree

Constructing a junction tree is done by following a series of steps to change the graph of the original network into a tree. I'd recommend doing this over a clone of the network, just to be on the safe side. It is also much more straightforward to let a user interact with what he thinks is the original BBN structure and hide the junction tree computations from him/her. That way the user does not need any knowledge about junction trees.

The data structure I used for representing my junction tree is a combination of clique nodes and edges with separators. Each clique node has references to its neighbouring nodes, as well as references to those edges which lead to it. The edge contains references to the two nodes it links, as well as to the separator object it contains.

I used this sort of mutually referential structure because at some points I needed quick access to neighbours of nodes to pass messages to them, while using separators from those edges I wanted to pass messages over. The above structure allowed me to do this easily.

The next few paragraphs describe the steps needed to transform a BBN into my junction tree in some detail. The actual algorithms I used to do this are on the CD.

Step 1 : Moralization

To make a graph moral, one has to take every node of the BBN in turn and add links between its parents (if they are not linked already). If we have a network made from a set of vertices ***V*** we can express this more formally like so:

> For all **w** ∈ ***V***: For all **u,v** ∈ ***parents*(w)**
> make sure there is an edge **e=u-v**
> where **u** ≠ **v**

To complete this step we remove directionality from all edges. Figure 3 gives an example of this over a sample network.

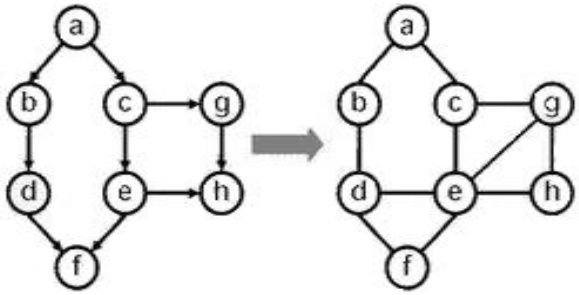

Figure 3 : Before and after the moralization step. The edges d-e and e-g have been added.

Step 2 : Triangulation

The next step involves making the graph triangulated. Note that we are now working directly on the moralized graph, not on the original DAG. Triangulation means that we have to make sure that whenever there is a cycle of length mote than 3 in our moral graph, that cycle has a chord. So essentially we keep adding edges to the moral graph until there is no cycle with length ≥ 4 without a chord.

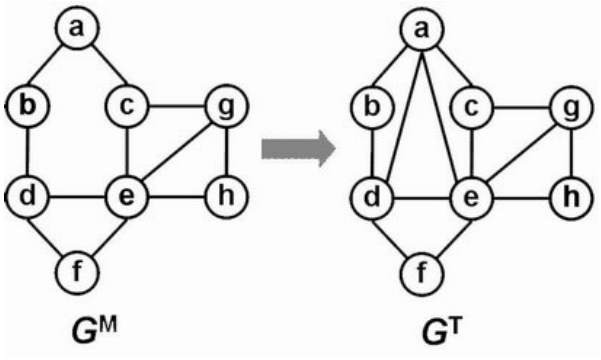

Figure 4 : Triangulating the moral graph G^m to achieve the (minimal) triangulated graph G^t.

An interesting problem that arises from the implementation of the triangulation algorithm is that there are many different ways of how to triangulate a graph. The triangulation above produced a minimal triangulation, meaning that only the smallest number of edges (chords) possible was added. The other extreme would be a fully connected graph.

In fact, the fewer edges you add at this stage, the easier and quicker it will be to use the resulting junction tree to propagate the network. That is because of the next step, where the new edges are used to create cliques. If many edges are added, this will produce large, unwieldy cliques. This should become apparent presently.

Step 3 : Clique Building

As mentioned before, cliques are the complex nodes that make up the junction tree. They are constructed by extracting complete (fully connected) subgraphs from G^t, the triangulated graph from the previous step. Intuitively, the triangulation done in the last step should have produced many such subgraphs, since it should have reduced all cycles of length >3 to cycles of length 3. These 3-edged cycles are actually complete subgraphs, and hence can easily be used to build cliques if taken out of the graph carefully. Once again there are different ways how to extract cliques from G^t, depending, for example, on where you start, and so a number of possible clique sets is possible. One example of the cliques that can be extracted from G^t is shown in Figure 5.

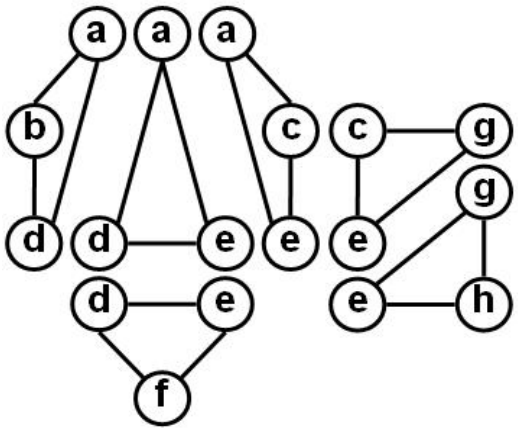

Figure 5 : Possible cliques that can be selected from the previous triangulated graph.

Step 4 : Constructing a junction graph

From the above collection of cliques, we can now build a junction graph. A junction graph is built by taking each clique and turning it into a node. Edges are then added between any cliques that have an intersection over the set of nodes they contain. This intersection is called a separator. Each such edge is 'labelled' with that separator. At this stage we only one step away from creating our tree data structure.

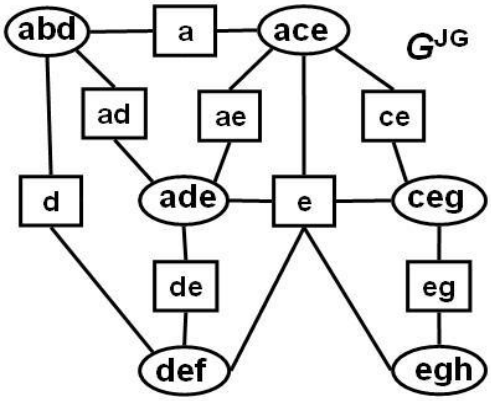

Figure 6 : A junction graph over the selected cliques.

Step 5 : Pruning the junction graph into a tree

The final step in the creation of the junction tree is to remove all redundant edges (separators) from the graph. Once this is done we should be left with a tree.

A redundant edge in this case is one whose separator is contained within more complex separators that leave from the redundant edge's starting point and arrive at its ending point. In other words, if you have an edge with a separator that is already contained in others, and you can use the others to act as a path to simulate your first edge, you can remove that edge. For example, take the edges :

a) (ABD – D – DEF)
b) (ABD – AD – ADE)
c) (ADE – DE – DEF)

We can see that the separator **D** of edge **a** is contained within the separators of the other 2. Further, we can see that if we follow **D** from edge **b** to edge **c** we still leave and arrive at the same destinations as the two ends of edge **a**. This means we can eliminate **a**.

So after removing all the redundant edges, we should now be left with:

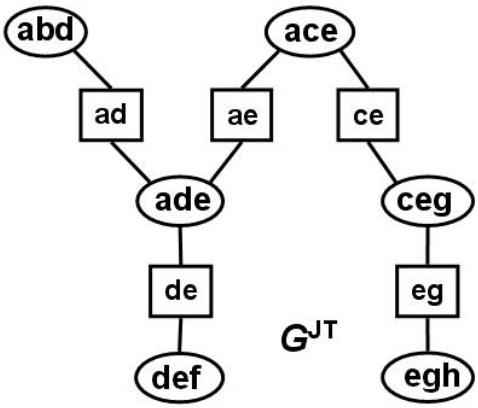

Figure 7 : The junction tree of our original network.

4.3 Initializing a Junction Tree

After having constructed a junction tree in the last section, we now have to initialize it by populating it with correct data from the original network so that we can use it to propagate. This means building up tables of probabilities called potentials for every clique and separator. A potential of a clique is created by multiplying the NPTs of the nodes contained in each clique. Initializing the tables in the separators is done by setting their values to 1. This step is done only once.

Before one can do any multiplication of tables, we have to determine which tables go where. This is because we can see that a node may be repeated in many cliques, but we only want to incorporate its NPT in exactly one clique. In the case of nodes with no dependencies (no parents) the choice of clique does not matter, but a node which has parents should be contained in a clique which also contains its parents. This is intuitive in that the node's NPT is dependent on

the node's parents, and if we are going to be multiplying them, they should be together.

This means that a decision has to be made such that the real NPT of a node gets assigned exactly to one clique. In the other cliques where the node is present but not assigned to the values in NPT are set to one. The initial propagation of the network will populate even these NPTs with the correct values, but this will be discussed later. The actual multiplication of the tables is done as follows:

You have to start out by making a table which is indexed by all the possible combinations of node states, based upon our original nodes contained in each clique. As an example I shall use the Burglar, Earthquake and Alarm nodes (and their NPTs) from Figure 1. I will assume that they would make up a clique, to and their 3 NPTs would all be assigned to it. Next, for every possible combination of states, you have to multiply the values in the NPTs corresponding to those states. This is difficult to grasp without an example, so please look at Figure 8 below for the details.

B	E	A	Result
T	T	T	P(B|E,A) * P(E|B,A) * P(A|B,E)
T	T	F	P(B|E,¬A) * P(E|B,¬A) * P(¬A|B,E)
T	F	T	P(B|¬E,A) * P(¬E|B,A) * P(A|B,¬E)
T	F	F	P(B|¬E,¬A) * P(¬E|B,¬A) * P(¬A|B,¬E)
F	T	T	P(¬B|E,A) * P(E|¬B,A) * P(A|¬B,E)
F	T	F	P(¬B|E,¬A) * P(E|¬B,¬A) * P(¬A|¬B,E)
F	F	T	P(¬B|¬E,A) * P(¬E|¬B,A) * P(A|¬B,¬E)
F	F	F	P(¬B|¬E,¬A) * P(¬E|¬B,¬A) * P(¬A|¬B,¬E)

Figure 8 : Initializing a potential

B	E	A	Result
T	T	T	P(B) * P(E) * P(A|B,E)
T	T	F	P(B) * P(E) * P(¬A|B,E)
T	F	T	P(B) * P(¬E) * P(A|B,¬E)
T	F	F	P(B) * P(¬E) * P(¬A|B,¬E)
F	T	T	P(¬B) * P(E) * P(A|¬B,E)
F	T	F	P(¬B) * P(E) * P(¬A|¬B,E)
F	F	T	P(¬B) * P(¬E) * P(A|¬B,¬E)
F	F	F	P(¬B) * P(¬E) * P(¬A|¬B,¬E)

Figure 9 : Simplifying the potential calculations

At this stage one can notice from our original network that Burglar is independent from Earthquake, and vice versa, since they do not share an edge. Also, they should be independent from Alarm as well, because of the directionality of the link. (At this stage one has to ignore the fact that in reality BBNs can propagate back up, and only deal with the graph topography as it is specified.) This means that we can reduce the table in Figure 8 to the one in Figure 9.

One can see that the probabilities being multiplied in the table in Figure 9 map directly into the node probability tables of the nodes of original network, and a simple lookup there will yield the numbers needed.

Now that we have set up the potentials, we can go on to the next step.

4.4 Propagating

Propagation happens at two types of time over the junction tree: when the tree is first initialized and then whenever inputs are entered into the network. It needs to be done to allow all nodes to be consistent with each other. Nodes are not consistent with each other initially because during initialization only one clique gets the node's real NPT. Later, if a new probability is input into a propagated network the NPT of the targeted node gets changed, so we have to propagate once more.

As I mentioned before, to make a network consistent, every node must collect evidence from all of its neighbours, and then be prepared to send out its own evidence when its neighbours request it in turn. This can be achieved by having two recursive methods, one called CollectEvidence and the other called DistributeEvidence which can be called on an arbitrary clique in the network to make it consistent. An example is given in Figure 10. The actual algorithms for this are on the companion CD.

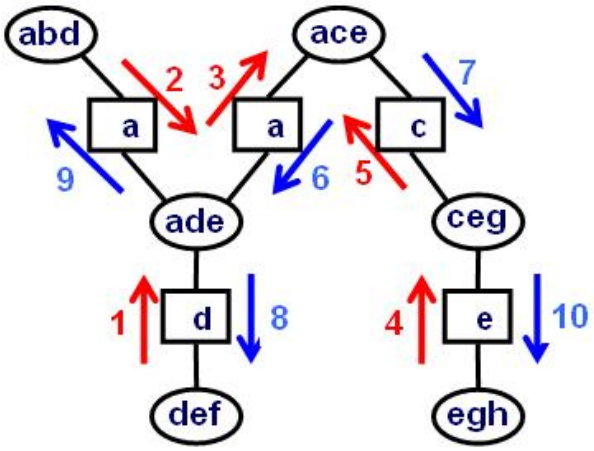

Figure 10 : Message passing over a junction tree.
Source node: 'ace' **red:** collect **blue:** distribute

Each one of the labelled arrows in Figure 10 represents a message being passed from a clique to another clique via a separator. While so far I have said that these messages do happen, I have not mentioned what it is that they actually do. Each message actually has 2 parts: the absorption part and the projection part.

Absorption:

Absorption means that a separator between two nodes absorbs evidence from one node. If we have a separator S between nodes A and B, then formally absorption from A can be defined so:

$$\phi_{S_{AB}}^{old} \leftarrow \phi S_{AB}$$
$$\phi S_{AB} \leftarrow \sum_{A \setminus S_{AB}} \phi A$$

Informally what it means is that for every combination of states in the separator's potential, one does a lookup in A's potential and adds the matching values found there together, putting the result back into the separator's potential.

Projection :

After absorbing evidence, the separator then projects it into B. This changes B's potential, and may be formally defined so:

$$\phi B \leftarrow \phi_B^{old} \bullet \frac{\phi S_{AB}}{\phi_{S_{AB}}^{old}}$$

Informally, this means that you divide the new separator potential by its old one, and then multiply the combinations found in your result with any corresponding ones in B's potential.

An example of a message being passed is depicted in Figure 11. After the CollectEvidence and DistributeEvidence methods have been called and passed all the messages they had to, the junction tree is now propagated and consistent. To see how to extract information from the junction tree back into a BBN, please see section 4.6

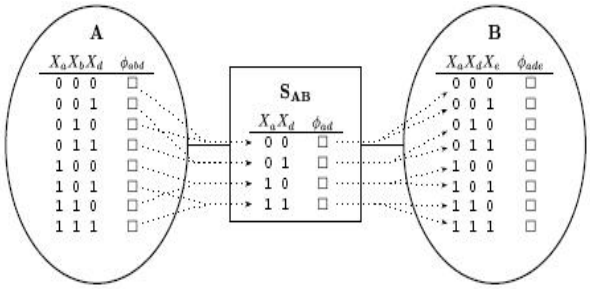

Figure 11 : Passing a message

4.5 Entering Evidence into a Junction Tree

As described in Section 3, one may either enter a finding or what some BBN tools referrer to as a 'likelihood'[5] (observational data) into a node.

Finding :

A finding on a node means that one of the states of the node is observed to be true, and hence gets set to 1 while all the other ones get set to 0. To enter a finding on a node into the junction tree, we must first find a clique which contains that node. To update it with the new evidence, every row in the clique's potential that does not contain the node's observed state is multiplied by 0. The potential of the clique must then be normalized to make it add up to one again.

Observational Data :

This sort of finding for a node means that there is a 'likelihood' associated with every state of the node. For example, in a 2-state node the first state might have a likelihood of 20% and the second naturally of 80%. To enter a likelihood for a node into the junction tree, we once again have to find a clique which contains that node. To update it with the new evidence, every row in the clique's potential must be multiplied by the likelihood of the node's state in it. So if the first row contains the node's first state, its value must get multiplied by 0.2 and so on. The potential of the clique must then be normalized to make it add up to one again.

In both cases the network has to be propagated as soon as the new evidence is entered.

4.6 Back to the Network

This section describes how to extract information from the Junction tree back into the probabilities of the nodes of our original network. And after all the complicated algorithms painfully described previously, one can finally see the end result.

The only thing you need to do to calculate the probability of a state of a node is to find any clique in the junction tree which contains that node, and sum over its NPT adding up just those values where your desired state holds. So, relating this to the potential we built in Figure 9, if we want the probability of an Alarm going off, we sum over all the state combinations where alarm is true:

B	E	A	Value
T	T	T	a
T	T	F	b
T	F	T	c
T	F	F	d
F	T	T	e
F	T	F	f
F	F	T	g
F	F	F	h

$$P(A=T) = \sum (a\,c\,e\,g)$$

Figure 12 : Summing over a potential to get final result

This type of summation is quite simple, and provides an elegant ending to an otherwise quite complicated algorithm.

5. Conclusion

I believe that looking at the use of intelligent prediction techniques on Java enabled devices is a potentially useful contribution in the field. These devices are proliferating very quickly, and their lack of adequate input techniques means that any smart automation of tasks that can be brought about by an intelligent application should be pursued.

By building my BBN Propagation Engine in Java, I believe that I have created an efficient, portable, and small piece of software that could actually be used to harness such intelligence on these devices.

On a more personal level, I have gained significant experience in the field of Bayesian Belief Networks and probability and graph theory in general, which will be useful in my future studies or career. In the way of practical skills I also got large amounts of experience in solo-managing a large software project, right down from the design and the parsing, to the large amounts of Java coding that was involved.

6. Acknowledgements

[1] Dr. Andrew Runnalls, for his explanations and assistance during our project meetings
[2] Henrik Bengtsson, for allowing me to use his BBN diagrams, many of which appear in this report.
[3] David McGaw of Motorola, for helping me figure out BBN basics

7. References/Bibliography

[1] Finn V. Jensen, *An Introduction to Bayesian Networks*, UCL Press, London, 1996
[2] R.G. Cowell, A.P. Dawid, S.L. Lauritzen, D.J. Spiegelhalter, *Probabilistic Networks and Expert Systems*, Springer-Verag, New York, 1999
[3] Henrik Gengtsson, "Bayesian networks - a self-contained introduction with implementation remarks", Masters Thesis and Presentation, Lund Institute of Technology, 1999
[4] "Introduction to Bayes Nets", www.norsys.com/tutorials/netica/nt_toc_A.htm
[5] www.murrayc.com/learning/AI/bbn.shtml
[6] Mark Paskin, "A Short Course on Graphical Models", http://www.cs.berkeley.edu/~paskin/gm-short-course/
[7] www.cs.pitt.edu/~milos/courses/cs2001/cs2001-1.pdf
[8] Jean Blair, Pinar Heggernes, and Jan Arne Telle , "A Practical Algorithm for Making Filled Graphs Minimal", Theoretical Computer Science 250-1/2 (2001), pages 125-141.
[9]"A Comparison of Lauritzen-Spiegelhalter, Hugin, and Shenoy-Shafer Architectures for Computing Marginals of Probability Distributions" in G. Cooper and S. Moral (eds.), Uncertainty in Artificial Intelligence, Vol. 14, 1998, pp. 328--337, Morgan Kaufmann, San Francisco