

Student Work

8-1-2000

Inference propagation engine for belief networks.

Alexander Churbanov

Follow this and additional works at: <https://digitalcommons.unomaha.edu/studentwork>

Recommended Citation

Churbanov, Alexander, "Inference propagation engine for belief networks." (2000). *Student Work*. 3554.
<https://digitalcommons.unomaha.edu/studentwork/3554>

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.



Inference propagation engine for belief networks

by

Alexander E. Churbanov

A Thesis

Presented to the
Department of Computer Science
and the Faculty of the Graduate College at the
University of Nebraska
in Partial Fulfillment of the Requirements
for the Degree of Master of Science

Major: Computer Science

Under the Supervision of Dr. Alexander D. Stoyen

Omaha, Nebraska

August, 2000

UMI Number: EP74752

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI EP74752

Published by ProQuest LLC (2015). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code

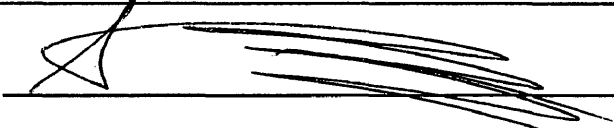


ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

THESIS ACCEPTANCE

Acceptance for the faculty of the Graduate College,
University of Nebraska, in partial fulfillment of the
requirements for the degree Master of Science,
University of Nebraska at Omaha

Committee

Chairperson  _____

Date 7-21-2000

ABSTRACT

Inference propagation engine for belief networks

Alexander E. Churbanov, MS, University of Nebraska, 2000

Advisor: Dr. Alexander D. Stoyen

Due to significant limitations of rule-based extensional decision-support systems researchers are looking for new theories, methods and semantics to efficiently encode causality. Artificial Intelligence community demonstrates significant interest for the approaches based on theory of probability. Graphical model approach offers significant benefits and leans on sound theoretical basement. Paper discusses benefits of Intentional (*declarative* or *model based*) vs. Extensional (*rule-based* or *production rules*) approaches. Probability Propagation in Trees of Clusters (PPTC) algorithm is one of the most efficient algorithm inspired by generalized distributive law. Paper focuses on details of this recently adapted algorithm. Applet written in Java culminates the research. Algorithm implementation as an applet opens new horizons of system use, since Java™ is supported by nearly all platforms nowadays. Optimized inference propagation algorithm was devised based on high-level description of algorithm, making the applet highly fit for real life applications. Object oriented language implementation is beneficial over other approaches, since it makes package reuse simple and handy. Graphical user interface was designed with idea of maximal ease of the applet use. Program can run both in applet mode and as a standalone application. Autonomous on-board computers,

enabling intelligence for mobile devices, could run the inference propagation engine which is essential part of newly created software.

Keywords: Artificial intelligence, Bayesian network, belief network, causal network, evidence, expert system, join tree, probabilistic inference, probability propagation, reasoning under uncertainty.

Contents

1 INTRODUCTION.....	1
2 EXTENSIONAL SYSTEMS: MERITS, DEFICIENCIES AND REMEDIES.....	5
2.1 COMPUTATIONAL MERITS	5
2.2 SEMANTIC DEFICIENCIES.....	8
2.2.1 <i>The role of bi-directional inferences</i>	8
2.2.2 <i>The limits of modularity</i>	9
2.2.3 <i>Correlated evidence</i>	12
3 INTENTIONAL SYSTEMS AND NETWORK REPRESENTATIONS.....	14
3.1 WHY NETWORKS.....	14
3.2 GRAPHOID AND FORMALIZATION OF RELEVANCE AND CASUALTY	15
3.3 THE PRIMITIVE RELATIONSHIPS OF PROBABILITY LANGUAGE.....	17
3.3.1 <i>Likelihood</i>	17
3.3.2 <i>Conditioning</i>	18
3.3.4 <i>Relevance</i>	19
3.3.5 <i>Causation</i>	20
4 BAYESIAN NETWORKS	22
4.1 BASIC AXIOMS	22
4.2 CONDITIONAL PROBABILITIES	22
4.3 LIKELIHOOD.....	24
4.4 SUBJECTIVE PROBABILITIES	24
4.5 PROBABILITY CALCULUS FOR VARIABLES	25
4.6 CONDITIONAL INDEPENDENCE	27
5 PROBLEM DEFINITION.....	29
5.1 DEFINITIONS	29
5.1.1 <i>Variables And Values</i>	30
5.1.2 <i>Potentials and distributions</i>	31
5.1.3 <i>Belief network</i>	33
5.1.4 <i>The secondary structure</i>	34
5.2.1 <i>Constructing The Moral Graph</i>	37
5.2.2 <i>Triangulating the Moral Graph</i>	38
5.2.3 <i>Identifying cliques</i>	41
5.2.4 <i>Building an optimal join tree</i>	41
5.2.5 <i>Building an Optimal Join Tree</i>	42
5.2.6 <i>Choosing The Appropriate Sepsets</i>	43
5.3 FURTHER STEPS IN MAKING INFERENCE.....	44
5.3.1 <i>Initialization</i>	45
5.3.2 <i>Global propagation</i>	46
5.3.3 <i>Marginalization</i>	50

5.4 HANDLING EVIDENCE.....	52
5.4.1 <i>Observations and likelihoods</i>	52
5.4.2 <i>PPTC Inference With Observations</i>	53
5.4.3 <i>Initialization with observations</i>	55
5.4.4 <i>Observation entry</i>	55
5.4.5 <i>Normalization</i>	56
5.5 HANDLING DYNAMIC OBSERVATIONS	57
5.6 IMPLEMENTATION NOTES	61
6 EXISTING SOFTWARE AND RELATED WORK.....	62
7 OUR SOLUTION APPROACH.....	69
8 DELIVERABLES	71
APPENDIX A	78
APPENDIX B	81
APPENDIX C	83
LITERATURE	84

List of figures

2.1	Example of certainty combination rules in extensional systems	6
2.2	Making the antecedent of a rule more credible can cause the consequent to become less credible	11
2.3	The Chernobyl disaster shows why rules cannot combine locally	13
5.1	Sample belief network	30
5.2	Secondary structure for sample belief network	31
5.3	Moral graph.....	39
5.4	Triangulating the moral graph	40
5.5	Making further inference after graphical transformation.....	45
5.6	Initialization of cluster ACE and sepset CE	47
5.7	Message passing during global propagation.....	50
5.8	Marginalization example	51
5.9	Block diagram of PPTC with observations.....	54
5.10	Block diagram of PPTC with dynamic observations.....	59
6.1	Sample edit interface form of Hugin Lite 5.3 system.....	62
6.2	Run mode for the sample wet grass problem with evidential support of raining	63
6.3	Sample wet grass problem in JavaBayes editor frame.....	66
6.4	JavaBayes console frame showing posterior probability of sprinkler discrete variable given the evidential support of rain	66
6.5	Bayes Net applet with wet grass example problem	68
7.1	Structure of Bayesian network applet	74

7.2	Frame open form.....	75
7.3	Inference propagation engine frame	76
7.4	Discrete variable editing	76
7.5	Discrete function editing.....	77

List of tables

4.1	An example of $P(A B)$. The columns sum to one.....	26
4.2	An example of $P(A, B)$. The sum of all entries is one.....	26
4.3	An example of $P(B A)$ as a result of applying Bayes' rule to Table 4.1 and $P(B) = (0.4 \quad 0.4 \quad 0.2)$	26
5.1	Likelihood encoding of $C = on, E = off$	54

1 Introduction

Artificial Intelligence community demonstrates significant growth of interest for Bayesian belief networks. Bayesian networks have been used as a fundamental tool for the representation and manipulation of beliefs in artificial intelligence. The networks allow implementing decision support systems with many important properties, such as encoding of uncertainty in the knowledge domain and utilization of the whole pool of information available to support a decision. The main benefit of Bayesian networks based systems is that they are flexible and could be easily trainable, expressing the information available in knowledge domain. We also can efficiently represent cause-effect relationships using Bayesian network approach.

Bayesian networks rely on inference algorithms to compute beliefs in the context of observed evidence. Network itself is a Directed Acyclic Graph (DAG). The heart of the system is inference propagation engine – generic algorithm for belief propagation in Bayesian networks. Propagating inferences through network, this engine assigns certain probabilities to discrete variables in the nodes.

The paper investigates the applicability of probabilistic methods to tasks requiring automated reasoning under uncertainty. The result is a computation-minded interpretation of probability theory, an interpretation that exposes the qualitative nature of this centuries-old formalism, its solid epistemological foundation, its compatibility with human intuition and, most importantly, its amenability to network representations and to parallel and distributed computation.

As part of this research work, the probability propagation algorithm was implemented. The algorithm closely resembles the generalized distributive law approach [4], widely used in practical algorithms (such as Fast Fourier Transforms (FFT)), to cut the computational burden. Implementation of algorithm in Java programming language offers significant benefit over other approaches, since Java is designed to be maximally portable to other platforms. This approach is quite new, since the older implementations in other programming languages tend to be focused on particular platform.

Application areas of the Bayesian reasoning include diagnosis, forecasting, image understanding, multi-sensor fusion, decision support systems, plan recognition, planning and control, speech recognition – in short, almost any task requiring that conclusions be drawn from uncertain clues and incomplete information.

Reasoning about any realistic domain always requires that some simplifications be made. The very act of preparing knowledge to support reasoning requires that we leave many facts unknown, unsaid, or crudely summarized. For example, if we choose to encode knowledge and behavior in rules such as “Birds fly” or “Smoke suggests fire”, the rules will have many exceptions, which we cannot afford to enumerate, and the conditions under which the rules apply (e.g., seeing a bird or smelling smoke) are usually ambiguously defined or difficult to satisfy precisely in real life.

An alternative to extremes of ignoring or enumerating exceptions is to summarize them, i.e., provide some warning signs to indicate higher odds of phenomenon in one place in comparison with others. Summarization is essential if we wish to find a reasonable compromise between safety and speed of movement. This paper studies a

language in which summaries of exceptions in the minefield of judgment and belief can be represented and processed.

One way to summarize exceptions is to assign to each proposition a numerical measure of uncertainty and then combine these measures according to uniform syntactic principles, the way truth values combined in logic. This approach has been adopted by first-generation expert systems, but it often yields unpredictable and counterintuitive results.

A more fundamental taxonomy can be drawn along the dimensions of *extensional* vs. *intentional* approaches. The extensional approach, also known as production system, rule-based systems, and procedure-based systems, treats uncertainty as a generalized truth value attached to formulas and (following the tradition of classical logic) computes the uncertainty of any formula as a function of the uncertainties of its subformulas. In the intentional approach, also known as declarative or model-based, uncertainty is attached to “states of affairs” or subsets of “possible worlds”.

Extensional systems treat uncertainty as a generalized truth value; that is, the certainty of a formula is defined to be unique function of the certainties of its subformulas. Thus, the connectives in the formula serve to select the appropriate weight-combining function. For example, the certainty of the conjunction $A \wedge B$ is given to be some function (e.g. minimum of the product) of the certainty measures assigned to A and B individually. By contrast, in intentional systems, a typical representative of which is probability theory, certainty measures are assigned to sets of worlds, and the connectives combine sets of worlds by set theory operations. For example, the probability $P(A \wedge B)$ is given by the weight assigned to the intersection of two sets of worlds – those in which A

is true and those in which B is true – but $P(A \wedge B)$ cannot be determined from individual probabilities $P(A)$ or $P(B)$.

The rules in extensional systems provide licenses for certain symbolic activities. For example, a rule $A \xrightarrow{m} B$ may mean “If you see A, then you are given a license to update the certainty of B by certain amount which is a function of the rule strength m ”. The rules are interpreted as a summary of past performance of the problem solver, describing the way an agent normally reacts to problem situations or to items of evidence.

In the Bayesian formalism the rule $A \xrightarrow{m} B$ is interpreted as a conditional probability expression $P(A | B) = m$, stating that among all worlds satisfying A, those that also satisfy B constitute m percent majority.

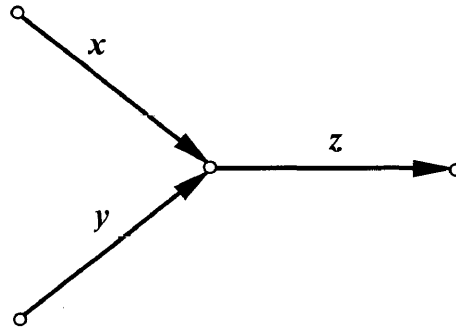
2 Extensional systems: merits, deficiencies and remedies

2.1 Computational merits

A good way to show the computational merits of extensional systems is to examine the way rules are handled in the certainty-factors formalism and contrast it with probability's theory's treatment of rules. Example on Fig.1 depicts combinational function that applies to serial and parallel rules, from which one can form a *rule network*. The result is a modular procedure for determining the certainty of a conclusion, given the credibility of each rule and the certainty of the premises. (i.e. roots of the network).

Rules:

- **If A then C(x)**
- **If B then C(y)**
- **If C then D(z)**

**1. Parallel computation**

$$F(C) = \begin{cases} x + y - xy & x, y > 0 \\ (x + y) / (1 - \min(x, y)) & x, y \text{ different sign} \\ x + y + xy & x, y < 0 \end{cases}$$

2. Series computation

$$F(D) = z \cdot \max(0, F(C))$$

3. Conjunction, negation ...

Figure 1. Example of certainty combination rules in extensional systems.

(x, y and z denote credibilities of the rules)

For example, the logical rule “If A then B” has the following procedural implementation: “If you see A anywhere in the knowledge base, then regardless of what other things the knowledge base contains and regardless of how A was derived, you are given the license to assert B and add it to the database.” This combination of *locality* (“regardless of other things”) and detachment (“regardless of how it was derived”) constitutes the principle of *modularity*.

The numerical parameters that decorate the combination functions in fig.1 do not alter this basic principle. The procedural license provided by the rule $A \xrightarrow{x} B$ reads as follows: “If you see the certainty of A undergoing a change δ_A , then regardless of what other things the knowledge base contains and regardless of how δ_A was triggered, you are given an unqualified license to modify the current certainty of B by some amount δ_B , which may depend on x , on δ_A , and on the current certainty of B.”

To appreciate the power of this interpretation, let us compare it with that given by an intentional formalism such as probability theory.

Interpreting rules as conditional probability statements, $P(B | A) = p$, does not give us license to do anything. Even if we are fortunate enough to find A true in the database, we still cannot assert a thing about B or $P(B)$, because the meaning of the statement is “If A is true and A is the only thing that you know, then you can attach to B a probability p .” As soon as other facts K appear in database, the license to assert $P(B) = p$ is automatically revoked, and we need to look up $P(B | A, K)$ instead. The probability statement leaves us totally impotent, unable to initiate any computation, unless we can verify that everything else in the knowledge base is irrelevant. That is why verification of irrelevancy is so crucial in intentional systems.

We shall now describe the semantic penalties imposed when relevance considerations are ignored in extensional systems.

2.2 Semantic deficiencies

Extensional systems often yield updating that is incoherent, i.e. subject to surprises and counterintuitive conclusions. These problems surface in several ways, most notably

1. Improper handling of bi-directional inferences,
2. Difficulties in retracting conclusions, and
3. Improper handling of correlated sources of evidence.

We shall describe these problems in order.

2.2.1 *The role of bi-directional inferences*

The ability to use both predictive and diagnostic information is an important component of plausible reasoning, and improper handling of such information leads to rather strange results. A common pattern of normal discourse is that of *abductive* reasoning – if A implies B, then finding the truth value of B makes A more credible. Some researches call it *induction* pattern. This pattern involves both ways, from A to B and from B to A. Moreover, it appears that people do not require two separate rules for performing these inferences; the first rule (e.g. “Fire implies smoke”) provides the license to invoke the second (e.g. “Smoke makes fire more credible”). Extensional systems, on the other hand, require that the second rule be stated explicitly and, even worse, that the first rule be removed. Otherwise, a cycle would be created where any slight evidence in favor of A would be amplified via B and fed back to A, quickly turning into a stronger confirmation (of A and B), with no apparent factual justification. The prevailing practice

in such systems is to cut off cycles of this sort, permitting any diagnostic reasoning and no predictive inferences.

Removal of the predictive component prevents the system from exhibiting another important pattern of plausible reasoning, one that we call *explaining away*: If A implies B, C implies B, and B is true, then finding that C is true makes A less credible. In other words, finding a second explanation for an item of data makes the first explanation less credible. Such interaction among multiple causes appears in many applications. For example, finding that a bad muffler could have produced the smoke makes fire less credible.

To exhibit this sort of reasoning, the system must use bi-directed inferences: from evidence to hypothesis (or explanation) and from hypothesis to evidence. While it is sometimes possible to use brute force (e.g. enumerating all exceptions) to restore “explaining away” without the danger of circular reasoning, we shall see that any systems that succeeds in doing this must sacrifice the principles of modularity, i.e. locality and detachment. More precisely, every system that updates beliefs modularly at the natural rule level and that treats all rules equally is bound to defy prevailing patterns of plausible reasoning.

2.2.2 The limits of modularity

The principle of locality is fully realized in the reference rules of classical logic. The rule “If P then Q” means that if P is found true, we can assert Q with no further analysis, even if the database contains some other knowledge K. In plausible reasoning, however, the luxury of ignoring the rest of database cannot be maintained. For example,

suppose we have a rule $R_1 = \text{"If ground is wet, then assume it rained (with certainty } C_1\text{)."}$ Validating the truth of "The ground is wet" does not permit us to increase the certainty of "It rained" because the knowledge base might contain strange items such as $K = \text{"The sprinkler was on last night."}$ These strange items, called *defeaters* or *suppressors*, are sometimes easy to discover (as with $K' = \text{"The neighbor's grass is dry,"}$ which directly opposes "it rained"), but sometimes they hide cleverly behind syntactical innocence. The neutral fact $K = \text{"Sprinkler was on"}$ neither support nor opposes the possibility of rain, yet K manages to undercut the rule R_1 . This undercutting cannot be implemented in an extensional system; once R_1 is invoked, the increase in the certainty of "It rained" will never be retracted, because no rule would normally connect "Sprinkler was on" to "It rained". Imposing such a connection by proclaiming "Sprinkler was on" as an explicit exception to R_1 defeats the spirit of modularity by forcing a rule writer to pack together items of information that are only remotely related to each other, and it burdens the rules with an unmanageably large number of exceptions.

Violation of detachment can also be demonstrated in this example. In deductive logic, if K implies P and P implies Q , then finding K true permits us to deduce Q by simple chaining; a derived proposition (P) can trigger a rule ($P \rightarrow Q$) with the same vigor as a directly observed proposition can. Chaining does not apply in plausible reasoning. The system may contain two innocent-looking rules – "If the ground is wet then it rained" and "If the sprinkler was on then the ground is wet" - but if you find that the sprinkler was on, you obviously do not wish to conclude that it rained. On the contrary, finding that the sprinkler was on only takes away support from "It rained."

As another example, consider the relationships shown in fig. 2. Normally an alarm sound alerts us to the possibility of a burglary.

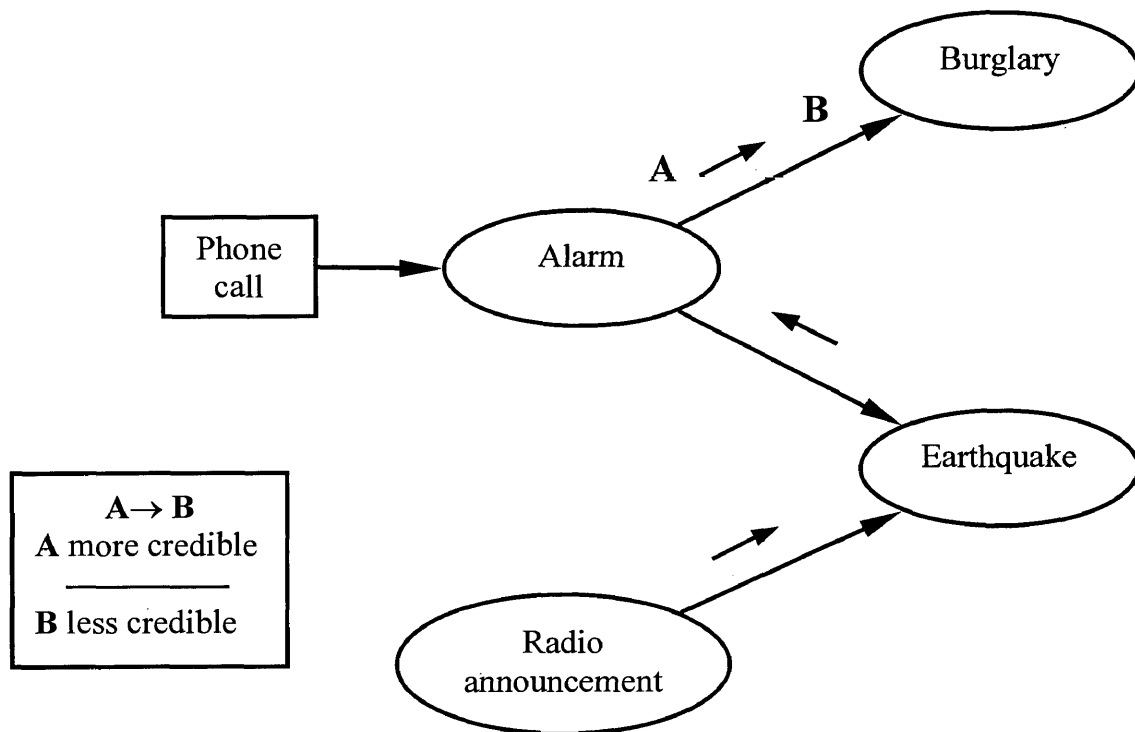


Figure 2. Making the antecedent of a rule more credible can cause the consequent to become less credible.

If somebody calls you at the office and tells you that your alarm went off, you will surely rush home in a hurry, even though there could be other causes for alarm sound. If you hear a radio announcement that there was an earthquake nearby, and if the last false alarm you recall was triggered by an earthquake, then your certainty of burglary will diminish. Again, this requires going both ways, from effect to cause (*Radio* → *Earthquake*), and from cause to effect (*Earthquake* → *Alarm*), and then from effect to cause again (*Alarm* → *Burglary*). Notice what pattern of reasoning results from such a chain, though: We have a rule, “If A (*Alarm*) then B (*Burglary*)”; you listen to the

radio, A becomes more credible, and the conclusion B becomes less credible. Overall, we have “If $A \rightarrow B$ and A becomes more credible, then B becomes less credible.” This behavior is clearly contrary to everything we expect from local belief updating.

In conclusion, we see that the difficulties plaguing classical logic do not stem from its nonnumeric, binary character. Equally troublesome difficulties emerge when truth and certainty are measured on a gray scale, whether by point values, by interval bounds, or by linguistic quantifiers such as “likely” and “credible.” There seems to be a basic struggle between procedural modularity and semantic coherence, independent of the notation used.

2.2.3 Correlated evidence

Extensional systems, greedily exploiting the licenses provided by locality and detachment, respond only to magnitudes of the weights and not to their origins. As a result they will produce the same conclusions whether the weights originate from identical or independent sources of information. An example about the Chernobyl disaster helps demonstrate the problem encountered by such local strategy. Fig. 3 shows how multiple, independent sources of evidence would normally increase the credibility of hypothesis (e.g., *Thousands dead*), but the discovery that these sources have a common origin should reduce the credibility.

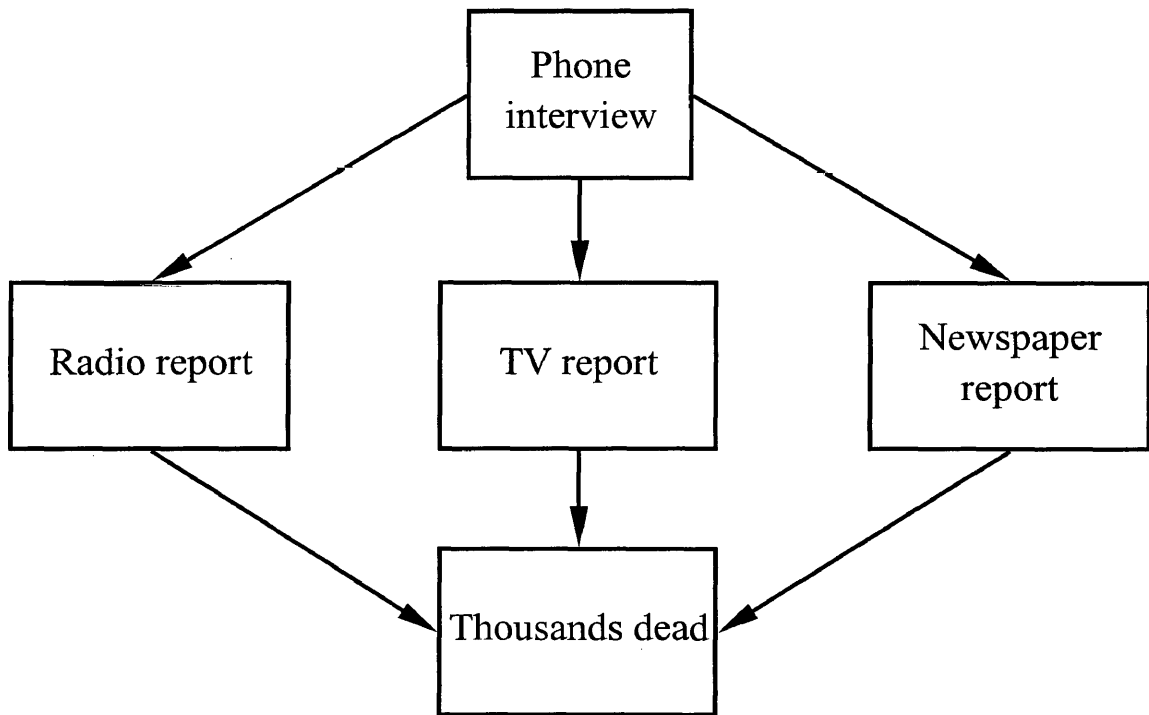


Figure 3. The Chernobyl disaster example shows why rules cannot combine locally

3 Intentional systems and network representations

In intentional systems, the syntax consists of declarative statements about states of affairs and hence mirrors world knowledge nicely. For example, conditional probability statements such as “Most birds fly” are both empirically testable and conceptually meaningful. Additionally, intentional systems have no problem handling bidirected inferences and correlated evidence; these emerge as built-in features of one globally coherent model. However, since the syntax does not point to any useful procedures, we need to construct special mechanisms that convert the declarative input into routines that answer queries. Such a mechanism is offered by techniques based on *belief networks*.

3.1 Why networks

Our goal is to make intentional systems operational by making relevance relationships explicit, thus curing the importance of declarative statements such as $P(B | A) = p$. As mentioned earlier, the reason one cannot act on the basis of such declarations is that one must first make sure that other items in the knowledge base are irrelevant to B and hence can be ignored. The trick, therefore, is to encode knowledge in such a way that the ignorable is recognizable, or better yet, that the unignorable is quickly identified and is readily accessible. Belief networks encode relevancies as neighboring nodes in a graph, thus ensuring that by consulting the neighborhood one gains a license to act; what you don't see locally doesn't matter. In effect, what network representations offer is a dynamically updated list of all currently valid licenses to ignore, and licenses to ignore constitute permissions to act.

Network representations are not foreign to AI systems. Most reasoning systems encode relevancies using intricate systems of pointers, i.e. networks of indices that group facts into structures, such as frames, scripts, causal chains, and inheritance hierarchies. These systems, though shunned by pure logicians, have proved to be indispensable in practice, because they place the information required to perform an inference task close to the propositions involved in the task. Indeed, many patterns of human reasoning can be explained only by people's tendency to follow the pathways laid out by such networks.

The special feature of the networks is that they have clear semantics. In other words they are not auxiliary devices contrived to make reasoning more efficient but are an integral part of the semantics of the knowledge base, and most of their features can even be derived from the knowledge base.

Belief networks play a central role in two uncertainty formalisms: probability theory, they are called Bayesian networks, causal nets, influence diagrams, qualitative Markov, or constraints networks.

3.2 Graphoid and formalization of relevance and casualty

A central requirement for managing intentional systems is to articulate the conditions under which one item of information is considered relevant to another, given what we already know and to encode knowledge in structures that display these conditions vividly as the knowledge undergoes changes. Different formalisms give rise to different definitions of relevance.

The essence of relevance can be identified with a structure common to all of these formalisms. It consists of four axioms, which convey the simple idea that when we learn

an irrelevant fact, the relevance relationships of all other propositions remain unaltered; any information that was irrelevant remains irrelevant, and that which was relevant remains relevant. Structures that conform these axioms are called *graphoids*. Interestingly, both undirected graph and directed acyclic graphs conform to the graphoids axiom (hence the name) if we associate the sentence “Variable X is irrelevant to variable Y once we know Z” with the graphical condition “Every path from X to Y is intercepted by the set of nodes corresponding to Z.”

With this perspective in mind, graphs, networks, and diagrams can be viewed as inference engines for efficient representing and manipulating relationships. The topology of the network is assembled from a list of local relevance statements (e.g., direct dependencies). This input list implies (using graphoid axiom) a host of additional statements, and the graph insures that a substantial portion of the latter can be verified by simple graphical procedures such as path tracing and path blocking. Such procedures enable one to determine, at any state of knowledge Z, what information is relevant to the task at hand and what can be ignored. Permission to ignore is the fuel that gives intentional systems power to act.

The probability theory is unique in its ability to process context-sensitive beliefs, and what the processing computationally feasible is that the information needed for specifying context dependencies can be represented by graphs and manipulated by local propagation. The primary appeal of probability theory is its ability to express useful qualitative relationships among beliefs and to process these relationships in a way that yields intuitively plausible conclusions. The fortunate match between human intuition

and the laws of proposition is not a coincidence. It came about because beliefs are formed not in a vacuum but rather as a distillation of sensory experience.

3.3 The primitive relationships of probability language

Although probabilities are expressed in numbers, the merit of probability calculus rests in providing a means for articulating and manipulating qualitative relationships that are found useful in normal discourse. The following four relationships are viewed as the basic primitives of the language:

1. Likelihood (“A is more likely than B”)
2. Conditioning (“If A then B”)
3. Relevance (“A depends on B”)
4. Causation (“A causes B”).

3.3.1 Likelihood

The qualitative relationship in the form “A is more likely than B” has traditionally been perceived as the prime purpose of using probabilities. The practical importance of determining whether one event is more likely than another is best represented by the fact that probability calculus was pioneered and developed by such ardent gamblers as Cardano (1501-1576) and De Moivre (1667-1754). However, the importance of likelihood relationships goes beyond gambling situations or even management decisions. Decisions depending on relative likelihood of events are important in every reasoning task because likelihood translates immediately to *processing time* - the time it takes to

verify the truth of a proposition, to consider the consequence of a rule, or to acquire more information. A reasoning system unguided by likelihood considerations would waste precious resources chasing the unlikely while neglecting the likely.

Philosophers and theorists have labored to obtain axiomatic basis for probability theory based solely on this primitive relationships of “more likely,” namely to identify conditions under which ordering of events has a numerical representation P that satisfies the properties of probability function.

3.3.2 Conditioning

Probability theory adopts the autoepistemic phrase “...given that I know is C ” as a primitive of the language. Syntactically, this is denoted by placing C behind the conditioning bar in a statement such as $P(A|C) = p$. This statement combines the notions of knowledge and belief by attributing to A degree of belief p , given the knowledge C . C is also called the context of the belief in A , and the notation $P(A|C)$ is called Bayesian conditionalism. Thomas Bayes (1702-1761) made his main contribution to the science of probability by associating English phrase “...given that I know C ” with the now-famous ratio formula

$$P(A|C) = \frac{P(A,C)}{P(C)}$$

which has become a definition of conditional probabilities.

It is by virtue of Bayes conditionalization that probability theory facilitates nonmonotonic reasoning, i.e. reasoning involving retraction of previous conclusions. For example, it is perfectly acceptable to assert simultaneously $P(\text{Fly}(A)|\text{Bird}(A)) = \text{HIGH}$

and $P(\text{Fly}(A) | \text{Bird}(A), \text{Sick}(A)) = \text{LOW}$. In other words, if all we know about individual A is that A is bird, we jump to conclusion that A most likely flies. However, upon learning that A is also sick, we retract our old conclusion and assert that A most likely cannot fly.

To facilitate such retraction it is necessary both that original belief be stated with less than absolute certainty and that the context upon which we condition beliefs be consulted constantly to see whether belief revision is warranted. The dynamic of belief revision under changing contexts is also totally arbitrary but must obey some basic laws of plausibility which, fortunately, are embedded in the syntactical rules of probability calculus. A typical example of such a plausibility law is the rule of the hypothetical middle:

If two diametrically opposed assumptions impart two different degrees of belief onto a proposition Q , then the unconditional degree of belief merited by Q should be somewhere between the two.

3.3.4 Relevance

Relevance is a relationship indicating a potential change of belief due to a specified change in knowledge. Two propositions A and B are said to be relevant to each other in context C if adding B to C would change a likelihood of A . Clearly, relevance can be defined in terms of likelihood and conditioning, but it is a notion more basic than likelihood. For example, a person might be hesitant to assess the likelihood of two events but feel confident about judging whether or not events are relevant to each other. People

provide such judgments swiftly and consistently because – we speculate – relevance relationships are stored explicitly as pointers in one’s knowledge base.

Relevance is also a primitive of the language of probability because the language permits us to specify relevance relationships directly and qualitatively before making any numerical assessments. Later on, when numerical assessments of likelihood are required, they can be added in a consistent fashion, without disturbing the original relevance structure.

3.3.5 Causation

Causation is ubiquitous notion in man’s conception of his environment, yet it has traditionally been as a psychological construct, outside the province of probability or even the physical science.

Causation is listed as one of the four basic primitives of the language of probability because it is an indispensable tool for structuring and specifying probabilistic knowledge and because the semantics of causal relationships are preserved by the syntax of probabilistic manipulations; no auxiliary devices are needed to force conclusions to conform with people’s conception of causation.

Causation is a language with which one can talk efficiently about certain structures of relevance relationships, with the objective of separating the relevant from superfluous. The asymmetry conveyed by causal directionality is viewed as a notational device for encoding still more intricate patterns of relevance relationships, such as nontransitive and induced dependencies. For example, by designating *Rain* and *Sprinkler* as potential causes of the *Wet pavement* we permit the two causes to be independent of each other and

still both be relevant to *Wet pavement* (hence forming a nontransitive relationship). Moreover, by this designation we also identify the consequences *Wet pavement* and *Accident* as potential sources of new dependencies between the two causes; once a consequence is observed, its causes can no longer remain independent, because confirming one cause lowers the likelihood of the other. This connection between nontransitive and induced dependencies is, again, a built-in feature of the syntax of probability theory – the syntax ensures that nontransitive dependencies always induce the appropriate dependencies between causes.

To summarize, causal directionality conveys the following pattern of dependency: Two events do not become relevant to each other merely by virtue of predicting a common consequence, but they do become relevant when the consequence is actually observed. The opposite is true for two consequences of a common cause.

4 Bayesian networks

So far nothing has been said about the quantitative part of certainty assessment. Various certainty calculus exist, but here we treat the so called Bayesian calculus, which is classical probability calculus.

4.1 Basic axioms

The probability $P(A)$ of an event A is a number in the unit interval $[0,1]$. Probabilities obey the following basic axioms.

1. $P(A)=1$ iff A is certain
2. If A and B are mutually exclusive, then $P(A \vee B) = P(A) + P(B)$.

4.2 Conditional probabilities

The basic concept in the Bayesian treatment of certainties in casual networks is *conditional probability*. Whenever a statement of the probability, $P(A)$, of an event A is given, then it is given conditioned by other known factors. A statement like “The probability of the die turning up 6 is $\frac{1}{6}$ ” usually has the unsaid prerequisite that it is a fair die – or rather, as long as I know nothing of it, I assume it to be a fair die. This means that the statement should be “Given that it is a fair die, the probability...”. In this way, any statement on probabilities is a statement conditioned on what else is known.

A conditional probability statement is of the following kind:

Given the event B , the probability of the event A is x .

The notation of the statement above is $P(A | B) = x$.

It should be stressed that $P(A | B) = x$ does not mean that whether B is true then the probability of A is x. It means that if B is true, and *everything else known is irrelevant for A*, then $P(A) = x$.

The fundamental rule for probability calculus is the following:

$$P(A | B) \cdot P(B) = P(A, B) \quad (1)$$

where $P(A, B)$ is the probability of the joint event $A \wedge B$. Remembering that probabilities should always be conditioned by a context C, the formula should read

$$P(A | B, C) \cdot P(B | C) = P(A, B | C). \quad (2)$$

From (1) it follows that $P(A | B) \cdot P(B) = P(B | A) \cdot P(A)$ and this yields the well known Bayes' rule:

$$P(B | A) = \frac{P(A | B) \cdot P(B)}{P(A)} \quad (3)$$

Bayes' rule conditioned on C reads

$$P(B | A, C) = \frac{P(A | B, C) \cdot P(B | C)}{P(A | C)}$$

Formula (2) should be considered an axiom for probability calculus rather than a theorem.

4.3 Likelihood

Sometimes $P(A|B)$ and is denoted $L(B|A)$.

The reason for this is the following. Assume B_1, \dots, B_n are possible scenarios with an effect on the event A , and we know A . Then $P(A|B_i)$ is a measure of how likely it is that B_i is the case. In particular, if all B_i s has the same prior probability, Bayes' rule yields

$$P(B_i|A) = \frac{P(A|B_i) \cdot P(B_i)}{P(A)} = k \cdot P(A|B_i)$$

where k is independent of i .

4.4 Subjective probabilities

The justification in the previous section for the fundamental rule was based on frequencies. This doesn't mean that we only consider probabilities based on frequencies. Probabilities also may be completely subjective estimates of the certainty of the event.

The notion of subjective probability initially appeared in works of Scottish philosopher David Hume (1711-1776). He clearly separated the notion of analytical and empirical claims, the former are product of thoughts, the latter matter of facts. Hume also identified the source of all empirical claims with human experience, namely sensory inputs.

The central theme is to view subjective estimates a computational scheme devised to facilitate prediction of the effects of actions.

4.5 Probability calculus for variables

The nodes in a casual network are variables with a finite number of mutually exclusive states.

If A is a variable with states a_1, \dots, a_n an, then $P(A)$ is a probability distribution over these states:

$$P(A) = (x_1, \dots, x_n) \quad x_i \geq 0 \quad \sum_{i=1}^n x_i = 1.$$

where x_i is the probability of A being in state a_i .

Notation. The probability of A being in state a_i is denoted $P(A = a_i)$ and denoted $P(a_i)$ if the variable is obvious from the context.

If the variable B has states b_1, \dots, b_m , then $P(A|B)$ is an $n \times m$ table containing numbers $P(a_i | b_j)$ (see table 1).

Table 1. An example of $P(A | B)$. The columns sum to one.

	b_1	b_2	b_3
a_1	0.4	0.3	0.6
a_2	0.6	0.7	0.4

Table 2. An example of $P(A, B)$. The sum of all entries is one.

	b_1	b_2	b_3
a_1	0.16	0.12	0.12
a_2	0.24	0.28	0.08

Table 3. An example of $P(B | A)$ as a result of applying Bayes' rule to Table 1 and

$$P(B) = (0.4 \quad 0.4 \quad 0.2).$$

	a_1	a_2
b_1	0.4	0.4
b_2	0.3	0.47
b_3	0.3	0.13

$P(A, B)$, the joint probability for the variables A and B, is also $n \times m$ table. It consists of a probability for each configuration (a_i, b_j) .

When the fundamental rule (1) is used on variables A and B, then the procedure is to apply the rule to the $m \cdot n$ configurations (a_i, b_j) :

$$P(a_i | b_j) \cdot P(b_j) = P(a_i, b_j).$$

This means that in the table $P(A|B)$, for each j the column for b_j is multiplied by $P(b_j)$ to obtain the table $P(A,B)$. If $P(B) = (0.4 \ 0.4 \ 0.2)$ then Table 2 is the result of using the fundamental rule on Table 1. When applied to variables, we use the same notation for the fundamental rule:

$$P(A|B) \cdot P(B) = P(A,B).$$

From a table $P(A,B)$ the probability distribution $P(A)$ can be calculated. Let a_i be a state of A. There are exactly m different events for which A is in state a_i , namely the mutually exclusive events $(a_i, b_1), \dots, (a_i, b_m)$. Therefore, by axiom (2)

$$P(a_i) = \sum_{j=1}^m P(a_i, b_j).$$

This calculation is called marginalization and we say that the variable B is marginalized out of $P(A,B)$ (resulting in $P(A)$). The notation is

$$P(A) = \sum_B P(A,B).$$

By marginalizing B out of Table 2 we get $P(A) = (0.4 \ 0.6)$.

The division in Bayes' rule (3) is treated in the same way as the multiplication in the fundamental rule (see Table 3).

4.6 Conditional independence

The blocking of transmission of evidence is, as described in Bayesian calculus, reflected in the concept of *conditional independence*. The variables A and C are independent given the variable B if

$$P(a_i | b_j) \cdot P(a_i | b_j, c_k) \text{ for all } i, j, k.$$

This mean that if the state of B is known then no knowledge of C will alter the probability of A. If condition B is empty, we simply say that A and C are independent.

5 Problem definition

5.1 Definitions

An increasing number of academic and commercial endeavors use belief networks to encode uncertain knowledge in complex domains. These networks rely on inference algorithms to compute beliefs of alternative hypotheses in the context of observed evidence. However, the task of realizing an inference algorithm is not trivial. Much effort is spent synthesizing methods that are scattered throughout the literature and converting them to algorithmic form. Additional effort is spent addressing undocumented, lower-level issues that are vital to producing a robust and efficient implementation. These issues exist, in the words of one colleague, as “open secrets” within the probabilistic inference community.

Here we introduce in procedural fashion, the Probability Propagation in Trees of Clusters (PPTC) method for probabilistic inference, as developed by Lauritzen and Spiegelhalter and refined by Jensen. The description of algorithm is taken from [1].

PPTC is a method for performing probabilistic inference on a belief network. Consider the belief network shown in Figure 1. An example of probabilistic inference would be to compute the probability that $A = on$, given the knowledge that $C = on$ and $E = off$. In general, probabilistic inference on a belief network is the process of computing $P(V = v \mid \mathbf{E} = \mathbf{e})$, or simply $P(v \mid \mathbf{e})$, where v is a value of a variable V and \mathbf{e} is an assignment of values to a set of variables \mathbf{E} in the belief network. Basically, $P(v \mid \mathbf{e})$ asks: Suppose that I observe \mathbf{e} on a set of variables \mathbf{E} , what is the probability that the variable V has value v , given \mathbf{e} ?

PPTC works in two steps. First, a belief network is converted into a secondary structure. Then, probabilities of interest are computed by operating on that secondary structure.

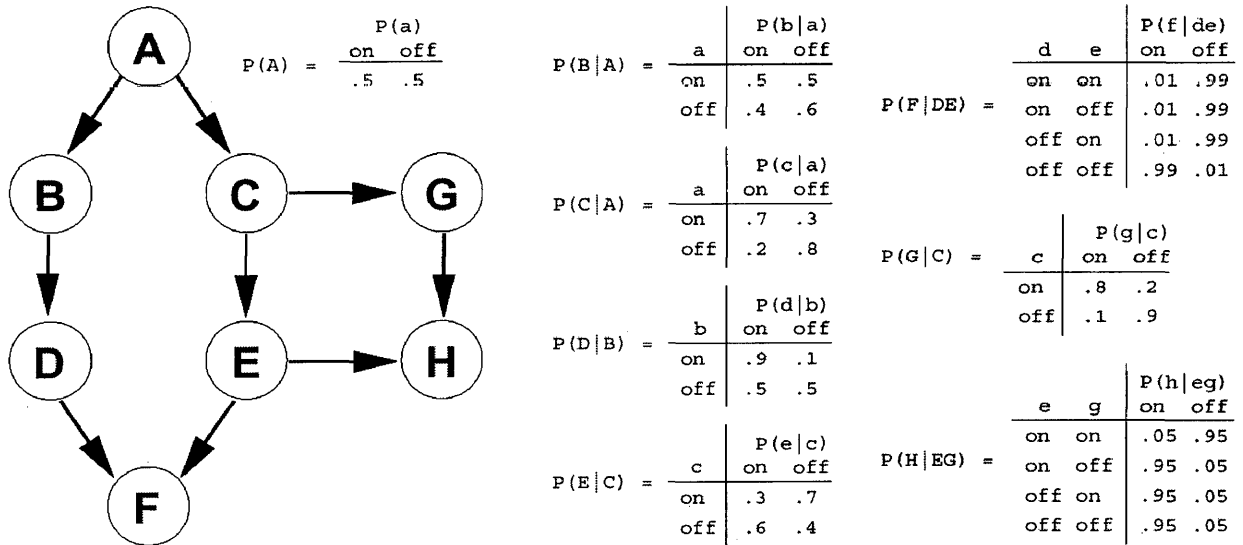


Figure 1. Sample belief network

We specify PPTC using the following notational conventions and fundamental concepts.

5.1.1 Variables And Values

We denote variables with uppercase letters (A, B, C), and variable values with lowercase letters (a, b, c). We instantiate a variable A by assigning it a value a; we call an instantiation of A.

Sets of variables are denoted by boldface uppercase letters (X, Y, Z), and their instantiations by boldface lowercase letters (x, y, z). We instantiate a set of variables X

by assigning a value to each variable in \mathbf{X} ; we denote this assignment with \mathbf{x} , and call \mathbf{x} an instantiation of \mathbf{X} .

5.1.2 Potentials and distributions

We define a potential over a set of variables \mathbf{X} as a function that maps each instantiation \mathbf{x} into a nonnegative real number; we denote this potential as $\phi_{\mathbf{X}}$. We use the notation $\phi_{\mathbf{X}}(\mathbf{x})$ to denote the number that $\phi_{\mathbf{X}}$ maps \mathbf{x} into; we call $\phi_{\mathbf{X}}(\mathbf{x})$ an element. Potentials can be viewed as matrices and implemented as tables, so we will also refer to them as matrices and tables. Example of potential's tables presented on fig. 2.

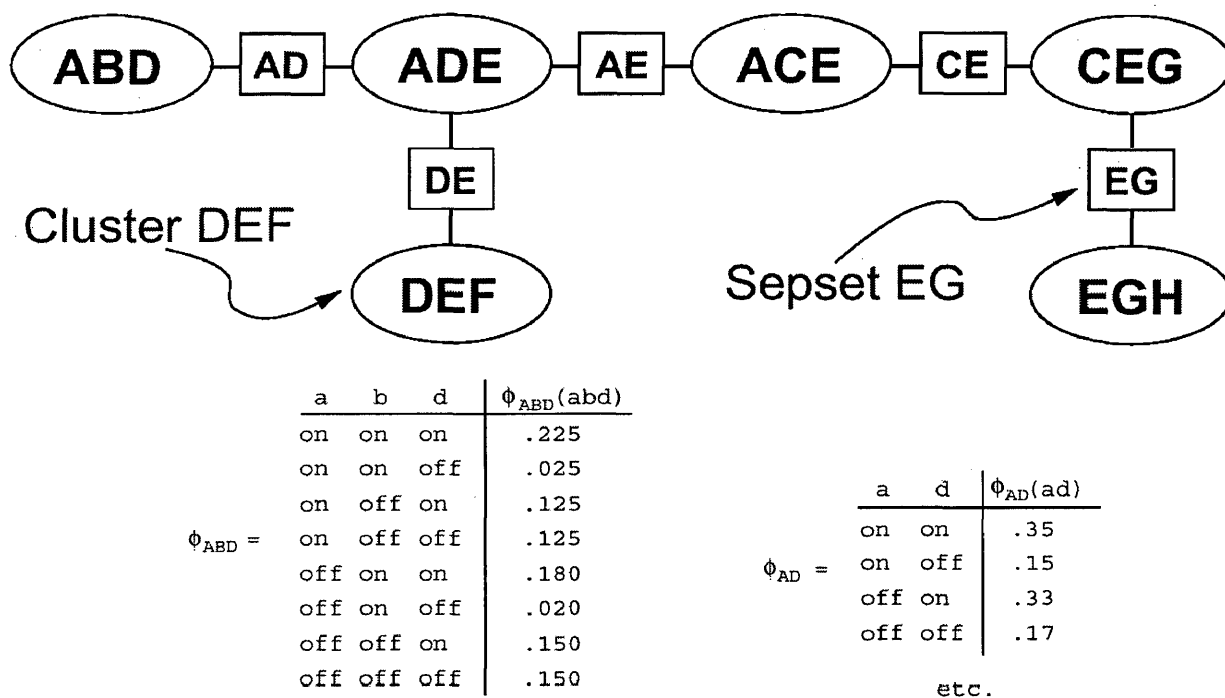


Figure 2. Secondary structure for sample belief network

We define two basic operation on potentials: *marginalization* and *multiplication*.

1. Marginalization

Suppose we have a set of variables \mathbf{Y} , its potential $\phi_{\mathbf{Y}}$, and a set of variables \mathbf{X} where $\mathbf{X} \subseteq \mathbf{Y}$. The marginalization of $\phi_{\mathbf{Y}}$ into \mathbf{X} is a potential $\phi_{\mathbf{X}}$, where each $\phi_{\mathbf{X}}(\mathbf{x})$ is computed as follows:

- a) Identify the instantiations $\mathbf{y}_1, \mathbf{y}_2, \dots$ that are consistent with \mathbf{x} .
- b) Assign to $\phi_{\mathbf{X}}(\mathbf{x})$ the sum $\phi_{\mathbf{Y}}(\mathbf{y}_1) + \phi_{\mathbf{Y}}(\mathbf{y}_2) + \dots$

$$\varphi_{\mathbf{X}} = \sum_{\mathbf{Y} \setminus \mathbf{X}} \varphi_{\mathbf{Y}}.$$

This marginalization is denoted as follows:

2. Multiplication

Given two sets of variables \mathbf{X} and \mathbf{Y} and their potentials $\phi_{\mathbf{X}}$ and $\phi_{\mathbf{Y}}$, the multiplication of $\phi_{\mathbf{X}}$ and $\phi_{\mathbf{Y}}$ is a potential $\phi_{\mathbf{Z}}$, where $\mathbf{Z} = \mathbf{X} \cup \mathbf{Y}$, and each $\phi_{\mathbf{Z}}(\mathbf{z})$ is computed as follows:

- a) Identify the instantiation \mathbf{x} and the instantiation \mathbf{y} that are consistent with \mathbf{z} .
- b) Assign to $\phi_{\mathbf{Z}}(\mathbf{z})$ the product $\phi_{\mathbf{X}}(\mathbf{x}) \phi_{\mathbf{Y}}(\mathbf{y})$.

$$\varphi_{\mathbf{Z}} = \varphi_{\mathbf{X}} \cdot \varphi_{\mathbf{Y}}.$$

This multiplication of potentials is denoted as follows:

Probability Distributions A probability distribution, or simply a distribution, is a special case of a potential. Given a set of variables \mathbf{X} , we use the notation $P(\mathbf{X})$ to denote the probability distribution of \mathbf{X} , or simply the probability of \mathbf{X} . $P(\mathbf{X})$ is a potential over

\mathbf{X} whose elements add up to 1. We denote the elements of $P(\mathbf{X})$ as $P(\mathbf{x})$, and we call each element $P(\mathbf{x})$ the probability of \mathbf{x} . With this notation, we have

$$\sum_{\mathbf{x}} P(\mathbf{x}) = 1.$$

Another important notion is that of conditional probability. Given sets of variables \mathbf{X} and \mathbf{Y} , we use the notation $P(\mathbf{X} | \mathbf{Y})$ to denote the conditional probability of \mathbf{X} given \mathbf{Y} , or simply the probability of \mathbf{X} given \mathbf{Y} . $P(\mathbf{X} | \mathbf{Y})$ is a collection of probability distributions indexed by the instantiations of \mathbf{Y} ; each $P(\mathbf{X} | \mathbf{y})$ is a probability distribution over \mathbf{X} .

We denote the elements of $P(\mathbf{X} | \mathbf{y})$ as $P(\mathbf{x} | \mathbf{y})$, and we call each element $P(\mathbf{x} | \mathbf{y})$ the probability of \mathbf{x} given \mathbf{y} . With this notation, we have, for each instantiation \mathbf{y} ,

$$\sum_{\mathbf{x}} P(\mathbf{x} | \mathbf{y}) = 1.$$

Note that $P(\mathbf{X})$ is a special case of $P(\mathbf{X} | \mathbf{Y})$ where $\mathbf{Y} = \emptyset$.

5.1.3 Belief network

Belief Networks are used by experts to encode selected aspects of their knowledge and beliefs about a domain. Once constructed, the network induces a probability distribution over its variables.

Definition A belief network over a set of variables $U = \{V_1, \dots, V_n\}$ consists of two components:

- A **directed acyclic graph (DAG)** G : Each vertex in the graph represents a variable V , which takes on values v_1, v_2 , etc. The parents of V in the graph are denoted by Π_V . The family of V , denoted by F_V , is defined as $\{V\} \cup \Pi_V$. The DAG structure encodes a set of **independence assertions**, which restrict the variety of interactions that can occur among variables.
- A **quantification** of G : Each variable in G is quantified with a conditional probability table $P(V | \Pi_V)$. While $P(V | \Pi_V)$ is technically a function of F_V , it is most helpful to think of it in the following way: for each instantiation π_V , real numbers in $[0; 1]$ are assigned to each value v , such that they add up to 1. When $\Pi_V \neq \emptyset$, $P(V | \Pi_V)$ is called the **conditional probability** of V given Π_V . When $\Pi_V = \emptyset$, $P(V | \Pi_V)$, or simply $P(V)$, is called the **prior probability** of V .

5.1.4 The secondary structure

While experts typically use belief networks to encode their domain, PPTC performs probabilistic inference on a secondary structure that we characterize below.

Definition

Given a belief network over a set of variables $U = \{V_1, \dots, V_n\}$, we define a secondary structure that contains a graphical and a numerical component. The graphical component consists of the following:

- An undirected *tree* T : Each node in T is a cluster (nonempty set) of variables. The clusters satisfy the join tree property: given two clusters X and Y in T , all clusters

on the path between \mathbf{X} and \mathbf{Y} contain $\mathbf{X} \cap \mathbf{Y}$. For each variable $V \in \mathbf{U}$, the family of V , \mathbf{F}_V (Section 5.1.5), is included in at least one of the clusters.

- *Sepsets*: Each edge in \mathbf{T} is labeled with the intersection of the adjacent clusters; these labels are called separator sets, or sepsets.

The numerical component is described using the notion of a belief potential. A belief potential is a function that maps each instantiation of a set of variables into a real number (Section 5.1.2). Belief potentials are defined over the following sets of variables:

- *Clusters*: Each cluster \mathbf{X} is associated with a belief potential $\phi_{\mathbf{X}}$ that maps each instantiation \mathbf{x} into a real number.
- *Sepsets*: Each sepset \mathbf{S} is associated with a belief potential $\phi_{\mathbf{S}}$ that maps each instantiation \mathbf{s} into a real number.

The belief potentials are not arbitrarily specified; they must satisfy the following constraints:

- For each cluster \mathbf{X} and neighboring sepset \mathbf{S} , it holds that

$$\sum_{\mathbf{X} \setminus \mathbf{S}} \phi_{\mathbf{X}} = \phi_{\mathbf{S}}. \quad (1)$$

When Equation 1 is satisfied for a cluster \mathbf{X} and neighboring sepset \mathbf{S} , we say that $\phi_{\mathbf{S}}$ is *consistent* with $\phi_{\mathbf{X}}$. When consistency holds for every cluster-sepset pair, we say that the secondary structure is *locally consistent*.

- The belief potentials encode the joint distribution $P(\mathbf{U})$ of the belief network according to

$$P(U) = \frac{\prod_i \phi_{X_i}}{\prod_j \phi_{S_j}}. \quad (2)$$

where ϕ_{X_i} and ϕ_{S_i} are the cluster and sepset potentials, respectively.

A key step in PPTC is the construction of a secondary structure that satisfies the above constraints. Such a secondary structure has the following important property: for each cluster (or sepset) \mathbf{X} , it holds that $\phi_{\mathbf{X}} = P(\mathbf{X})$. Using this property, we can compute the probability distribution of any variable V , using any cluster (or sepset) \mathbf{X} that contains V , as follows:

$$P(V) = \sum_{\mathbf{X} \setminus \{V\}} \phi_{\mathbf{X}}. \quad (3)$$

The secondary structure has been referred to in the literature as a *join tree*, *junction tree*, *tree of belief universes*, *cluster tree*, and *clique tree*, among other designations. In this document, we use the term *join tree* to refer to the graphical component, and the term *join tree potential* to refer generically to a cluster or sepset belief potential. We will also use the term *join tree* to refer to the entire secondary structure, as it is being created; the meaning of *join tree* will be clear from the context. In Section 5.2, we show how to build a join tree from the DAG of a belief network, and in Section 5.3, we describe how PPTC manipulates the join tree potentials so that they satisfy Equations (1) and (2).

5.2 Building the secondary structure

We begin with the DAG of a belief network (fig. 1), and apply a series of graphical transformations that result in a join tree. These transformations involve a number of intermediate structures, and can be summarized as follows:

1. Construct an undirected graph, called a moral graph, from the DAG.
2. Selectively add arcs to the moral graph to form a triangulated graph.
3. From the triangulated graph, identify select subsets of nodes, called **cliques**.
4. Build a join tree, starting with the cliques as clusters: connect the clusters to form an undirected tree satisfying the join tree property, inserting the appropriate sepsets.

Steps 2 and 4 are non-deterministic; consequently, many different join trees can be built from the same DAG.

5.2.1 Constructing The Moral Graph

Let G be the DAG of a belief network. The moral graph G_M corresponding to G is constructed as follows:

1. Create the undirected graph G_u by copying G , but dropping the directions of the arcs.
2. Create G_M from G_u as follows: For each node V , identify its parents Π_V in G .

Connect each pair of nodes in Π_V by adding undirected arcs to G_u .

Figure 3 illustrates this transformation on the DAG from Fig. 1. The undirected arcs added to G_u are called moral arcs, shown as dashed lines in the figure.

5.2.2 *Triangulating the Moral Graph*

An undirected graph is triangulated iff every cycle of length four or greater contains an edge that connects two nonadjacent nodes in the cycle.

1. Make a copy of G_M ; call it G'_M .
2. While there are still nodes left in G'_M :
 - a) Select a node V from G'_M , according to the criterion described below.
 - b) The node V and its neighbors in G'_M form a cluster. Connect all of the nodes in this cluster. For each edge added to G'_M , add the same corresponding edge to G_M .
 - c) Remove V from G'_M .
3. G_M , modified by the additional arcs introduced in the previous steps, is now triangulated.

To describe the criterion for selecting the nodes in Step 2a, we rely on the following notion of a weight:

- The weight of a node V is the number of values of V .
- The weight of a cluster is the product of the weights of its constituent nodes.

The criterion for selecting nodes to remove is now stated as follows:

Choose the node that causes the least number of edges to be added in Step 2b, breaking ties by choosing the node that induces the cluster with the smallest weight.

Here we access the next node to be removed by of G_M a binary heap. Each node V is associated with a primary key (the number of edges added if V were selected next) and a secondary key (the weight of the cluster induced if V were selected next). When V is removed, each of V 's neighbors needs to have its keys recalculated, and, therefore, its position in the heap modified. Removing a node V costs $O(k \cdot \lg n)$ time, where k is the number of neighbors of V in G_M' , and n is the number of nodes remaining in G_M' .

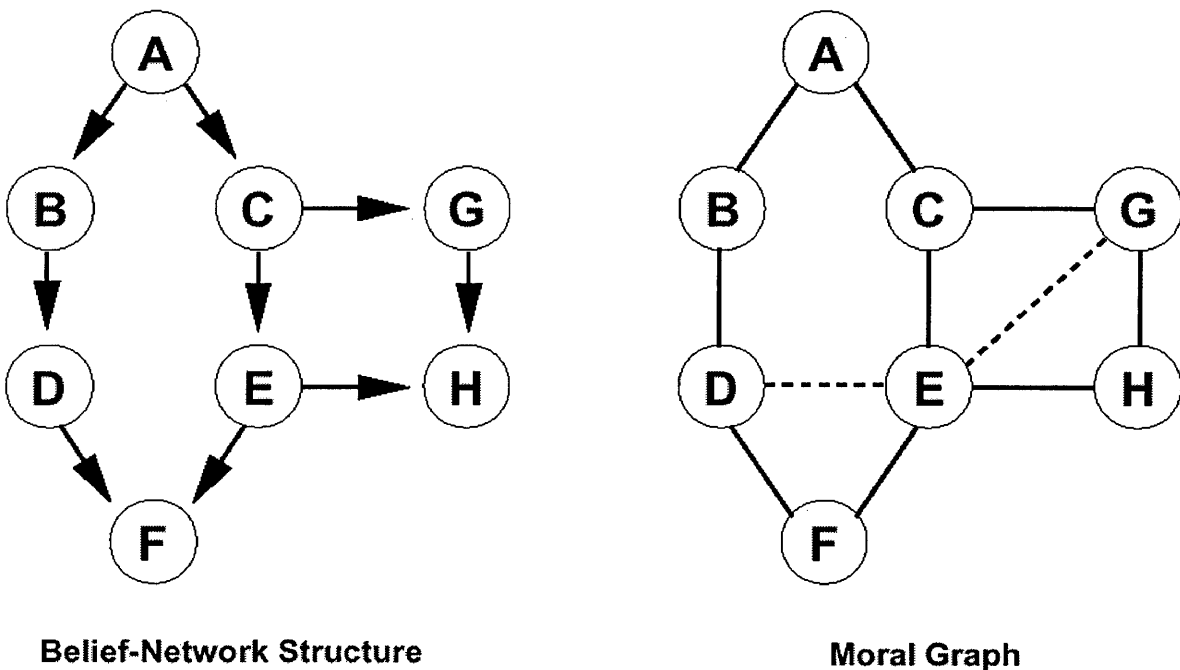
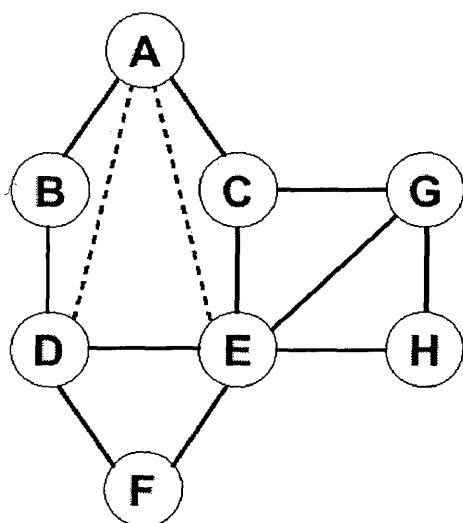


Figure 3. Moral graph

Fig. 4 depicts the triangulated graph, as obtained from the moral graph in Fig. 3. The dashed lines in the figure indicate the edges added to triangulate the moral graph. Elimination ordering of the nodes is also shown, so that the interested reader can trace each step in the triangulation process.

In general, there are many ways to triangulate an undirected graph. An optimal triangulation is one that minimizes the sum of the state space sizes of the cliques of the triangulated graph. The task of finding an optimal triangulation is *NP*-complete. However, the node-selection criterion in Step 2a is a greedy, polynomial-time heuristic that produces high-quality triangulations in real-world settings.



Triangulated Graph

Eliminated Vertex	Induced Cluster	Edges Added
H	EGH	none
G	CEG	none
F	DEF	none
C	ACE	(A, E)
B	ABD	(A, D)
D	ADE	none
E	AE	none
A	A	none

Elimination Ordering

Figure 4. Triangulating the moral graph

5.2.3 Identifying cliques

A clique in an undirected graph G is a subgraph of G that is complete and maximal. Complete means that every pair of distinct nodes is connected by an edge. Maximal means that the clique is not properly contained in a larger, complete subgraph.

By adapting the triangulation procedure in Section 5.2.2, though, we can identify the cliques of the triangulated graph as it is being constructed.

Our procedure relies on the following two observations:

- Each clique in the triangulated graph is an induced cluster from step 2b of Section 5.2.2.
- An induced cluster can never be a subset of a subsequently induced cluster.

These observations suggest that we can extract the cliques during the triangulation process by saving each induced cluster that is not a subset of any previously saved cluster. Revisiting Figure 2, we see that the cliques of the triangulated graph are EGH , CEG , DEF , ACE , ABD , and ADE .

5.2.4 Building an optimal join tree

From this point on, we no longer need the undirected graph. We seek to build an optimal join tree by connecting the cliques obtained in Section 5.2.3 above. To build an optimal join tree, we must connect the cliques so that the resulting clique tree satisfies the join tree property and an optimality criterion that we will define below. The join tree

property is essential for the tree to be useful for probabilistic inference, and the optimality criterion favors those join trees that minimize the computational time required for inference [4].

Given a set of n cliques, we can form a clique tree by iteratively inserting edges between pairs of cliques, until the cliques are connected by $n - 1$ edges.

We can also view this task as iteratively inserting sepsets between pairs of cliques, until the cliques are connected by $n - 1$ sepsets. We take this latter approach in specifying how to build an optimal join tree. We divide our specification of the algorithm into two parts: First, in Section 5.2.5, we provide a generic procedure that forms a clique tree by iteratively selecting and inserting candidate sepsets. Then, in Section 5.2.6, we show how the sepsets must be chosen, in order for the clique tree to be an optimal join tree.

The following procedure builds an optimal join tree by iteratively selecting and inserting candidate sepsets; the criterion in Step 3a is specified later in Section 5.2.6 below.

5.2.5 Building an Optimal Join Tree

1. Begin with a set of n trees, each consisting of a single clique, and an empty set S .
2. For each distinct pair of cliques X and Y :
 - a) Create a candidate sepset, labeled $X \cap Y$, with backpointers to the cliques X and Y . Refer to this sepset as S_{XY} .
 - b) Insert S_{XY} into S .
3. Repeat until $n - 1$ sepsets have been inserted into the forest.

- a) Select a sepset S_{XY} from S , according to the criterion specified in Section 5.2.6. Delete S_{XY} from S .
- b) Insert the sepset S_{XY} between the cliques X and Y *only if* X and Y are on different trees in the forest. (Note that the insertion of such a sepset would merge two trees into a larger tree.)

5.2.6 Choosing The Appropriate Sepsets

In order to describe how to choose the next candidate sepset, we define the notions of mass and cost, as follows:

- The *mass* of a sepset S_{XY} is the number of variables it contains, or the number of variables in $X \cap Y$.
- The *cost* of a sepset S_{XY} is the weight of X plus the weight of Y , where weight is defined as follows:
 - The *weight* of a variable V is the number of values of V .
 - The *weight* of a set of variables X is the product of the weights of the variables in X .

With these notions established, we can now state how to select the next candidate sepset from S whenever we execute Step 3a in Section 5.2.5:

- For the resulting clique tree to satisfy the join tree property, we must *choose the candidate sepset with the largest mass*.

- When two or more sepsets of equal mass can be chosen, we can optimize the inference time on the resulting join tree by breaking the tie as follows: *choose the candidate sepset with the smallest cost.*

5.3 Further steps in making inference

After we build joint tree structure, we need to do the following steps to make an inference (fig. 5).

- *Initialization.* Quantify the join tree with belief potentials. The result is an inconsistent join tree, as this initial assignment of belief potentials does not meet the local consistency requirements.
- *Global Propagation.* Perform an ordered series of local manipulations, called message passes, on the join tree potentials. The message passes rearrange the join tree potentials so that they become locally consistent; thus, the result of global propagation is a consistent join tree, which satisfies both Equations (1) and (2).
- *Marginalization.* From the consistent join tree, compute $P(V)$ for each variable of interest V .

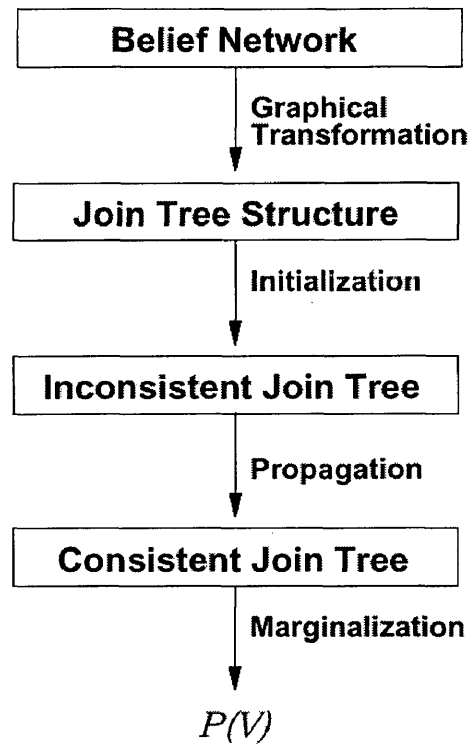


Figure 5. Making further inference after graphical transformation

5.3.1 Initialization

The following procedure assigns initial join tree potentials, using the conditional probabilities from the belief network:

1. For each cluster and sepset \mathbf{X} , set each $\phi_{\mathbf{X}}(\mathbf{X})$ to 1:

$$\phi_{\mathbf{X}} \leftarrow 1.$$

2. For each variable V , perform the following: Assign to V a cluster \mathbf{X} that contains F_V ; call \mathbf{X} the parent cluster of F_V . Multiply $\phi_{\mathbf{X}}(\mathbf{X})$ by $P(V | \Pi_V)$:

$$\phi_{\mathbf{X}} \leftarrow \phi_{\mathbf{X}} P(V | \Pi_V).$$

After initialization, the conditional distribution $P(V | \Pi_V)$ of each variable V has been multiplied into some cluster potential. The initialization procedure satisfies Equation (2) 5.1.4 as follows:

$$\frac{\prod_{i=1}^N \phi_{X_i}}{\prod_{j=1}^{N-1} \phi_{S_j}} = \frac{\prod_{k=1}^Q P(V_k | C_{V_k})}{1} = P(U).$$

where N is the number of clusters,

Q is the number of variables,

and ϕ_{X_i} and ϕ_{S_j} are the cluster and sepset potentials, respectively.

Fig. 6 illustrates the initialization procedure on the tables of cluster ACE and sepset CE from the secondary structure of Figure 1. In this example, ACE is the parent cluster of F_C and F_E , but not F_A . Thus, after initialization, $\phi_{ACE} = P(C | A)P(E | C)$, and $\phi_{CE} = 1$.

5.3.2 Global propagation

Having initialized the join tree potentials, we now perform global propagation in order to make them locally consistent. Global propagation consists of a series of local manipulations, called *message passes*, that occur between a cluster X and a neighboring cluster Y . A message pass from X to Y forces the belief potential of the intervening sepset to be consistent with ϕ_X (see Equation (1)) 5.1.4, while preserving the invariance of Equation (2) 5.1.4. Global propagation causes each cluster to pass a message to each of

its neighbors; these message passes are ordered so that each message pass will preserve the consistency introduced by previous message passes. When global propagation is completed, each cluster-sepset pair is consistent, and the join tree is locally consistent.

In numbering item 1 below, we describe a single message pass between two adjacent clusters. Then in numbering item 2, we explain how global propagation achieves local consistency by coordinating multiple message passes.

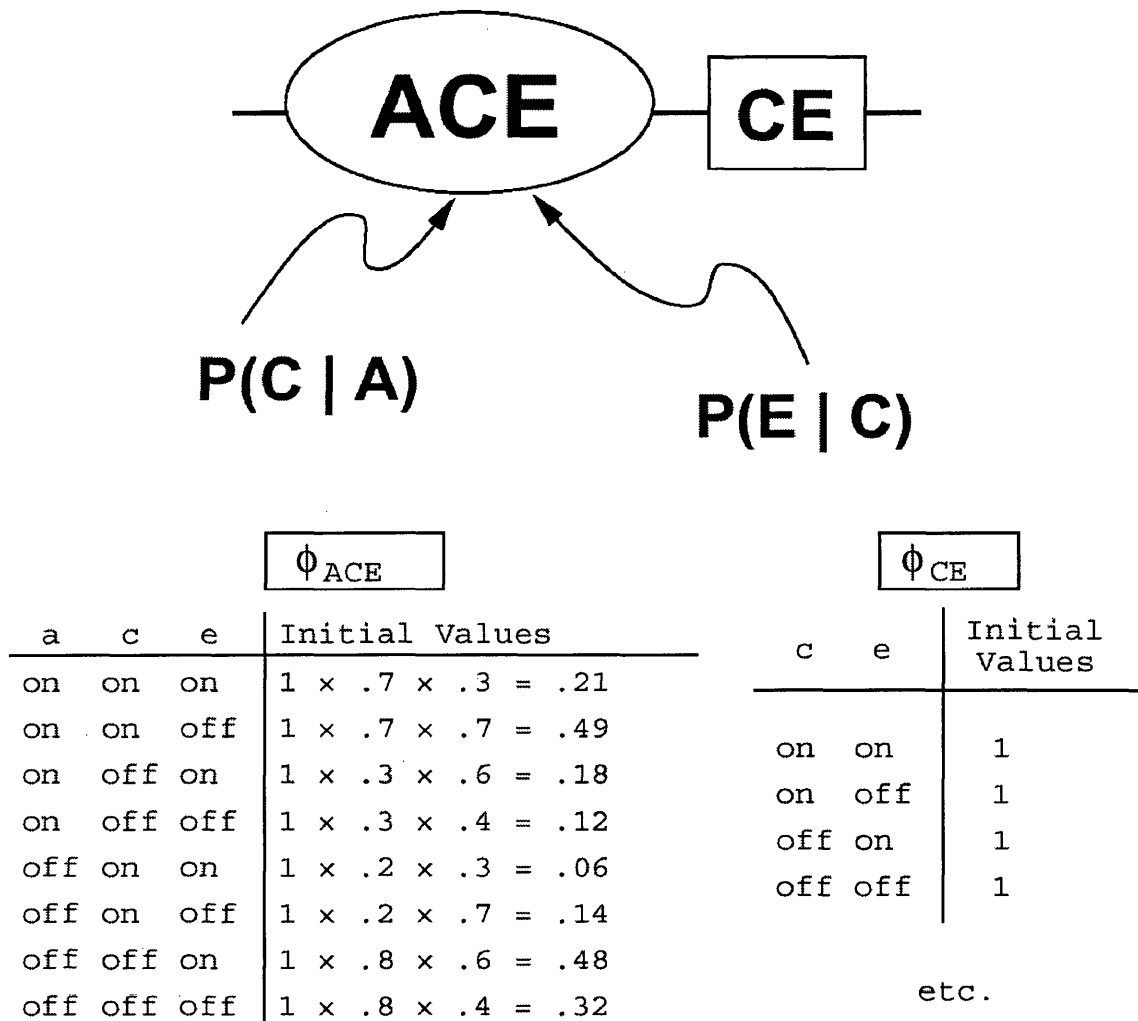


Figure 6. Initialization of cluster ACE and sepset CE

1. *Single Message Pass* Consider two adjacent clusters \mathbf{X} and \mathbf{Y} with sepset \mathbf{R} , and their associated belief potentials $\phi_{\mathbf{X}}$, $\phi_{\mathbf{Y}}$, and $\phi_{\mathbf{R}}$. A *message pass* from \mathbf{X} to \mathbf{Y} occurs in two steps:

- a) *Projection*. Assign a new table to \mathbf{R} , saving the old table:

$$\begin{aligned}\phi_{\mathbf{R}}^{old} &\leftarrow \phi_{\mathbf{R}}. \\ \phi_{\mathbf{R}} &\leftarrow \sum_{\mathbf{X} \setminus \mathbf{R}} \phi_{\mathbf{X}}.\end{aligned}\tag{1}$$

- b) *Absorption*. Assign a new value to \mathbf{Y} , using both the old and the new tables of \mathbf{R} :

$$\phi_{\mathbf{Y}} \leftarrow \phi_{\mathbf{Y}} \frac{\phi_{\mathbf{R}}}{\phi_{\mathbf{R}}^{old}}.\tag{2}$$

Equations (1) and (2) assign new potentials to \mathbf{R} and \mathbf{Y} ; however, the left-hand side of equation (2) remains constant, thus preserving the invariance of Equation (2) 5.1.4:

$$\left(\frac{\prod_i \phi_{X_i}}{\prod_j \phi_{S_j}} \right) \frac{\phi_{\mathbf{R}}^{old}}{\phi_{\mathbf{R}}} \frac{\phi_{\mathbf{Y}}}{\phi_{\mathbf{Y}}^{old}} = \left(\frac{\prod_i \phi_{X_i}}{\prod_j \phi_{X_j}} \right) \frac{\phi_{\mathbf{R}}^{old}}{\phi_{\mathbf{R}}} \frac{\phi_{\mathbf{Y}}^{old}}{\phi_{\mathbf{Y}}^{old}} \frac{\phi_{\mathbf{R}}}{\phi_{\mathbf{R}}^{old}} = P(\mathbf{U}).$$

2. *Coordinating Multiple Messages* Given a join tree with n clusters, the PPTC global propagation algorithm begins by choosing an arbitrary cluster \mathbf{X} , and then performing $2 \cdot (n - 1)$ message passes, divided into two phases. During the *Collect-Evidence* phase, each cluster passes a message to its neighbor in \mathbf{X} 's direction, beginning with the clusters farthest from \mathbf{X} . During the *Distribute-Evidence* phase, each cluster passes messages to its neighbors away from \mathbf{X} 's

direction, beginning with \mathbf{X} itself. The *Collect-Evidence* phase causes $n - 1$ messages to be passed, and the *Distribute-Evidence* phase causes another $n - 1$ messages to be passed.

Global propagation starts from arbitrary cluster and uses Deep First Search (DFS) algorithms to collect and propagate evidence in the cluster tree.

Steps are the following:

1. Choose an arbitrary cluster \mathbf{X} .
2. Unmark all clusters. Call $\text{Collect-Evidence}(\mathbf{X})$ function.
3. Unmark all clusters. Call $\text{Distribute-Evidence}(\mathbf{X})$ function.

COLLECT-EVIDENCE (\mathbf{X})

1. Mark \mathbf{X} .
2. Call Collect-Evidence recursively on \mathbf{X} 's unmarked neighboring clusters, if any.
3. Pass a message from \mathbf{X} to the cluster, which invoked $\text{Collect-Evidence}(\mathbf{X})$.

DISTRIBUTE-EVIDENCE (\mathbf{X})

1. Mark \mathbf{X} .
2. Pass a message from \mathbf{X} to each of its unmarked neighboring clusters, if any.
3. Call $\text{Distribute-Evidence}$ recursively on \mathbf{X} 's unmarked neighboring clusters, if any.

The net result of this message passing is that each cluster passes its information, as encoded in its belief potential, to all of the other clusters in the join tree. Note that in this message-passing scheme, a cluster passes a message to a neighbor only after it has

received messages from all of its other neighbors. This condition assures local consistency of the join tree when global propagation is completed.

Example on Fig. 7 illustrates the PPTC propagation step on the join tree from Fig. 1. Here, *ACE* is the starting cluster. During the Collect-Evidence phase, messages are passed in *ACE*'s direction, beginning with the clusters *ABD*, *DEF*, and *EGH*; these messages are indicated by the solid arrows. During the Distribute-Evidence phase, messages are passed away from cluster *ACE*, beginning with *ACE*; these messages are indicated by the dashed arrows. The numbers indicate one possible message passing order.

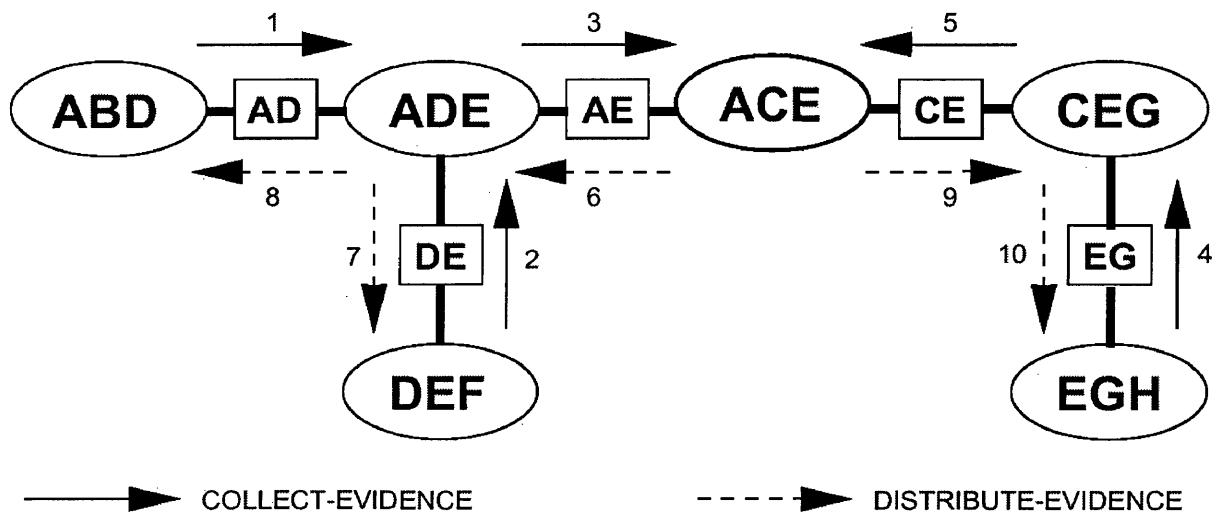


Figure 7. Message passing during global propagation

5.3.3 Marginalization

Once we have a consistent join tree, we can compute $P(V)$ for each variable of interest V as follows:

1. Identify a cluster (or sepset) X that contains V .
2. Compute $P(V)$ by marginalizing ϕ_X according to Equation (3) 5.1.4, repeated for convenience:

$$P(V) = \sum_{X \setminus \{V\}} \phi_X.$$

Fig. 8 illustrates an example of marginalization. The cluster potential ϕ_{ABD} is from the consistent join tree of Fig. 1. ϕ_{ABD} is marginalized once to compute $P(A)$, and then marginalized again to compute $P(D)$.

	a	b	d	$\phi_{ABD}(abd)$
$\phi_{ABD} =$	on	on	on	.225
	on	on	off	.025
	on	off	on	.125
	on	off	off	.125
	off	on	on	.180
	off	on	off	.020
	off	off	on	.150
	off	off	off	.150

	a	$P(a)$
$P(A) = \sum_{BD} \phi_{ABD} =$	on	.225 + .025 + .125 + .125 = .500
	off	.180 + .020 + .150 + .150 = .500

	d	$P(d)$
$P(D) = \sum_{AB} \phi_{ABD} =$	on	.225 + .125 + .180 + .150 = .680
	off	.025 + .125 + .020 + .150 = .320

Figure 8. Marginalization example.

5.4 Handling evidence

We are now able to compute $P(V)$ for any variable V . In the following sections, we show how to modify the procedures in Section 5.3 in order to compute $P(V | \mathbf{e})$ in the context of evidence \mathbf{e} . First we introduce observations, the simplest notion of evidence, in Section 5.4.1. Then, in Sections 5.4.2 – 5.5, we show how to compute $P(V | \mathbf{e})$ for sets of observations \mathbf{e} .

5.4.1 Observations and likelihoods

Observations are the simplest forms of evidence. An observation is a statement of the form $V = v$. Collections of observations may be denoted by $E = \mathbf{e}$, where \mathbf{e} is the instantiation of the set of variables \mathbf{E} . Observations are also referred to as *hard evidence*.

To encode observations in a form suitable for PPTC, we define the notion of a likelihood. Given a variable V , the *likelihood* of V , denoted as Λ_V , as a potential over $\{V\}$; in other words, Λ_V maps each value v to a real number (see Section 5.1.2). We encode an arbitrary set of observations \mathbf{e} by using a likelihood Λ_V for each variable V , as follows:

- If $V \in \mathbf{E}$ - that is, if V is observed - then assign each $\Lambda_V(v)$ as follows:

$$\Lambda_V(v) = \begin{cases} 1, & \text{when } v \text{ is the observed value of } V \\ 0, & \text{otherwise} \end{cases}$$

- If $V \notin \mathbf{E}$ - that is, if the value of V is unknown - then assign $\Lambda_V(v) = 1$ for each value v .

Note that when there are no observations, the likelihood of each variable consists of all 1's. Table 1 illustrates how likelihoods are used to encode the observations $C = on$ and $E = off$, where C and E are variables from the join tree in Fig. 1.

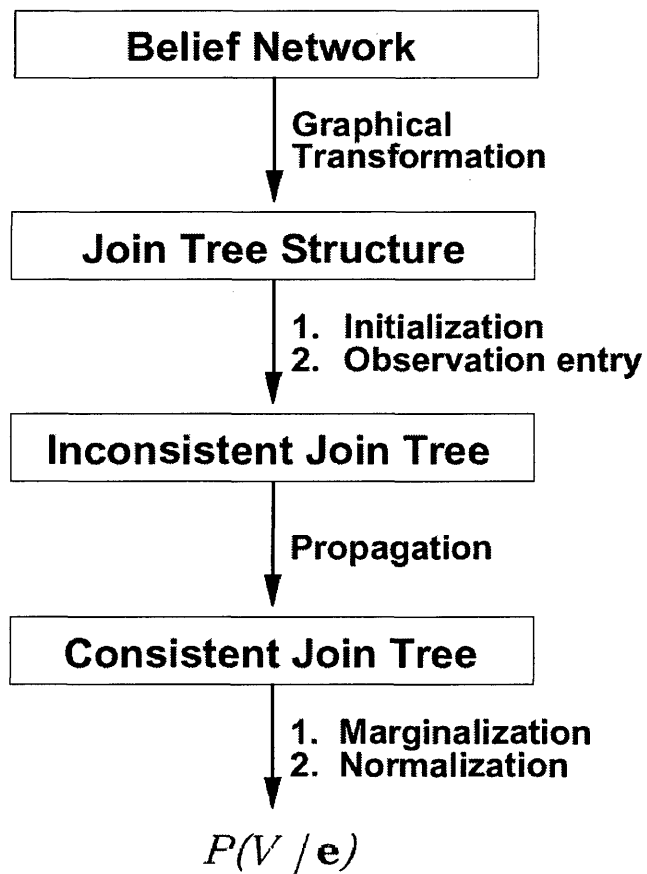
5.4.2 PPTC Inference With Observations

Fig. 9 illustrates the overall control for PPTC with observations. We modify the control from Fig. 5 to incorporate observations, as follows:

- *Initialization* (Section 5.3.1). We modify initialization from Section 5.3.1 by introducing an additional step: for each variable V , we initialize the likelihood Λ_V .

Table 1. Likelihood encoding of $C = on$, $E = off$.

Variable V	$\Lambda_V(v)$	
	$v = on$	$v = off$
A	1	1
B	1	1
C	1	0
D	1	1
E	0	1
F	1	1
G	1	1
H	1	1

**Figure 9.** Block diagram of PPTC with observations.

- *Observation Entry* (Section 5.4.4). Following initialization, we encode and incorporate observations into the join tree, this step results in further modification of the join tree potentials.
- *Normalization* (Section 5.4.5). To compute $P(V | e)$ for a variable of interest V , we perform marginalization and an additional step called *normalization*.

5.4.3 Initialization with observations

We keep track of observations by maintaining a likelihood for each variable. We initialize these likelihoods by adding step 2b to the initialization procedure below:

1. For each cluster and sepset X , set each $\phi_x(\mathbf{x})$ to 1:

$$\phi_x \leftarrow 1.$$

2. For each variable V :

- a) Assign to V a cluster X that contains F_V ; multiply ϕ_x by $P(V | \Pi_V)$:

$$\phi_x \leftarrow \phi_x P(V | \Pi_V).$$

- b) Set each likelihood element $\Lambda_V(v)$ to 1:

$$\Lambda_V \leftarrow 1.$$

5.4.4 Observation entry

Note that upon completion of initialization, the likelihoods encode no observations.

We incorporate each observation $V = v$ by encoding the observation as a likelihood, and then incorporating this likelihood into the join tree, as follows:

1. Encode the observation $V = v$ as a likelihood Λ_V^{new} .

2. Identify a cluster \mathbf{X} that contains V .
3. Update $\phi_{\mathbf{X}}$ and Λ_V :

$$\phi_{\mathbf{X}} \leftarrow \phi_{\mathbf{X}} \Lambda_V^{new}. \quad (1)$$

$$\Lambda_V \leftarrow \Lambda_V^{new}.$$

By entering a set of observations \mathbf{e} as described above, we modify the join tree potentials, so that all subsequent probabilities derived from the join tree are probabilities of events that are conjoined with evidence \mathbf{e} . In other words, instead of computing $P(\mathbf{X})$ and $P(V)$, we compute $P(\mathbf{X} | \mathbf{e})$ and $P(V | \mathbf{e})$, respectively. Note also that the join tree encodes $P(\mathbf{U}, \mathbf{e})$ instead of $P(\mathbf{U})$ (see Equation (2)).

5.4.5 Normalization

After the join tree is made consistent through global propagation, we have, for each cluster (or sepset) \mathbf{X} , $\phi_{\mathbf{X}} = P(\mathbf{X}, \mathbf{e})$, where \mathbf{e} denotes the observations incorporated into the join tree according to Section 5.4.4. When we marginalize a cluster potential $\phi_{\mathbf{X}}$ into a variable V , we obtain the probability of V and \mathbf{e} :

$$P(V, \mathbf{e}) = \sum_{\mathbf{X} \setminus \{V\}} \phi_{\mathbf{X}}.$$

Our goal is to compute $P(V | \mathbf{e})$, the probability of V given \mathbf{e} . We obtain $P(V | \mathbf{e})$ from $P(V, \mathbf{e})$ by normalizing $P(V, \mathbf{e})$ as follows:

$$P(V | \mathbf{e}) = \frac{P(V, \mathbf{e})}{P(\mathbf{e})} = \frac{P(V, \mathbf{e})}{\sum_V P(V, \mathbf{e})}. \quad (2)$$

The probability of the observations $P(e)$ is often referred to as a *normalizing constant*.

5.5 Handling Dynamic Observations

Suppose that after computing $P(V | e_1)$, we wish to compute $P(V | e_2)$, where e_2 is a different set of observations from e_1 . We could start anew by building a join tree structure, initializing its potentials, entering the new set of observations e_2 , performing global propagation, and marginalizing and normalizing. However, this amount of additional work is not necessary, because we can directly modify the join tree potentials in response to changes in the set of observations. We can imagine a dynamic system in which the consistent join tree is the steady state, and incoming observations disturb this steady state. In this subsection, we refine the control of PPTC by adding procedures to handle such dynamic observations.

Overall Control Fig. 10 shows the control for PPTC with dynamic observations. Note that there are two dotted paths going from consistent join tree to inconsistent join tree, one labeled global update and the other global retraction. Depending on how we change the set of observations, we must perform one of these two procedures. A global update is used to incorporate new observations, while a global retraction is required for modifying or retracting previous observations. Global retraction requires reinitialization of the join tree potentials, because undoing an observation involves restoring table elements that have been zeroed out by previous observations. To describe these

procedures more precisely, we first establish some basic notions of changes in observations.

Updates And Retractions To describe changes in observations, we establish the notion of an observed state. The *observed state* of a variable V is its observed value v , if V is observed; otherwise, the observed state of V is unknown, and we say that V is *unobserved*.

Suppose we change a set of observations e_1 to a different set of observations e_2 . Then the observed state of each variable V undergoes one of three changes:

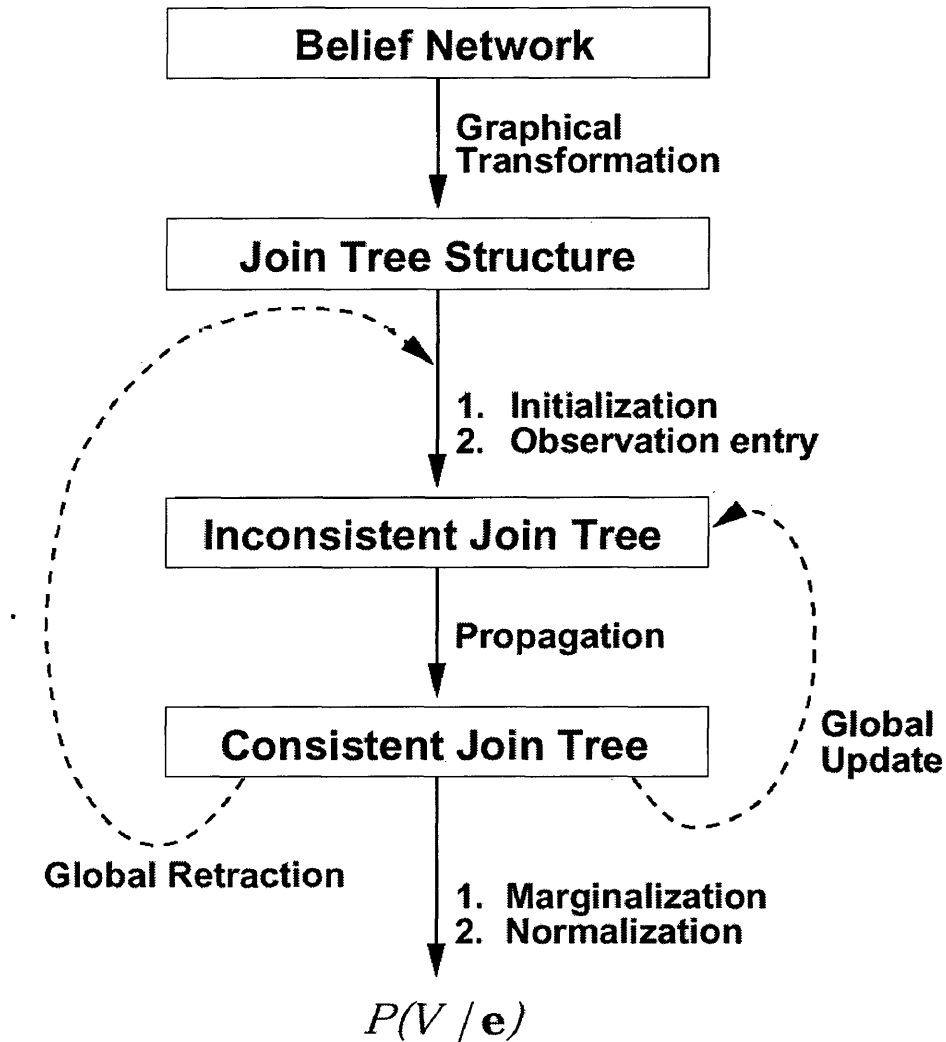


Figure 10. Block diagram of PPTC with dynamic observations.

- *No change.* If V is unobserved in e_1 , it remains unobserved in e_2 . If $V = v$ in e_1 , then $V = v$ in e_2 .
- *Update.* V is unobserved in e_1 , and $V = v$ in e_2 for some value v .
- *Retraction.* $V = v_1$ in e_1 . In e_2 , V is either unobserved, or $V = v_2$, where $v_1 \neq v_2$.

We can now state how we should handle changes in observations. Suppose we have a consistent join tree that incorporates the set of observations e_1 , and we wish to compute $P(V | e_2)$ for variables of interest V , where e_2 is different from e_1 . We incorporate e_2 into the join tree by performing one of the following:

- *Global update.* We perform a global update if, for each variable V , the observed state of V is unchanged or updated from e_1 to e_2 .
- *Global retraction.* We perform a global retraction if, for some variable V , the observed state of V is retracted.

A global update executes an observation entry (see Section 5.4.4 above) for each variable V whose observed state is updated to $V = v$. Global updating destroys the consistency of the join tree; we restore consistency by performing a global propagation. However, if the belief potential of only one cluster X is modified through global updating, then it is sufficient to unmark all clusters and call `Distribute-Evidence(X)`.

We perform a *global retraction* as follows:

1. For each variable V , update the likelihood Λ_V to reflect any changes in V 's observed state.
2. Reinitialize the join tree tables according to Section 5.4.3.
3. Incorporate each observation in e_2 according to Section 5.4.4.

We cannot handle retractions in the same way that we handle updates, because in a retraction, we are trying to recover join tree potential elements that have been zeroed out

by previous observations. Our only recourse, therefore, is to reinitialize the join tree tables and then enter the new set of observations.

5.6 Implementation notes

Implementation of PPTC algorithm consists of several classes. The main class of inference algorithm is implemented as Java application. The main data structure directed acyclic graph with further transformation to undirected, moral graph, triangulated moral graph, Clique implementation as well as sepset implementation. Application uses heap data structure for building of optimal cluster tree (secondary structure). Since application must be very flexible, application of graph data structure as an adjacency list allows achieving it.

6 Existing software and related work

There are numerous packages in existence allowing building decision support systems based on Bayesian networks.

One of the most famous and widely used is Hugin Lite 5.4 from the Research Unit of Decision Support Systems in Aalborg University (Denmark). This package is robust, has highly user-friendly interface, and allows compilation of standalone decision support applications. Inference engine of this package is highly optimized PPTC algorithm, based on scientific research of professor Finn V. Jensen and his group [2].

The edit form of the system Hugin Lite 5.3 for sample Sherlock's wet grass problem [2] is shown on fig. 1.

The screenshot shows the Hugin Lite 5.3 software interface. The title bar reads "Hugin Lite 5.3 - [c:\program files\hugin\hugin lite\samples\...". The menu bar includes File, Edit, View, Network, Table, Options, Window, and Help. Below the menu bar is a toolbar with various icons for file operations and network editing. The main window contains a variable declaration section with "Holmes" selected in a dropdown, "Labelled" in another dropdown, and "Holmes?" in a text field. Below this is a probability table for the "Holmes" node, with "Sprinkler?" and "Rain?" as parent nodes. The table shows conditional probabilities for "yes" and "no" outcomes of the parents. Below the table is a graphical representation of the Bayesian network with nodes "Rain?", "Sprinkler?", "Watson?", and "Holmes?". Arrows indicate dependencies: "Rain?" and "Sprinkler?" are parents of "Holmes?", and "Rain?" is a parent of "Watson?".

		yes		no	
		yes	no	yes	no
Rain?	yes	1	0.9	1	0
	no	0	0.1	0	1

Figure 1. Sample edit interface form of Hugin Lite 5.3 system

The system has two distinct modes – Edit and Run. In order to run the system we need to compile secondary structure for PPTC algorithm [2]. The run mode form of the system is shown on fig.2. We can easily enter evidences into left pane and propagate messages with single button click.

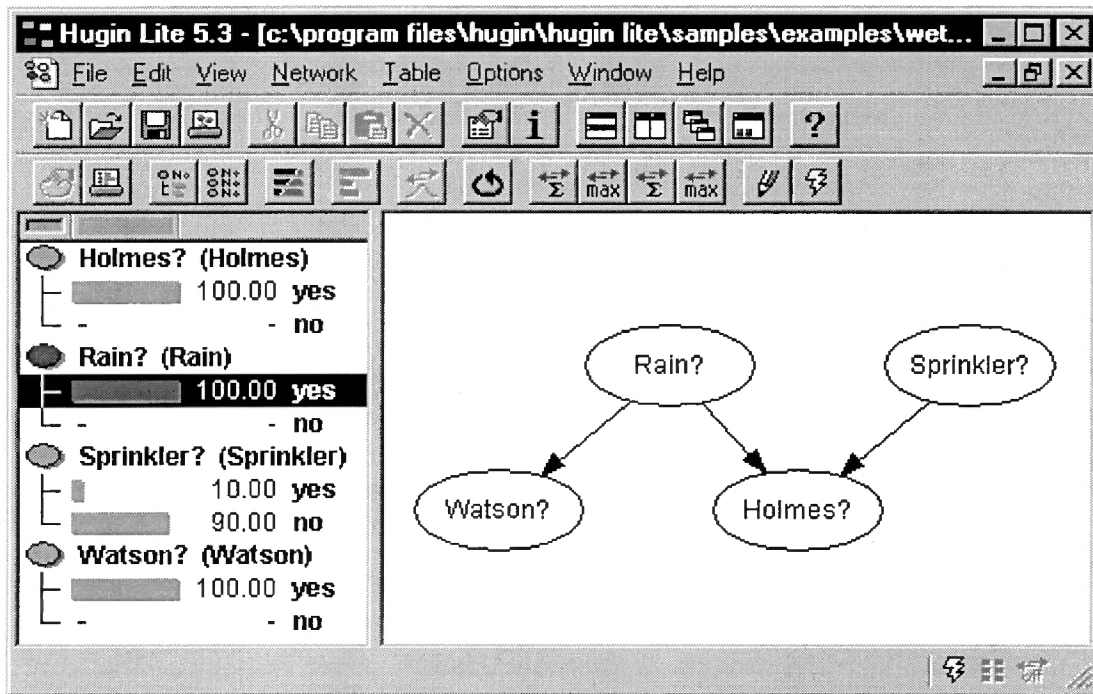


Figure 2. Run mode for the sample wet grass problem with evidential support of raining.

Hugin inference engine is highly portable. User applications could communicate with Hugin engine through special version of Hugin API – either DDE or C++ portable library. System fully supports discrete probability functions as well as continuous functions with some limitations. Software enables building influence diagrams – BN with loss discrete functions to support proper decisions.

Another software tool - Bayes Net Toolbox 2.0 for Matlab 5 is distributed and maintained for free by Ph. D. student Kevin Patrick Murphy at U.C. Berkeley. This toolbox implements both exact and approximate inference for static and dynamic BN (DBN) as well as numerous probabilistic models. The main features include:

- Node types: tabular (multinomial), Gaussian, softmax (logistic/sigmoid), noisy-or [3].
- It supports static and dynamic (temporal) BNs.
- Many different inference algorithms are supported:
 - Exact inference for static BNs:
 - junction tree[1, 2], variable elimination, naive enumeration (for discrete nets), linear algebraic methods (for Gaussian nets), Pearl's algorithm (for polytrees) [3], quickscore.
 - Approximate inference for static BNs:
 - likelihood weighting, loopy belief propagation.
 - Exact inference for DBNs:
 - junction tree, frontier algorithm, forwards-backwards, and Kalman-RTS
 - Approximate inference for DBNs:
 - Boyen-Koller, factored-frontier, loopy belief propagation.
- Parameter learning using batch (generalized) EM. A variety of regularization methods are offered. Any node can have its parameters clamped (made non-adjustable). Any set of compatible nodes can have their parameters tied (c.f.,

weight sharing in a neural net). Gaussian covariance matrices can be declared full or diagonal, and can be tied across states of their discrete parents (if any).

- Structure learning. (Currently BNT 2.0 only learns the inter-slice adjacency matrix for discrete DBNs assuming full observability.)

The source code is readable, object-oriented, and free, making it an excellent teaching and research tool. Although this toolbox has open source, it requires expensive Matlab software to run as well as lacks efficient GUI and API. This toolbox is focused mainly on advanced theoretic research and preliminary modeling of systems.

There are several BN programs, written in Java, available for free on the Internet.

JavaBayes Version 0.343 is open source package written by Fabio Gagliardi Cozman at Carnegie Mellon University. The benefit of this package is that it has open source code written in Java and distributed under GNU licence, which is significant benefit in comparison with other software products. Tool enables writing networks to the hard drive (if we run it in application mode) in different industry-wide supported formats, such as Bayesian Interchange Format v.0.1 (BIF 0.1), BIF 0.15, BUGS, XMLBIF 0.3 experimental format based on new XML specification.

The main difference of the applet from other software is that 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. Robust Bayesian inference is the calculation of bounds on posterior values given such perturbations, which is more applicable to the real world problem, many of which fit imperfectly into BN concept.

Sample wet grass problem is run again in this applet (fig.3 and fig. 4). Interface of the tool is very convenient to use. Information is displayed in two frames – one for graphical model editing and another for textual messages to user.

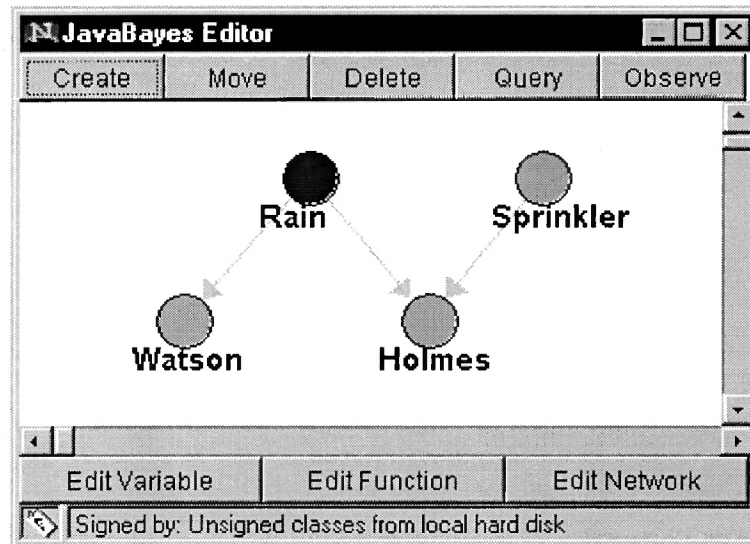


Figure 3. Sample wet grass problem in JavaBayes editor frame.

```

Posterior distribution:
probability ( "Sprinkler" ) { //1 variable(s) and 2 values
    table 0.1 0.9 ;
}

```

Figure 4. JavaBayes console frame showing posterior probability of sprinkler discrete variable given the evidential support of rain.

Network implements two exact inference algorithm for discrete nodes – variable elimination and junction tree. Network could be run in different modes – sensitivity analysis, explanation mode, posterior marginal and expectations.

Ebayes package from the same researcher consists only of highly optimized inference propagation engine written in Java, extracted from JavaBayes system. This application is capable to run on embedded platforms, such as Micro PC. Ebayes generic inference engine is intended to be an inference propagator for on-board intelligence.

CIspace is applet from Laboratory for Computational Intelligence at the University of British Columbia. This applet has a robust user interface based on VGJ graph drawing tool, although it has some unnecessary rudimentary features from this bigger tool. The real problem with this package is that Java source code is unavailable. The package could run in an Internet browser and proves to be useful in teaching of artificial intelligence. The main benefit of the tool is colored representation of posterior probability values of discrete binary variables. It significantly simplifies reading the values of variables. Textual representation of Bayesian network graphical model structure significantly simplifies use of the applet, since the hard disk is inaccessible from java applet “sandbox”. Tool is very basic and allows calculating only posterior probabilities given an observation on discrete nodes. The main form is presented on fig. 5.

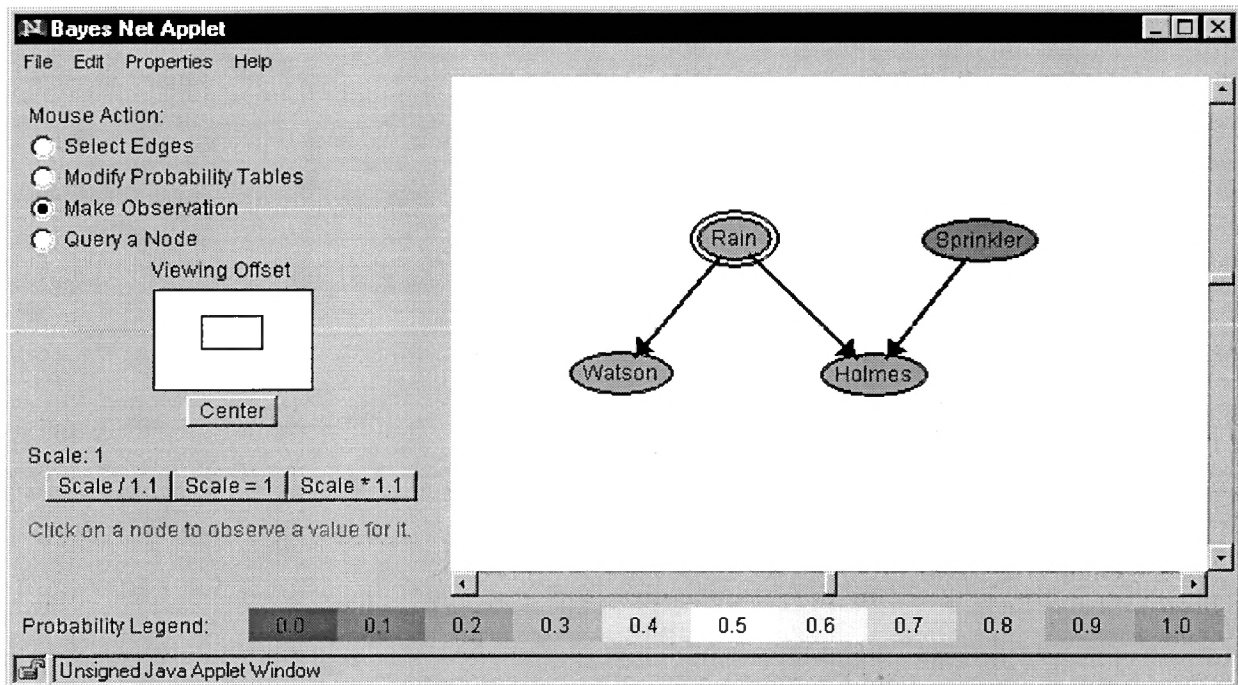


Figure 5. Bayes Net applet with wet grass example problem

7 Our solution approach

The purpose of the thesis research is to understand exactly how the inference engine works. There are many optimization techniques proposed in article on the subject [1]. Some of them, such as cluster-set mapping and evidence shrinking, were implemented in the system.

Java programming language was chosen to implement the project. The language has several significant 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 Internet-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.

Another reason for building new Java application is that many commercially available applications have proprietary inference engines. Even programs with open source code available (such as JavaBayes and Bayesian networks toolbox for Matlab)

are difficult to reuse without sound understanding of the concepts. Most of source code is insufficiently documented and is not ready to reuse in real life extended projects.

The good news is that JavaBayes application is distributed under GNU license. It opens the opportunity for reuse of the JavaBayes classes in other projects. But it still requires sound understanding of the internal inference engine structure and its interface to the other part of the program, such as GUI classes. Without apriory experience with BN architectures, its algorithms and object's interfaces it would be impossible to reuse it, even if the efficient source code is freely available.

Our final purpose is to build inference engine in Java, and having complete understanding of its interface and the crucial features to ensure its further use and reuse the other available codes in future projects. To make sure that the newly built inference engine works correctly, the comparison of processing results for several simple networks will be presented.

8 Deliverables

The deliverables of this project is Bayesian Network generic tool that could be run in applet or application mode. Tool could be used for reasoning in different real life domains.

The structure of Bayesian network application is shown on fig. 1.

Application is modularized using Java packages. Use of packages eases further software reuse and clarifies the functionality of particular class.

The packages DataStructures, Supporting and Exceptions were taken from [5]. The book [5] is common source of the most efficient and up-to-date implementation of data structures in Java.

The package Graph was designed from scratch. The main data structure of the system is Directed Acyclic Graph (DAG), implemented in class AcyclicGraph. The implementation of graph data structure is different from ordinary approach, since every node keeps adjacency Vectors of references to its parents and children in DAG structure. It allows not only to avoid using Edge class in this case, but escape from the unnecessary copying of the whole graph, proposed in [1]. AcyclicGraph class allows using the same data structure for both algorithms operating on Directed Graph, and algorithms for Undirected Graph.

JunctionTree class is inherited from AcyclicGraph. It benefits from using the methods of AcyclicGraph class and introduces new methods and Vectors member objects for the sake of efficiency operating on the secondary data structure (junction tree).

The secondary structure is directed acyclic graph as well. For the secondary structure JunctionTreeVertex class keeps Vectors of references to both parents and children, contained in the class JunctionTreeEdge. In this case JunctionTreeEdge class is used to keep references to edge destination class JunctionTreeVertex and sepset data structure [1], implemented as a class Sepsct.

Bayes package contains all necessary data structures for propagating inferences in tree of clusters. Potential class implements potential discrete function, described in chapter 5.1.2. The two the most important operations on Potential class, described in chapter 5.1.2, are multiplication and marginalization. From efficiency of these operations the overall speed of inference engine depends significantly.

The source code of multiplication method is presented in Appendix A. For the sake of efficiency this method is implemented as *static*. It takes two references to Potentials we wish to multiply. At first, method creates new Potential class with united set of discrete variables. Then it runs consistent enumeration for both classes. The main problem here was that enumeration operates on the potential index expansion that with different bases of powers. Converter method, presented in appendix C efficiently calculates the index in the table given an array of digits of heterogeneous expansion of index. An array *powers* is generated during the class initialization to avoid recalculation of expansion base powers every time we convert index. Method throws Exception if expansion digits fail the check against maxIndexes array (array of expansion bases).

Marginalization uses the same principles. This method is not static, since it marginalizes the particular Potential. First argument of the method is an array of discrete variables names we wish to marginalize out of this potential. The second argument is

reference to array of marginalization mask we need for efficient inference propagation. So, this method carries the dual role of making computationally *inefficient* marginalization and producing the marginalization mask for *efficient* future marginalization. We keep these masks in potentials of sepsets, to interface information with both Cliques efficiently [1]. Method utilizes binary masks for marginalization of consistent discrete variables. If the name of discrete variable we want to marginalize out is not present in this potential, method throws Exception which is to be caught in classes utilizing the Potential.

Clique and Sepset classes implement corresponding data structures for the tree of clusters, described in section 5.1.4.

Package Interface contains the main interface forms of the system. The Editor frame is implemented in class EditorFrame. This frame contains scroll panel inherited from Panel class and implemented as ScrollingPanel class. Scrolling panel contains NetworkPanel, which is the most important component of the interface, displaying the graphical model. Network Panel contains a bunch of eventHandlers for all mouse-generated events, making the application GUI highly interactive and flexible. Menu of Editor frame eases the choice of the currently implemented modes of applet functionality (fig.2).

There are five modes implemented: *Edit Node*, *Create Mode*, *Delete Mode*, *Move Mode*, *Edit Potential*. Create mode allows you to build the graphical model, adding nodes and connecting them by edges. *Edit Node* mode gives you an opportunity to change the name of a node as well as editing X-Y position and varying variables attribute.

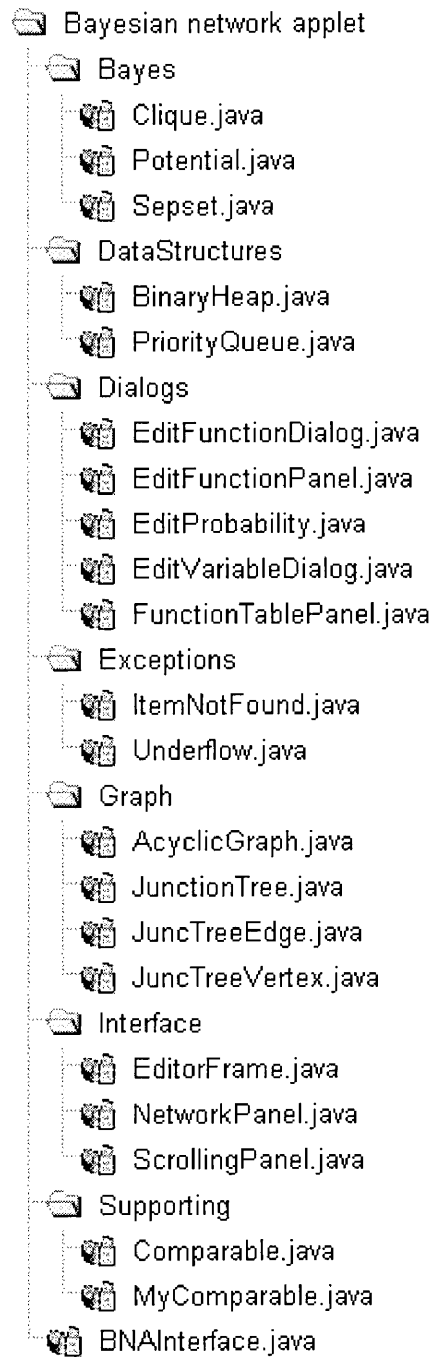


Figure 1. Structure of Bayesian network applet

Sample discrete variable editing dialog is shown on fig. 4. *Delete node* mode is useful when we want to delete node by single mouse click on the node. Node is deleted with

corresponding adjacent edges. *Move node* allows changing position of the node by drag-and-drop. *Edit potential* enables editing potential of clicked node. Edit potential dialog is shown on fig. 5. It has a quite complicated algorithm running in background, operating on heterogeneous potential's index expansion, described above. In contrast with internal complications, the dialog itself is quite simple. User just chooses the values of concrete discrete variables in potential, which results in update of the text fields for values.

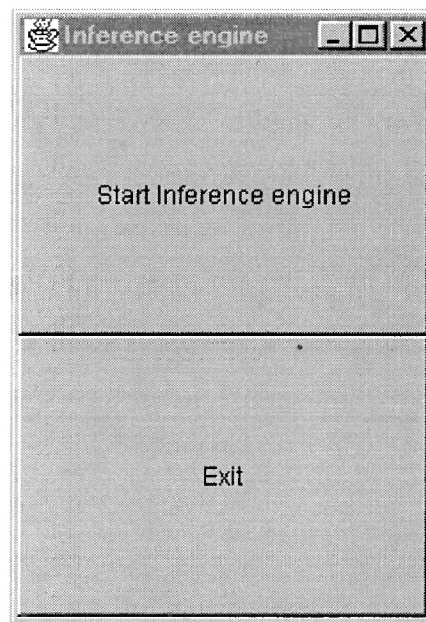


Figure 2. Frame open form

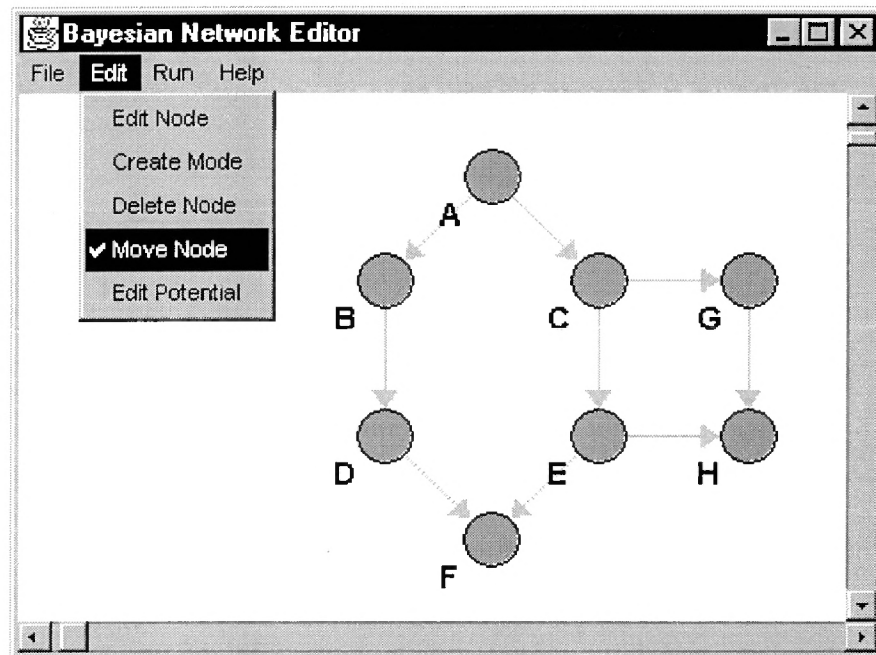


Figure 3. Inference propagation engine frame

Figure 4. Discrete variable editing

Edit Function [X]

$P(R | M, O, L, Q, N)$

Values for parents:

L	on
M	on
N	on
O	on
Q	on

on	0.99
off	0.01

Apply Dismiss

Figure 5. Discrete function editing

Appendix A

```

/**
 * Creates new potential by multiplication of two potentials.
 * Variable's names in potentials must be presorted
 */
public static final Potential multiply(Potential lhs, Potential
    rhs) throws Exception
{
    int i,j, tmpPos;
    // New String array with names of variables
    String [] newNamesTemp = new String [lhs.varNames.length +
        rhs.varNames.length];
    // New String with maximum indexes
    int [] newMaxIndexesTemp = new int [lhs.maxIndexes.length +
        rhs.maxIndexes.length];
    // Index masks for enumeration matching
    int [] lhsIndexMask = new int [lhs.maxIndexes.length];
    int [] rhsIndexMask = new int [rhs.maxIndexes.length];

    // Main loop.
    // Merge of two String arrays with elimination of repetitions.
    for(i = 0, j = 0, tmpPos = 0; i < lhs.varNames.length &&
        j < rhs.varNames.length; )
    {
        if( lhs.varNames[ i ].compareTo( rhs.varNames[ j ] ) < 0 )
        {
            lhsIndexMask[ i ] = tmpPos;
            newNamesTemp[ tmpPos ] = lhs.varNames[ i ];
            newMaxIndexesTemp [ tmpPos++ ] = lhs.maxIndexes[ i++ ];
            continue;
        }
        if( lhs.varNames[ i ].compareTo( rhs.varNames[ j ] ) > 0 )
        {
            rhsIndexMask[ j ] = tmpPos;
            newNamesTemp[ tmpPos ] = rhs.varNames[ j ];
            newMaxIndexesTemp [ tmpPos++ ] = rhs.maxIndexes[ j++ ];
            continue;
        }
        lhsIndexMask[ i ] = tmpPos;
        i++;
    }

    while( i < lhs.varNames.length ) // Copy the rest of left half
    {
        lhsIndexMask[ i ] = tmpPos;
        newNamesTemp[ tmpPos ] = lhs.varNames[ i ];
        newMaxIndexesTemp[ tmpPos++ ] = lhs.maxIndexes[ i++ ];
    }

    while( j < rhs.varNames.length ) // Copy rest of right half
    {
        rhsIndexMask[ j ] = tmpPos;

```

```

        newNamesTemp[ tmpPos ] = rhs.varNames[ j ];
        newMaxIndexesTemp [ tmpPos++ ] = rhs.maxIndexes[ j++ ];
    }

    // Memory allocation for a new string array for variables names
    // and maximum indexes
    String [] newNames = new String [tmpPos];
    int [] newMaxIndexes = new int [tmpPos];

    // This loop copies elements from temporary lists to
    // new data structure (max Indexes List and names
    // of variables).
    for (i = 0; i < tmpPos; i++)
    {
        newNames[i] = newNamesTemp[i];
        newMaxIndexes[i] = newMaxIndexesTemp[i];
    }

    // Allocation of new potential
    Potential newPotential = new Potential(newMaxIndexes,
        newNames);

    MyInteger [] enumeration = new MyInteger[tmpPos];
    // When we make enumeration of all possible values of new
    // potential.
    // These two reference arrays will help to select consistent
    // items.
    MyInteger [] lhsEnumRef = new MyInteger[lhs.maxIndexes.length];
    MyInteger [] rhsEnumRef = new MyInteger[rhs.maxIndexes.length];
    // Form the consistency references for variables in lhs
    // potential
    for(i = 0; i < tmpPos; i++)
    {
        enumeration[i] = new MyInteger(0);
    }
    // Form the consistency references for variables in lhs
    // potential
    for(i = 0; i < lhs.maxIndexes.length; i++)
    {
        lhsEnumRef[i] = enumeration[lhs.IndexMask[ i ]];
    }
    // Form the consistency references for variables in rhs
    // potential
    for(i = 0; i < rhs.maxIndexes.length; i++){
        rhsEnumRef[i] = enumeration[rhs.IndexMask[ i ]];
    }

    // This loop enumerates all possible states of
    // newly created multiplied potential.
    while(true) {
        newPotential.setPotential(enumeration,
            lhs.getPotential(lhsEnumRef) *
            rhs.getPotential(rhsEnumRef));
        // Overflow happened in the last significant digit
        if (enumerator(enumeration, newMaxIndexes) break;
    }

```

```
    }  
    return newPotential;  
}
```

Appendix B

```

/**
 * Creates new marginalized potential.
 * Names is array of vertices, saying which variables to marginalize.
 * names - variable names we want to marginalize out.
 * mapping - array of integer indexes for a fast marginalization.
 */
public Potential marginalize(String [] names, int [] [] mapping)
    throws Exception
{
    int i, j, ind;
    // Array enumerates all possible values
    MyInteger [] enumeration = new MyInteger [maxIndexes.length];
    // Array enumerates variables to be marginalized out
    MyInteger [] margEnum = new MyInteger [varNames.length -
        names.length];
    // Create new string without marginalized out arguments
    String [] margVarNames = new String [varNames.length -
        names.length];
    // Create array of maximum indexes for marginalized potential
    int [] margMaxIndex = new int [varNames.length - names.length];
    // Allocate marginalization mapping
    mapping [0] = new int [numOfEntries];

    // Generation of binary mask by comparison of variable names
    for(i = 0, j = 0, ind = 0; i < varNames.length; i++)
    {
        enumeration[i] = new MyInteger(0);
        if(ind < names.length && varNames[i].equals(names[ind]))
        {
            ind++;
        } else {
            // Corresponding list to enumerate marginalized
            // list
            margEnum[j] = enumeration[i];
            // Parameters for new marginalized list
            margVarNames[j] = varNames[i];
            margMaxIndex[j++] = maxIndexes[i];
        }
    }

    // If variables we want marginalize out are not present
    if (names.length != ind) throw new Exception( "Marginalization
        error!!!" );
    // Create new marginalized potential
    Potential margPotential = new Potential(margMaxIndex,
        margVarNames);
    // Before summation we must zero table with potentials
    margPotential.zero();

    while(true)

```

```
{
    margPotential.setPotential(margEnum,
        margPotential.getPotential(margEnum)
        + this.getPotential(enumeration));
    // Initilaize marginalization mapping
    mapping [0] [converter(enumeration)] =
        margPotential.converter(margEnum);
    // Overflow in the most significant digit causes break in
    // iteration.
    // Enumerator method enumerates all possible combinations
    // of indexing
    // values.
    if (enumerator(enumeration, maxIndexes)) break;
}

return margPotential;
}
```

Appendix C

```
/**
 * Method converts array of digits into index.
 */
private int converter(MyInteger [] indexes) throws Exception
{
    int i, ind;
    for(i = powers.length - 1, ind = 0; i >= 0; i--)
    {
        if(indexes[i].intValue() >= maxIndexes[i] ||
           indexes[i].intValue() < 0)
            throw new Exception( "Decimal converter error!!!" );
        ind += indexes[i].intValue() * powers[i];
    }
    return ind;
}
```

LITERATURE

- [1] Cecil Huang and Adnan Darwiche, Inference in Belies networks: A procedural guide. (International journal of approximate reasoning, 1994) 11:1-158
- [2] Finn V. Jensen. Bayesian Networks. New York: Springer Verlag, 1996.
- [3] Pearl, J., Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference, Second edition, Morgan Kaufmann, San Mateo, Calif., 1988.
- [4] Srinavas M. Aji and Robert J. McEliece, The generalized Distributive Law (Draft of 23 Sep 1999 11:14am).
- [5] Mark Allen Weiss, Data Structures and Algorithm Analysis in Java, Addison-Wesley, 1999