Dynamic Bayesian Networks

by

Michael C. Horsch

A Thesis Submitted in Partial Fulfillment of
the Requirements for the Degree of
Master of Science

in

The Faculty of Graduate Studies
Department of Computer Science

We accept this thesis as conforming
to the required standard

University of British Columbia
October 1990

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of COMPUTER SCIENCE

The University of British Columbia
Vancouver, Canada

Date 12 OCTOBER 1990

Abstract

Given the complexity of the domains for which we would like to use computers as reasoning engines, an automated reasoning process will often be required to perform under some state of uncertainty. Probability provides a normative theory with which uncertainty can be modelled. Without assumptions of independence from the domain, naive computations of probability are intractible. If probability theory is to be used effectively in AI applications, the independence assumptions from the domain should be represented explicitly, and used to greatest possible advantage. One such representation is a class of mathematical structures called Bayesian networks.

This thesis presents a framework for dynamically constructing and evaluating Bayesian networks. In particular, this thesis investigates the issue of representing probabilistic knowledge which has been abstracted from particular individuals to which this knowledge may apply, resulting in a simple representation language. This language makes the independence assumptions for a domain explicit.

A simple procedure is provided for building networks from knowledge expressed in this language. The mapping between the knowledge base and network created is precisely defined, so that the network always represents a consistent probability distribution.

Finally, this thesis investigates the issue of modifying the network after some evaluation has taken place, and several techniques for correcting the state of the resulting model are derived.

Contents

List of Tables

List of Figures

Acknowledgements

I am very grateful to the many people who helped and supported me during the process of researching and writing this thesis.

David Poole, my supervisor and friend, for patience, encouragement, financial support, and lots of good ideas.

David Lowe, for patience and enthusiasm during the final stages of revising.

My parents, Hartmut and Heidi, and my siblings, Monika, Steven, and Andrea. For love and support bridging the Euclidean distance which separates us.

Andrew Csinger and Manny Noik. For being invaluable colleagues, every–day heroes, occasional maniacs, and friends in the truest sense.

The gang: Norm Goldstein, Hilde Larsen, Steve and Sarah Mason, Sue Rathie, David Sidilkover. For the right things at the right times, more often than I should have hoped.

The heretics: Brad Cotterall, Ken Gehrs, Maeghan Kenney, Michelle McMaster, Jon Mikkelsen, Ray Schultz, Joeane Zadrah. For imagination and community, for helping to keep my life interesting and distinct from my occupation.

Chapter 1

Introduction

Given the complexity of the domains for which we would like to use computers as reasoning engines, and the complexity of our world in general, an automated reasoning process will often be required to perform under some state of uncertainty. For example, in applications which perform some kind of diagnosis, a single fault or abnormal behaviour may be the direct result of many factors, and determining which factor is responsible usually involves choosing a best hypothesis from amongst many alternatives.

Probabilistic analysis is one means by which the uncertainty of many interesting domains can be modelled.

Probability provides a normative theory with which uncertainty can be modelled. Without assumptions of independence from the domain, naive computations are quite intractable. For example, suppose we have five random variables, $\{A, B, C, D, E\}$ [1] and we wanted to calculate $p(A)$. Without assuming any independencies, we would have to perform the following summation:

$$p(A) = \sum_{B,C,D,E} p(A \wedge B \wedge C \wedge D \wedge E)$$

which sums $p(A \wedge B \wedge C \wedge D \wedge E)$ over every possible combination of the values taken by the random variables $\{B, C, D, E\}$, possibly an exponential number of probabilities. By exploiting independence assumptions for particular domains, we can drastically improve the complexity of such computations.

[1] I denote random variables as upper case math italics. If X can take on a set of multiple values, I will write them in lower case, as in $\{x_1, \ldots, x_n\}$. If Y is propositional, I will write $+y$ for *true* and $\neg y$ for false. More syntactic conventions will be clarified in later sections.

If probability theory is to be used effectively in AI applications, the independence assumptions from the domain should be represented explicitly, and used to greatest possible advantage. One such representation is a class of mathematical structures called Bayesian networks.

This thesis presents a framework for dynamically constructing and evaluating Bayesian networks. The remainder of this chapter will outline the basic theory of Bayesian networks, motivate the approach taken in this thesis, and outline the main issues discussed in future chapters.

1.1 Bayesian Networks

Bayesian networks are known by many names. They are also called *belief networks* and *causal probabilistic networks.* I will refer to them as Bayesian networks, emphasizing the fact that they are based on the assumptions of Bayesian probability. A related network formalism, which subsumes Bayesian networks, is known by the name *influence diagram.*

This section briefly introduces Bayesian networks; for a more thorough treatment, see Pearl [Pearl, 1988]. For a good introduction to Bayesian probability, see [Lindley, 1965].

A Bayesian network is a directed acyclic graph[2] (DAG) which represents in graphical form the independence assumptions of the probability distribution for a set of random variables. Nodes in the graph represent random variables, and the arcs connecting the nodes represent the notion of direct dependence: if a random variable A, represented in the graph by `a`, is known to be directly dependent on the random variable B, represented in the graph by node `b`, an arc is drawn from `b` to `a`. The direction of the arc is often associated with the notion of causality, directed from cause to effect.

The set of variables deemed to have direct causal or influential effect on a random variable X are called the parents, or conditioning variables, of X; these are denoted in this thesis by *parents(X).*

The strength of the direct dependency relation is quantified by a contingency table, which are a complete specification of the conditional probabilities associated with a random variable and its parents. Denoted $p(X|parents(X))$, this table specifies the effect of each combination of the possible outcomes of *parents(X)* on X.

[2] A directed acyclic graph is a graph with directed arcs such that no path following the arcs can lead from a node to itself. For introductory presentation of graph theory, see [Aho *et al.*, 1974].

The conditional independence assumptions for the domain are represented explicitly in Bayesian networks, and using these assumptions, probability calculations can be performed more efficiently than the naive computation, requiring much fewer prior probabilities.

A Bayesian network, as Pearl [Pearl, 1988] argues, is a natural, intuitive framework for writing down knowledge about a domain taking into consideration the ideas of likelihood, relevance, dependency, and causality. These ideas help to provide guidelines for creating networks for particular domains, as I will outline in Chapter 3.

A node in the network represents a random variable, which is used to represent possible outcomes, events, or states in the domain. For instance, we could represent the possibility of an earthquake happening by using a random variable, E, which takes on two values: {+earthquake, ¬earthquake}, or perhaps for simplicity, {true,false}.

The following example is borrowed from Pearl [Pearl, 1988]:

Example 1.1–1:

> Mr Holmes receives a telephone call from his neighbour, Dr Watson, who states that he hears the sound of a burglar alarm from the direction of Mr Holmes' house. While preparing to rush home , Mr Homes recalls that Dr Watson is a tasteless practical joker, and decides first to call another neighbour, Mrs Gibbons, who,despite occasional drinking problems, is far more reliable. Mr Holmes remembers reading in the instruction manual of his alarm system that the device is sensitive to earthquakes, and can be accidentally triggered by one. He realizes that if an earthquake had occurred, it surely would be on the news.

Figure 1.1 shows a a Bayesian network which explicitly represents the independence assumptions for our example. The network corresponds to the following equality:

$$\begin{aligned} & p(Alarm \wedge Burglary \wedge Earthquake \wedge Gibbons \wedge Newsreport \wedge Watson) \\ = \; & p(Alarm|Burglary \wedge Earthquake)p(Gibbons|Alarm)p(Watson|Alarm) \\ & p(Newsreport|Earthquake)p(Burglary)p(Earthquake) \end{aligned}$$

This factorization is possible because of the conditional independence assumptions for this example. Note that this equation makes no assumption about the values taken by the random variables; it is true for any of the possible combinations.

Table 1.1 shows the contingency tables required for the network in Figure 1.1.

A Bayesian network with at most one path between any two nodes (ignoring the directions of the arcs) is said to be singly connected, and possesses conditional independence properties which are exploitable computationally. Multiply–connected Bayesian networks, that

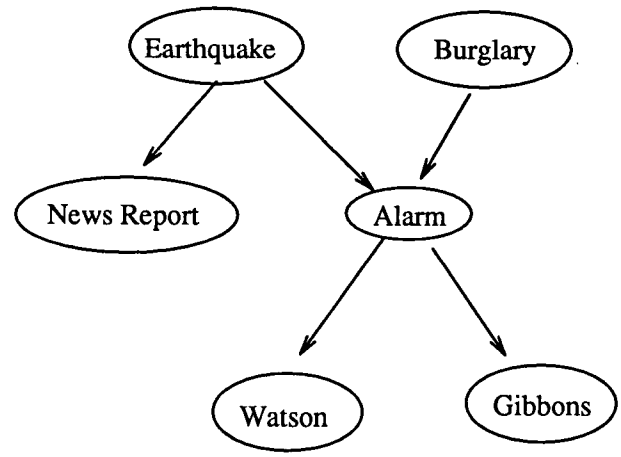

Figure 1.1: A small Bayesian network.

$p(Alarm \mid Burglary \wedge Earthquake)$		
	$+burglary$	$\neg burglary$
$+earthquake$	0.95	0.15
$\neg earthquake$	0.95	0.01

$p(Newsreport \mid Earthquake)$	
$+earthquake$	0.95
$\neg earthquake$	0.001

$p(Watson \mid Alarm)$	
$+alarm$	0.80
$\neg alarm$	0.45

$p(Gibbons \mid Alarm)$	
$+alarm$	0.60
$\neg alarm$	0.25

$p(Burglary)$	0.10

$p(Earthquake)$	0.08

Table 1.1: The contingency tables for the network in Figure 1.1.

is, Bayesian networks with multiple paths between at least one pair of nodes, are more general, and have fewer exploitable independencies. This topic will be discussed in more detail in Chapter 4.

The graphical representation is used in various ways to analyze a problem probabilistically, as Section 1.3 outlines. It is important to emphasize that the arcs in the graph are used as guidelines for computation, and without the conditional independence assumptions which are represented by the arcs in Bayesian networks, probabilistic analysis would be wildly impractical.

1.2 A dynamic approach to using Bayesian networks

1.2.1 Motivation

Bayesian networks used in typical applications represent static probability models; that is, while the networks are designed to accept conditioning based on observations, they typically do not change their structure. Consider the example in Figure 1.1. This network can only be used as a model of the particular situation for which it was created, and cannot be used, say, to tell us how likely it is that a burglary occurred given that Holmes' third neighbour Bob (who is not represented at all in the network) calls up with a report that

he hears an alarm.

In order to incorporate Bob in the model, we have to add a node representing Bob's report and an arc between the node representing the alarm and our new node.

Adding a single node and arc in this situation is conceptually a simple matter, but it required knowledge: we, the users of the system, needed to know that the independence assumption for the random variable representing Bob's report is reasonable, and that no other events of interest might cause Bob to report an alarm. As well, we needed to know how Bob's report depends on the event that Holmes' alarm sounded. That is, we needed to know the contingency table $p(Bob|Alarm)$.

This example demonstrates the kind of modification to the network structure that this thesis intends to address. However, the example is much too simple, for two reasons: First, we need no special training to gain expertise in this toy domain. Second, the example required very little expertise at building Bayesian networks, certainly no more than the intuition a novice might use, to add the new information.

For more complicated domains, a user may have trouble adding nodes and arcs to a Bayesian network because she is not an expert in the domain. However, it may be the case that we are using a Bayesian network application *precisely because* we are not experts, for consultation purposes perhaps. If, in fact, we know which new aspects of our domain ought to be incorporated in our model, but are not able to change the network suitably, then the application is useless for our situation.

There are at least three ways to address the static model problem. One way is to ask the domain expert for a revision to the network when the need arises. The second is for the domain expert to detail the domain so exactly that the need to add arcs never arises. A third way is to automate the process of building Bayesian networks so that the expert's knowledge can be accumulated in a knowledge base before hand; this generalized knowledge could then be retrieved and applied whenever a user determines the need.

This thesis examines the third approach—automating the process of building Bayesian networks from a background knowledge base of generalized probabilistic information. This approach adds flexibility to Bayesian networks applications, and tends to keep the networks required reasonably small.

1.2.2 The approach

Domain knowledge can be divided into:

individuals: entities in the domain of interest

possible properties: the properties an individual *may* have and the relationship between these properties

observed state: the properties an individual *does* have

In traditional approaches, a knowledge engineer, perhaps in consultation with a domain expert, is responsible for knowing the individuals to be included in the domain, and the properties these individuals may have. In other words, the knowledge engineer must build the network. The user is responsible for supplying information as to the actual state of the situation (i.e. the properties some of the the individuals do have). This division of knowledge is demonstrated figuratively in Figure 1.2a.

In contrast, the dynamic approach taken in this thesis divides the knowledge differently. The knowledge engineer provides a background knowledge base relating properties without specifying the particular individuals. The user supplies two kinds of information: the individuals known to be in the domain, which the dynamic system will use to build an appropriate network automatically, and the observations pertaining to the state of the model. This is figured in Figure 1.2b. In this way, the same background knowledge can be used to model different situations in which only the set of individuals under consideration changes. In the traditional approach, such changes would have to be made by the knowledge engineer.[3]

The dynamic approach provides a flexibility previously unrealized in Bayesian network applications.

1.2.3 The realization of this approach

To automate the process of building Bayesian networks, we need two steps. First, we need to be able to represent domain knowledge which is abstracted away from particular

[3]The knowledge engineer–user distinction made here is deliberately simplified. In many cases, the user of dynamic system like the one presented in this system may also be the knowledge engineer. No principle prohibits the user from modifying the knowledge base, but in doing so, the user switches roles.

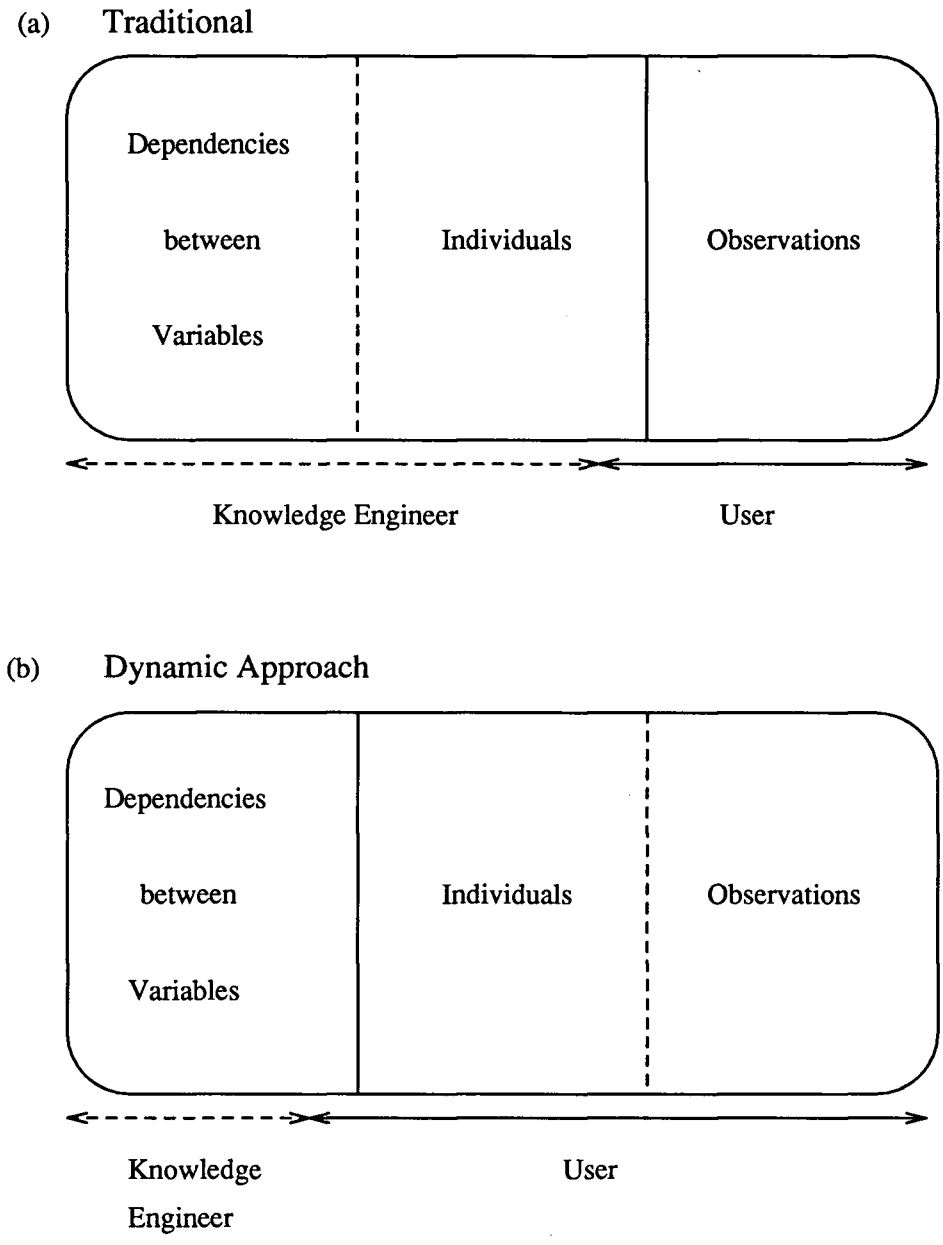

Figure 1.2: A division of probabilistic knowledge.

individuals. Second, we need to be able to build a network from this knowledge based on the information we can gain from the user.

The first step corresponds roughly to knowledge of possible properties (as in Section 1.2.2), which we elicit from the domain expert. These properties are modelled with random variables, and to abstract away from particular individuals, parameterized random variables (PRVs) are used. The knowledge engineer supplies the relationships between PRVs in the form of a background knowledge base.

The second step corresponds roughly to knowing the individuals that are to be considered in the model, and which the user can supply at the time of consultation. The system combines these two kinds of knowledge automatically, to create a Bayesian network for the situation the user has specified. The user can then consult the system, supplying the information about the observed state of the domain, and querying the network as necessary to obtain the inferred probabilistic analysis.

It may arise that, as the user is working with the system, some individuals become known after a network has already been constructed, and after observations have been submitted to the network. The user is able to supply this information to the system, and the system is able to use this information and modify the network structure appropriately.

1.3 Related work

In this section, I will provide a brief introduction to some of the related work done on Bayesian network technology. In particular, I will introduce various methods used to evaluate Bayesian networks, and review similar notions of dynamic Bayesian networks presented by Goldman and Charniak [1990], and Breese [1990].

1.3.1 Inference using Bayesian networks

There are several methods by which Bayesian networks are used to calculate probabilities. A more comprehensive discussion of much of the following will be presented in later chapters.

Pearl [Pearl, 1988] treats singly connected Bayesian networks as networks of communicating processes. The processes use arcs to propagate causal and diagnostic support values, which originate from the activation of evidence variables.

Lauritzen and Spiegelhalter [Lauritzen and Spiegelhalter, 1988] perform evidence absorption and propagation by transforming the Bayesian network into a triangulated undirected graph structure. Posterior probabilities are computed using specialized computations based on this graph.

Shachter [Shachter, 1988] uses the arcs to perform arc reversals and node reduction on the network. Evidence absorption and propagation can also be performed using arc reversals and node reductions as well [Shachter, 1989]. [4]

Poole and Neufeld [Poole and Neufeld, 1989] provide an axiomatization of probability theory in Prolog which uses the arcs of the network to calculate probabilities using "reasoning by cases" with the conditioning variables. The prototype implementation of my approach is based (but not dependent) on this work.

1.3.2 Other work on dynamic construction of Bayesian networks

During the writing of this thesis, several authors have presented similar work on building Bayesian networks from schematic knowledge collected in a knowledge base. Goldman and Charniak [Goldman and Charniak, 1990], and Breese [Breese, 1990] have developed, independently of the work presented in this thesis, similar theories of dynamic Bayesian networks. Since their work is very similar to this approach (and each other's), I will treat their work more deeply in following sections.

Laskey [Laskey, 1990] presents a framework for using dynamically created Bayesian networks in a reasoning system based on a hierarchical approach. Knowledge in the form of probabilistic schemata are used to form an *argument*, which is transformed into a Bayesian network. Conclusions or conflicting evidence which defeat the argument trigger knowledge in the schemata, forming an augmented argument. This process proceeds until no conflicting arguments are found.

A quite different approach to creating Bayesian networks involves the statistical analysis of case data. Pearl [Pearl, 1988] provides a good introduction to the process, which identifies dependencies and independencies. This area of research is current, but essentially unrelated to the methods presented in this thesis.

[4] Shachter's influence diagrams contain decision and test nodes, as well as other devices not being considered in this thesis. This thesis only looks at probabilistic nodes.

Goldman and Charniak

Goldman and Charniak [Goldman and Charniak, 1990] describe a knowledge–based approach to building networks dynamically, as part of a story understanding application. A forward–chaining inference engine takes information in the form of propositions and builds a network according to rules in the knowledge base. These rules provide ways to combine the influence of different causes on common effects by providing specialized functions for the domain and by taking advantage of the occasions in which causes interact stereotypically.

When the building process is complete, an evaluation of the network takes place, and the network is simplified by accepting as true those statements which are highly probable, and rejecting those propositions which are considered too improbable. Simplifying the network in this way may lead to additional modification of the network.

Breese

The work done by Breese [Breese, 1990] on building Bayesian networks bears many similarities to Goldman and Charniak's work. Breese's approach includes the use of such decision analytic tools as value nodes and decision nodes, which are part of the influence diagram model.

Knowledge is stored in a hybrid knowledge base of Horn clause rules, facts, and probabilistic and informational dependencies. The rules and facts are used to determine some of the probabilistic events which should be included in the model. The probabilistic dependencies are used to structure the network once it is determined which nodes ought to be included. Furthermore, these dependencies may identify other events which might play a part in the model, but are not definitely determined by the Horn clause rules.

The building process is started by the query entered by the user of the system, who also includes given information and evidence for the problem. The first step of the process is to determine the events which directly influence the query. Backward chaining through the probabilistic dependencies and the Horn clause rules identifies many nodes and arcs to be included in the model. Once this process stops, a forward chaining process identifies those events which are influenced by the nodes in the partial network from the previous step.

Discussion

There are many similarities in the approaches of Goldman and Charniak, and Breese. Both systems create Bayesian networks as a secondary reasoning tool after a primary inference engine determines which nodes to consider, and which dependencies to include in the network. In this respect, knowledge about the domain is found in two separate knowledge bases: the rules used to create the Bayesian networks, and the network itself.

In contrast, the approach taken in this thesis uses only one knowledge base, and provides only a simple process by which Bayesian networks can be created using this knowledge. This approach is much simpler, and precisely defines the mapping between the knowledge base and the networks which can be created.

Both approaches by Goldman and Charniak, and Breese could be used to implement the ideas presented in this thesis: the rule bases in both systems could be used to implement dynamic Bayesian networks as this thesis presents them. However, this thesis presents only a language for representing general probabilistic knowledge, and a simple process for creating Bayesian networks using this language; no work has been done herein on allowing another program to use this language to create networks as a sub–task.

1.4 The main contributions of this thesis

This thesis investigates the issue of representing probabilistic knowledge which has been abstracted from the knowledge of particular individuals, resulting in a simple representation language.

As well, a simple procedure for building networks from knowledge expressed in this language is provided. The mapping between the knowledge base and network created is precisely defined, so that the network always maintains a consistent probability distribution.

Finally, this thesis investigates the issue of modifying the network after some evaluation has taken place, and several techniques for correcting the state of the resulting model are derived.

1.5 An outline of this thesis

In Chapter 2, dynamic Bayesian networks are presented formally.

The major claim in this thesis is that Bayesian networks can be dynamic, flexible, and simple to use. I demonstrate the ability of my implementation in Chapter 3, thereby giving evidence for my claim. I apply my approach in such diverse areas as simple decision problems, diagnosis of multiple faults, and interpretation of sketch maps.

In Chapter 4 I briefly discuss the details of adapting the various evaluation algorithms to make use of dynamic Bayesian networks.

Finally, in Chapter 5 I conclude that the approach to Bayesian probability demonstrated here is useful in domains for which *a priori* knowledge is of a general nature, and for which specific details are most conveniently provided during the reasoning or decision–making process.

Chapter 2

Dynamic Bayesian networks

In order to make Bayesian networks versatile and reasonably manageable in terms of size, I have argued for an approach which creates networks by instantiating generalized probabilistic knowledge for the individuals known to be in the model. Thus, when information about new individuals is discovered, the system itself should be able to modify the network, by combining the new information about individuals with the knowledge found in a background knowledge base created by a domain expert.

This chapter first presents a simple and declarative language for expressing probabilistic knowledge, which I demonstrate is too unconstrained for the purpose of providing a precise framework for creating and modifying Bayesian networks dynamically. This language is restricted to remove the possibility of ambiguous knowledge, and then augmented with special purpose constructs which are based on *Canonical Models of Multicausal Interactions*, as described in Pearl [Pearl, 1988]. The resulting language is shown to be at least as expressive as Bayesian networks, and flexible enough to provide a tool for modelling many kinds of domains.

I conclude this chapter by outlining a simple procedure to create Bayesian networks from a knowledge base of statements in this language.

2.1 The background knowledge base

Dynamic Bayesian networks are created from a parameterized background knowledge base of schemata by combining it with individuals who are known to be part of the model. This

section presents the syntax of the parameterized knowledge base, and describes the process of instantiating schemata for individuals.

2.1.1 Schemata

It is necessary to represent parameterized probabilistic knowledge unambiguously, so that the intended meaning of this knowledge is apparent from the syntax. In particular, the statements in our language must clearly show how the parameters are to be instantiated, and how the resulting instantiation is used in the Bayesian network built from this knowledge base.

An *atom* is any alphanumeric string beginning with a letter. A *parameter* is represented by a capitalized atom. An *individual* is represented by a lower case atom. A random variable is an atom followed by a possible empty list of individuals; for convenience, an empty list is usually dropped from the random variable notation. Random variables take values from a finite set of discrete values called the *range*, and in many cases we use binary–valued random variables, which take {true,false} as values.

A *parameterized random variable* (PRV) abstracts the notion of a random variable away from any particular individual; it is represented as a lower case atom with a list of parameters, each of which stands in place of an individual. *Instantiating* a parameterized random variable replaces all the parameters with individuals, creating a random variable.

Associated with parameters and individuals is a corresponding *type*, which constrains the instantiation of a parameter to only individuals of the associated type.

A *schema* is the fundamental statement in this language. The syntax for a schema is as follows:

$$\begin{aligned} \mathsf{a}_1, \ldots, \mathsf{a}_n &\longrightarrow \mathsf{b} \\ &:= [v_1, \ldots, v_k] \end{aligned}$$

where b, a_i are parameterized random variables. The left hand side of the arc ($\longrightarrow$) is a conjunction of PRVs (called the *parents*), which directly influence the single PRV (called the *child*) on the right hand side. The list $[v_1, \ldots, v_k]$ is a short–hand notation which lists the probabilities for the contingency table corresponding to the schema. Recall that a contingency table must have a probability for each combination of values for the PRVs. The short–hand list orders these probabilities by varying each parent over its set of values with the left–most parent varying most quickly. This list contains 2^n probabilities when the a_i are binary–valued, and even more when any have more than two possible values.

In a collection (or knowledge base) of schemata, a PRV can occur any number of times as a parent for some other PRV, but may occur as the child in only one schema. This restriction is intended to simplify the task of writing schemata: keeping the list of parent PRVs in one schemata localizes relevant knowledge to one statement, and a knowledge base is easier to maintain.

A schema is instantiated when all PRVs in the schema have been instantiated. In the scope of a single schema, parameters shared across PRVs are instantiated with the same individual. The instantiated schema is used to build new arcs in a Bayesian network by creating a node for each random variable, and directing an arc from each parent variable to the child variable.

For example, the schema:

$$\begin{aligned} \mathsf{alarm} &\longrightarrow \mathsf{reports\_alarm(X : person)} \\ &:= [0.9, 0.01] \end{aligned}$$

might occur in a knowledge base for the alarm example, of Chapter 1. It specifies a single parent PRV, alarm, which has no parameters, and a child variable, reports_alarm(X:person), which has a single parameter, X. Only individuals of the type person can be used to instantiate this schema. In this example (as in most of the simple examples in this chapter, unless otherwise stated), the PRVs are assumed to take values from {true, false}.

This example should be interpreted as representing the idea that an alarm might be reported by a person, and explicitly assumes that only the presence or absence of an alarm sound has direct affect on the report the person might make.

The two numbers in square brackets give the contingency table for this schema, stating consisely the following two probabilities:

$$\begin{aligned} \mathsf{p(reports\_alarm(X)|\ alarm)} &= 0.9 \\ \mathsf{p(reports\_alarm(X)|\ \neg alarm)} &= 0.01 \end{aligned}$$

These numbers quantify the effect of the parent variables on the child variable. Note that the probabilities

$$\begin{aligned} \mathsf{p(\neg reports\_alarm(X)|\ alarm)} &= 0.1 \\ \mathsf{p(\neg reports\_alarm(X)|\ \neg alarm)} &= 0.99 \end{aligned}$$

can be inferred by the *Negation* axiom of probability theory.

Discussion

When instantiated, each schema of the form

$$\mathsf{a}_1, \ldots, \mathsf{a}_n \longrightarrow \mathsf{b}$$

corresponds to a conditional probability of the form $p(b|a_1, \ldots, a_n)$ Creating a Bayesian network from these schemata corresponds to building an expression for the joint probability distribution from these conditional probabilities. Thus, the intended use of these schemata is always well defined.

Example 2.1–1:
Consider the following schema:

$$\begin{aligned} \mathsf{alarm} &\longrightarrow \mathsf{reports\_alarm(X : person)} \\ &:= [0.9, 0.01] \end{aligned}$$

When instantiated with the set person = {john, mary}, the Bayesian network in Figure 2.1 is created, which represents the following expression for the joint probability distribution:

$$\begin{aligned} & p(Alarm \wedge Reports\_alarm(john) \wedge Reports\_alarm(mary)) \\ &= p(Alarm)\, p(Reports\_alarm(john)|Alarm)\, p(Reports\_alarm(mary)|Alarm) \end{aligned}$$

This simple declarative language is at least as expressive as Bayesian networks. This can be shown by taking a Bayesian network and writing it in terms of unparameterized schemata. The direct and unambiguous correspondence between unparameterized schemata and Bayesian networks is obvious. Adding parameters to the language allows us to express knowledge which can be used several times by instantiating them differently.

However, the correspondence between the knowledge base and the Bayesian networks which can be constructed from it is no longer guaranteed to be unambiguous. These ambiguities arise as a direct result of attempting to parameterize probabilistic knowledge, as we shall see in the next section, and are easily restricted.

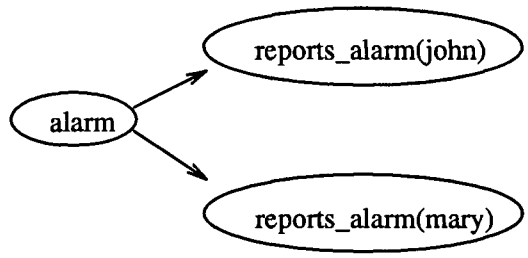

Figure 2.1: The Bayesian network created in Example 2.1–1.

2.1.2 Ambiguities in schemata

In the previous section, it was mentioned that allowing parameterized knowledge in our language could lead to ambiguous knowledge bases. In this section the possible ambiguities are illustrated and a simple constraint is imposed on the language to remove possible ambiguities.

It is useful to examine three extremes which can be attained when writing schemata in parameterized form, in order to bring to light the issues of parameterizing conditional probabilities.[1] These extremes can be labelled as:

Unique schemata: every instantiation creates a set of random variables which no other instantiation of the same schema shares

Right–multiple schemata: different instantiations may create child variables which have identical parent variables

Left–multiple schemata: different instantiations may create different parents for a single child variable

We will look at each one of these in detail in the following sections. It is worth mentioning that in general, schemata may exhibit the characteristics of several of these extremes.

[1] As well, identifying these extremes will be helpful in the discussion of Chapter 4.

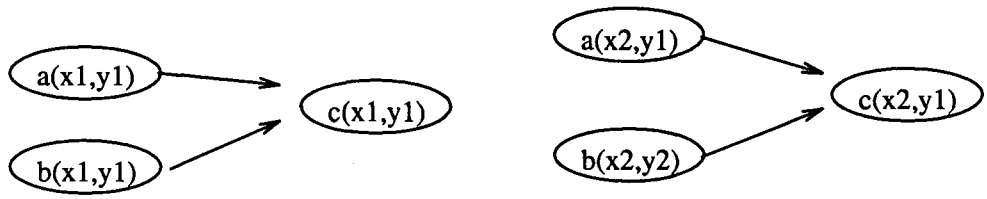

Figure 2.2: The network created by instantiating the parameterized unique schemata from Section 2.1.2.

Unique schemata

These are parameterized schemata in which every parameter in the parent variables (on the left side) also occurs in the child variable. For example:

$$\mathsf{a}(\mathsf{X},\mathsf{Y}),\mathsf{b}(\mathsf{X},\mathsf{Y}) \longrightarrow \mathsf{c}(\mathsf{X},\mathsf{Y})$$

This is called unique because every instantiation of X, Y creates a unique dependency between parents and child. If, for example, we instantiate X with $\{\mathsf{x}_1, \mathsf{x}_2\}$ and Y with $\{y_1\}$, we get two distinct sets of arcs in our network, as in the network of Figure 2.2.

There are no ambiguities which result from unique schemata.

Right–multiple schemata

These are schemata in which parameters occurring in the child variable do not appear in the parent variables, and all other parameter in the schemata occur on both sides. For example:

$$\mathsf{a},\mathsf{b} \longrightarrow \mathsf{c}(\mathsf{X},\mathsf{Y})$$

Each instantiation of X and Y results in a new child variable which may share parents with other instantiations of the schema. If we instantiate X with $\{\mathsf{x}_1, \mathsf{x}_2\}$ and Y with $\{y_1\}$, we

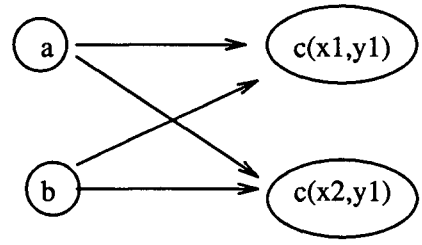

Figure 2.3: The network created by instantiating the parameterized right–multiple schema from Section 2.1.2.

obtain the dependencies shown in the network of Figure 2.3.

No ambiguities arise as a result of right–multiple schemata.

Left–multiple schemata

These are schemata in which some parameters occurring in the parent variables do not occur in the child variable. For example:

$$\mathsf{a}(\mathsf{X}) \longrightarrow \mathsf{c}$$

Each instantiation adds a parent for the child variable. For example, if X is instantiated with $\{\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n\}$, we get the dependencies shown in the network of Figure 2.4.

This kind of schema is ambiguous as there is no way to know, when the schema is written down, how many instantiations of the parent variables could be needed. However, since we must provide a contingency table which can be used in numeric calculations, we must supply a contingency table for every possible number of instantiations. This is clearly

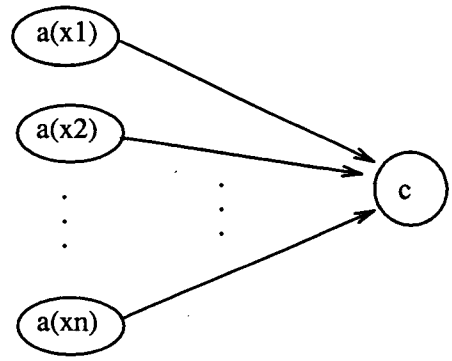

Figure 2.4: The network created by instantiating the parameterized left–multiple schema from Section 2.1.2.

impossible. Even putting a bound on the number of possible instantiations is impractical because of the fact that the size of each contingency table is exponential in the number of parent variables.[2]

Summary

This simple language of conditional dependencies is at least as expressive as Bayesian networks. Parameterizing these schemata adds flexibility to the language, but also introduces possibilities for ambiguous knowledge.

The ambiguities arising from left–multiple schemata (or hybrid schemata which have their characteristic) is an issue of pragmatics, and one possible solution is to disallow left–multiple schemata from our language. This restriction should be taken as much the same kind of restriction prohibiting left–recursive programs in Prolog, for example.

The resulting restricted language now lacks the ability to combine an arbitrary number of causes (or influences) into a single common effect. The next section presents two constructs which help to redress this deficiency without reintroducing ambiguities or requiring unreasonable numbers of contingency tables.

[2]In fact, if we were solely concerned with qualitative statements of dependency, this ambiguity would not be a problem.

2.2 Combination nodes

The main problem which arises from the ambiguity of left–multiple schemata is that the number of instantiations of the parent variables required for a particular consultation is unknown at the time the knowledge base is being created. If we assume that these instances combine their effects in the domain in a simple and regular manner, providing the required contingency table is only a matter of enumerating the instances.

In the following sections, two constructs are presented which assume a simple and regular combination of the effects of an indeterminate number of parent variables on a single child variable. Briefly, these are:

Existential Combination: if one of the parent variables takes **true**, the child variable takes the value **true**.

Universal Combination: if all of the parent variables take the value **true**, the child takes the value **true**.

These two are currently used in the language, but similar structures, exploiting some other regularities, might be added. For example, a combination mechanism could be defined corresponding to the notion that exactly one parent variable (out of many) is true. The two discussed in this section are based on the *noisy–or* and *noisy–and* gates as described in Pearl [Pearl, 1988].

2.2.1 Existential Combination

This combination is intended to be used when it is known that any one of a number of conditions is likely to affect another variable.

The syntax of this structure is as follows:

$$\begin{aligned} \exists \mathsf{X} \in \mathsf{type} \cdot \mathsf{a}(\mathsf{X}) \quad &\longrightarrow \quad \mathsf{b} \\ &:= \quad [v_1, v_2] \end{aligned}$$

where $\mathsf{a}(\mathsf{X})$, b are binary PRVs, and X is a parameter of type **type**. The variable b depends on the variable $\exists X \in type \cdot a(X)$, which is a binary PRV dependent (implicitly) on every instantiation of $\mathsf{a}(\mathsf{X})$.

The contingency table $p(b|\exists X \in type \cdot a(X))$ for this schema is given in square brackets, and is provided by the knowledge engineer in the knowledge base. The contingency table for $p(\exists X \in type \cdot a(X)|a(X))$ is automatically generated by the system and takes the form of an *Or* table based on each instantiation of a(X).

Example 2.2–1:

People who smell smoke may set off a fire alarm, and the alarm actually making a noise depends on at least one person sounding the alarm. Anyone who hears an alarm is likely to leave the building in which the alarm is found. Fire is a likely cause for someone smelling smoke.

$$\begin{aligned} \text{fire} &\longrightarrow \text{smells\_smoke(X)} \\ &:= [0.7, 0.2] \\ \text{smells\_smoke(X)} &\longrightarrow \text{sets\_off\_alarm(X)} \\ &:= [0.95, 0.01] \\ \exists \text{Y} \in \text{person} \cdot \text{sets\_off\_alarm(Y)} &\longrightarrow \text{alarm\_sounds} \\ &:= [0.98, 0.1] \\ \text{alarm\_sounds} &\longrightarrow \text{leaves\_building(Z)} \\ &:= [0.7, 0.4] \end{aligned}$$

Suppose we are given that *john* and *mary* are the only known members of the set *person*. This information combined with the above schemata creates the network shown in Figure 2.5.

The combination node combines the effects of all known members of the set. In the preceding example, it is not unreasonable to expect that some unknown person has set off the alarm, and this possibility is granted in the contingency table for $\exists Y \in person \cdot sets\_off\_alarm(Y)$:

$$\begin{aligned} \text{p(alarm\_sounds}|\ \exists \text{Y} \in \text{person} \cdot \text{sets\_off\_alarm(Y))} &= 0.98 \\ \text{p(alarm\_sounds}|\ \neg\exists \text{Y} \in \text{person} \cdot \text{sets\_off\_alarm(Y))} &= 0.1 \end{aligned}$$

The first probability indicates that with high probability, if a person sets off the alarm, the fire alarm will make a noise. The second number indicates that with low probability,

the alarm will sound when no *known* member of the set **person** has set off the alarm. This includes the possibility of a malfunction of the alarm, the possibility that some person unknown to the system has set off the alarm, as well as other unknown causes. There is no facility in our language for making hypotheses for these unknown causes.

2.2.2 Universal Combination

This combination is intended to be used when it is known that every one of a number of conditions must hold to affect another variable.

The syntax of this structure is as follows:

$$\begin{aligned} \forall \mathsf{X} \in \mathsf{type} \cdot \mathsf{a(X)} &\longrightarrow \mathsf{b} \\ &:= [u_1, u_2] \end{aligned}$$

where a(X), b are binary PRVs, and X is a parameter of type **type**. The variable b depends on the variable $\forall X \in type \cdot a(X)$, which is a binary PRV dependent (implicitly) on every instantiation of a(X).

The contingency table $p(b|\forall X \in type \cdot a(X))$ for this schema is given in square brackets, and is provided by the knowledge engineer in the knowledge base. The contingency table for $p(\forall X \in type \cdot a(X)|a(X))$ is automatically generated by the system and takes the form of an *And* table based on each instantiation of a(X).

Example 2.2–2:

A board meeting for a large corporation requires the presence of all board members, and the meeting may result in some actions, say, buying out a smaller company. A board member may attend the meeting, depending on her reliability and state of health.

$$\begin{aligned} \forall \mathsf{X} \in \mathsf{board\_members} \cdot \mathsf{present(X)} &\longrightarrow \mathsf{meeting} \\ &:= [0.8, 0.1] \\ \mathsf{meeting} &\longrightarrow \mathsf{buy\_out} \\ &:= [0.7, 0.3] \\ \mathsf{healthy(X)}, \mathsf{reliable(X)} &\longrightarrow \mathsf{present(X)} \\ &:= [0.9, 0.4, 0.6, 0.1] \end{aligned}$$

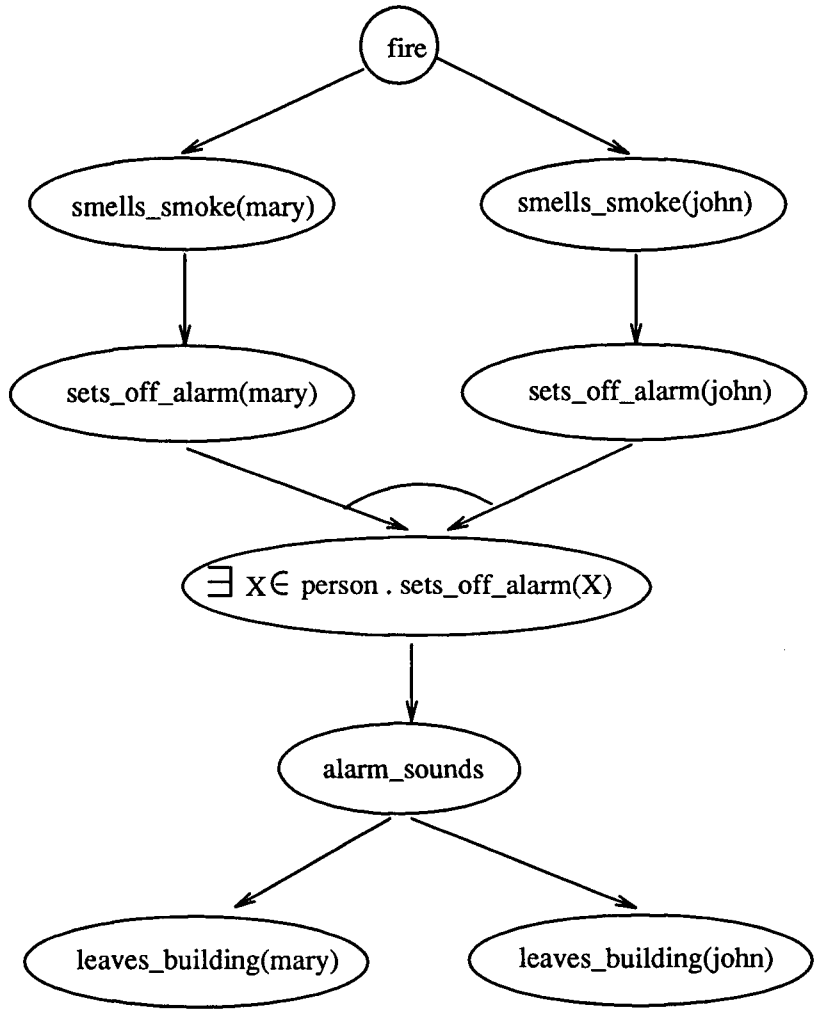

Figure 2.5: The Bayesian network created for Example 2.2–1

We note that at the time the schemata are written, it is not important to know how many board members there may be. However, we must know exactly who is a member on the board before the network can be created. This information is supplied at run–time, and the construction process can construct the appropriate network. Evidence concerning the reliability and health of any board member can then be submitted to the network, providing appropriate conditioning for queries.

2.2.3 Summary

The combination nodes presented in this section are used to allow a number of conditioning PRVs to have direct influence on a single child PRV. The number of these conditioning variables is unknown when the knowledge base is created, and the user of the system can indicate as many instantiations as required in the model. The combination nodes effectively collect each relevant instantiation, and combines them using an *or–rule* for existential nodes, and an *and–rule* for universal nodes.

These combination nodes have been implemented in the language, and another possibility would be to add a similar mechanism which performs a combination that exclusively selects a single individual as the causal agent. Others are possible in principle, but any similar mechanism should maintain the properties of being a simple and regular combination.

2.3 Creating Bayesian networks

Creating a Bayesian network from a knowledge base is a simple process. The initial state is a Bayesian network with no nodes and no arcs.

The system takes every known individual (which was supplied by the user) and instantiates every schema which has a parameter of the corresponding type. Within a schema, several parameterized random variables may have shared parameters (*i.e.*, parameters with the same name), and when the schema is instantiated, every occurance of the common parameter is replaced by a common individual.

The instantiated schema relates random variables, and is interpreted by the system, adding nodes and arcs to the current Bayesian network. The system creates a node representing each random variable in the instantiated schema, and checks whether an identical node already exists in the network. If not, a node is entered into the network; if a node already

exists in the network, it is not added again. An arc is directed from each node representing a parent random variable to the child variable. In this way, a complex structure can be created from a simple set of schemata.

This procedure for creating Bayesian networks from parameterized knowledge could be used a number of ways. First, a user might be responsible for submitting the individuals of the model to the system, and then use the Bayesian network created by the system interactively.

Alternatively, a higher level program might use the network creation/evaluation system as a sub–system, and submit individuals and make queries in order to perform some action or decision.

The `Influence` implementation, described briefly in Chapter 3 and more extensively in [Horsch, 1990], can be used as a (somewhat simple) interface between the user and the system. It can also be used as a module for another program to manipulate.

Chapter 3

Using Dynamic Bayesian Networks

In previous chapters, I presented the dynamic Bayesian network as a useful tool for reasoning under uncertainty. Chapter 2 presented some simple examples showing the results of building networks dynamically. Chapter 4 will demonstrate that the dynamic constructs are independent of the method by which the posterior distributions are calculated.

In this chapter, I demonstrate with some examples the usefulness of dynamic Bayesian networks. Since the examples are directly executable in the prototype implementation, the syntax for the `Influence` interpreter, presented in [Horsch, 1990], will be used. I will briefly introduce the syntax in the next section.

The rest of this chapter is organized in the following way. The first example takes the domain of diagnosing faults in simple electronic circuits. The second example takes the domain of interpreting simple sketch maps, which is the domain of the Mapsee project at UBC [Mulder *et al.*, 1978], and has been treated logically by Reiter and Mackworth [Reiter and Mackworth, 1989], and with default logic by Poole [Poole, 1989]. The third example is borrowed from Breese [Breese, 1990]. A summary section concludes this chapter with some observations about how knowledge bases for dynamic Bayesian networks can be created.

3.1 : An interpreter for dynamic Bayesian networks

The `Influence` interpreter is an implementation of the techniques of this thesis, written in Prolog. Writing a knowledge base using this implementation has two parts: declaring the PRVs, and writing the schemata. The knowledge base is used within the implementation to create the networks and evaluate queries.

3.1.1 Declarations

Parameterized random variables (PRVs) are represented by Prolog terms, and must be declared as being *binary*, *i.e.*, taking a value from {*true, false*}, or multi–valued, *i.e.*, taking a value from a given list of mutually exclusive and exhaustive values. The syntax is:

`binary` prv_1 / *arity*.
`multi` prv_2 / *arity* `@` [*value–list*].

where the prv_i are PRVs, and *arity* is the number of parameters the node has. For multi–valued PRVs, the list of values must be separated by commas.

Several variables, separated by commas, can be declared at once using either declaration command. In the case of multi–valued nodes, the intention is to declare several variables all taking values from the same set.

The / *arity* can be omitted in place of the direct specification of the parameters, or, if the node has no parameters, then an *arity* of zero is assumed. Within a PRV, parameters are represented by Prolog variables, *i.e.*, atoms which begin with a capital.[1]

A special declaration is required for PRVs whose dependence can be written functionally in terms of the values taken by its parents values. For example, we might want to use a PRV for the subtraction operation, such that:

$$p(diff = z | val_1 = x\, val_2 = y) = \begin{cases} 1.0 & \text{if z = x-y} \\ 0.0 & \text{otherwise} \end{cases}$$

[1]This is the first and last time that parameters will be referred to as any kind of variable. To reinforce the correct understanding, parameters for parameterized random variables in a knowledge base of schemata are implemented here as Prolog variables.

The following declaration is used:

`function` *prv*.

Any function for a PRV must be a function of the values taken by its parent's PRVs, as opposed to being a function of the individuals which may later instantiate the schema. The function itself is written with the schema in which it appears as the child. Writing schemata is the topic of the next section.

It is important to emphasize that the parameterized random variables which appear in these declarations are neither random variables nor nodes in the network. A PRV must be instantiated to become a random variable, as described in Chapter 2, and the network creating procedure will create a node in the Bayesian network corresponding to the instantiated PRV. However, the phrase *the node representing the instantiated parameterized random variable* {prv} quickly becomes tiresome, and often will be simply referred to as *node*. It should be clear by context whether the strict meaning of node is intended.

3.1.2 Schemata

Once the PRVs for a knowledge base are declared, the schemata may be written. The syntax is simply:

$$p\text{–}prv_1\text{ , }\ldots\text{ , }p\text{–}prv_n\ \texttt{=>}\ c\text{–}prv$$
$$\texttt{:=}\ [p_1,\ldots,p_k]\,.$$

which is interpreted as indicating that instantiations of this schema will be used to create a Bayesian network such that arcs will be directed from the nodes representing the parent PRVs to the node representing the child PRV. A restriction on schemata is that every parameter which appears in the list of parent PRVs must also appear in the child PRV; in the terminology of Chapter 2, schemata are not allowed to be left–multiple. The list [$p_1,\ldots,p_k$] is a short–hand notation which lists the probabilities for the contingency table corresponding to the schema. Recall that a contingency table must have a probability for each combination of values for the PRVs. The short–hand list orders these probabilities by varying each parent over its set of values (in the order of the value list given in the `multi` declaration) with the left–most parent varying most quickly. If the child PRV is declared as `binary`, only the case in which the child PRV takes the value `true` is specified. Otherwise, the parent PRVs are varied for each value of the child PRV.

For PRVs declared as `function`, the syntax is:

$p\text{-}prv_1$, ..., $p\text{-}prv_n$ => $c\text{-}prv$
:= {p($c\text{-}prv$=$c\text{-}val$, [$p\text{-}prv_1$=p_1-val, ..., $p\text{-}prv_n$=p_n-val])
= if *test* then v_{true} else v_{false} }.

where *test* is a Prolog condition, v_{true} and v_{false} are numbers in $[0, 1]$. The

Example 3.1–1:

Suppose we wanted to model a decision for someone to attend a baseball game. Assume that a baseball fan will almost certainly have lots of fun at the game if it doesn't rain, but may find something fun to do elsewhere. The more rain at the game, the less likely it will be that the fan has a lot of fun.

```
binary attends(X).
multi rains @ [none,drizzle,pours].
multi has-fun(X) @ [lots,little].

attends(X:fan), rains => has-fun(X:fan)
        := [0.95, 0.5, 0.65, 0.5, 0.2, 0.5,
            0.05, 0.5, 0.35, 0.5, 0.8, 0.5].
```

The ordering of the contingency table (the list `[0.95, ..., 0.5]`) is as follows:

$$
\begin{aligned}
p(has-fun(X) = lots | attends(X) \wedge rains = none) &= 0.95 \\
p(has-fun(X) = lots | \neg attends(X) \wedge rains = none) &= 0.5 \\
p(has-fun(X) = lots | attends(X) \wedge rains = drizzle) &= 0.65 \\
p(has-fun(X) = lots | \neg attends(X) \wedge rains = drizzle) &= 0.5 \\
p(has-fun(X) = lots | attends(X) \wedge rains = pours) &= 0.2 \\
p(has-fun(X) = lots | \neg attends(X) \wedge rains = pours) &= 0.5 \\
p(has-fun(X) = little | attends(X) \wedge rains = none) &= 0.05 \\
p(has-fun(X) = little | \neg attends(X) \wedge rains = none) &= 0.5 \\
p(has-fun(X) = little | attends(X) \wedge rains = drizzle) &= 0.35 \\
p(has-fun(X) = little | \neg attends(X) \wedge rains = drizzle) &= 0.5 \\
p(has-fun(X) = little | attends(X) \wedge rains = pours) &= 0.8
\end{aligned}
$$

$$p(has - fun(X) = little|\neg attends(X) \wedge rains = pours) \quad = \quad 0.5$$

As an example of using `function` declarations, consider the following example, which models the probability of taking the sum of two six–sided dice:

```
multi d1,d2 @ [1,2,3,4,5,6].
multi sum @ [2,3,4,5,6,7,8,9,10,11,12].

d1, d2 => sum
    := { p(sum=Z, [d1=X, d2=Y])
        = if (Z is X + Y) then 1.0 else 0.0}.
```

3.1.3 Combination Nodes

Existential and universal combination PRVs must be declared using the reserved PRV names `exists` and `forall`:

```
binary exists(param, type, prv).
binary forall(param, type, prv).
```

where *param* is a parameter which occurs in the PRV *prv* and is of type *type*. Declared combination PRVs can be used in schemata as follows:

```
exists(param, type, p-prv) => c-prv
                := [v1, v2].
```

Note that a combination PRV must be the only parent of a child PRV, and is necessarily a `binary` PRV.

3.1.4 Creating networks

Given a set of schemata, the interpreter needs to be given individuals to create a network. An individual can be specified by the following:

```
observe type(indiv, type).
```

where *indiv* is the name of an individual, and *type* is the individual's type.

3.1.5 Queries

Querying the Bayesian network consists of asking the interpreter to compute the probability of a conjunction of random variables given a conjunction of conditioning knowledge. The syntax is:

```
p([query-conjunction],[conditioning-knowledge]).
```

The interpreter will return the determined value, repeating the query information. This part of the interpreter uses the Bayesian network evaluation techniques of Poole and Neufeld [Poole and Neufeld, 1989] modified for dynamic Bayesian networks as outlined in Chapter 4.

3.2 Diagnosis of multiple faults for digital circuits

In this section, I present a simple `Influence` knowledge base which can be used to create a probability model for any digital circuit containing and–gates, or–gates, and xor–gates. This is a simplification of a common diagnostic domain [Reiter, 1987, de Kleer and Williams, 1987, Pearl, 1988], but the purpose of this example is not to demonstrate a new diagnostic technique, but to show how Bayesian networks can be built dynamically based on a carefully designed knowledge base of schemata.

Assume that digital gates exhibit random but non–intermittent behaviour when they are broken, and that they have a prior probability of being broken. A gate that is working correctly has an output that is dependent directly on the inputs to the gate. As well, the behaviour of a gate is the only criterion for diagnosis.

A gate is assumed to have two inputs, and to distinguish them, we will label them with input port numbers, as in `input(and\_gate(g157),1)`.

First we define the variables used in our domain, indicating the parameter types used:[2]

[2]Because this example is useful pedagogically, it is broken into several pieces, with discussion intermingled with code. The complete source is found in Appendix A.

```
binary ok(Gate:gate).
binary output(Gate:gate).
binary input(Gate:gate, Port:port).
binary exists(conn(G1, G2, Port), connections, output(G1)).
```

The node `ok(Gate)` has the value true iff the gate is functional. The node `output(Gate)` is true iff the output of the gate is on, and similarly `input(Gate, Port)` is true iff the input to the indicated port of a gate is on. The existential node `exists(conn(G1, G2, Port), connections, output(G1))` tells us there is a connection between the output of a gate and an input port to another gate. We use this parameter `conn(G1, G2, Port)` to allow us to submit observations of the network structure to the system, allowing it to create the corresponding network.

The behaviour of the gates is modelled with the following schemata:

```
/* and--gates */
ok(and_gate(G):gates),
input(and_gate(G):gates,1),
input(and_gate(G):gates,2) => output(and_gate(G):gates)
                := [1,0,0,0,0.5,0.5,0.5,0.5].

/* or--gates */
ok(or_gate(G):gates),
input(or_gate(G):gates,1),
input(or_gate(G):gates,2) => output(or_gate(G):gates)
                := [1,1,1,0,0.5,0.5,0.5,0.5].

/* xor--gates */
ok(xor_gate(G):gates),
input(xor_gate(G):gates,1),
input(xor_gate(G):gates,2) => output(xor_gate(G):gates)
                := [0,1,1,0,0.5,0.5,0.5,0.5].
```

which states the assumptions about gates. The first four entries in the contingency table for each gate is simply the truth table for the boolean expression associated with each type of gate. The last four entries, all 0.5, merely makes explicit the assumption that if the gate

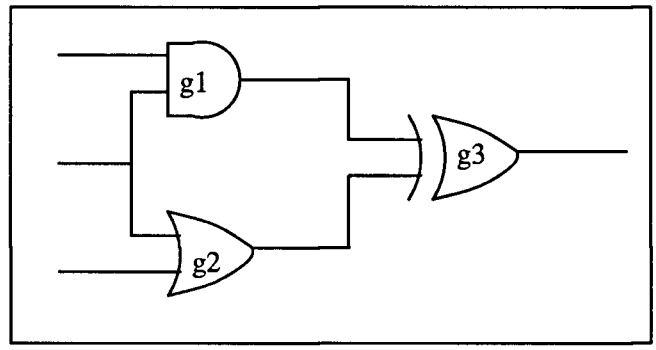

Figure 3.1: A simple circuit.

is not working correctly due to a malfunction, the output of the gate will be *on* or *off* with equal probability.

The topological connections between the gates is modelled using an existential combination:

```
exists(conn(G1,G2,Port), connections, output(G1))
            => input(G2:gates,Port:ports).
                 := [1,0].
```

This schema has several interesting features. First, the use of the connection triple ensures that the output of the gate G1 is associated with the input of the gate G2 according to the observations to be supplied by the user. Furthermore, every input to a gate will be associated with unique value—there will be no more than one value coming into the input, and only a very trivial combination (of one value) will be performed. This technique is discussed further in Section 3.5

Note that by setting the contingency table differently, we can use the same schema to model the circuit under the assumption that the connections are not always faultless. For instance, using the following table `[0.85,0]` says that the connection has a finite probability of being broken.

Figure 3.1 shows a very simple circuit. Gate `g1` is an and–gate, `g2` is an or–gate, and `g3`

is an xor–gate. The knowledge base will be able to model this circuit if we supply the following information about types:

```
observe type(and_gate(g1),gates).
observe type(or_gate(g2),gates).
observe type(xor_gate(g3),gates).
observe type(conn(and_gate(g1),xor_gate(g3),1), connections).
observe type(conn(or_gate(g2),xor_gate(g3),2), connections).
```

This information creates the Bayesian network shown in Figure 3.2, which we can use for a variety of tasks. We can model the correct behaviour of the circuit by conditioning our queries on the assumptions that every known gate is working correctly. We can also condition a query on the known inputs and outputs to determine the probability of a particular configuration of gates working incorrectly.

3.3 Interpreting sketch maps

The domain of interpreting sketch maps provides a good test–bed for knowledge representation techniques and automated reasoning tools. The Mapsee project at the University of British Columbia has used a variety of hierarchical knowledge structures and constraint satisfaction techniques to interpret hand drawn sketch maps (for a good overview, see [Mulder *et al.*, 1978]). To formalize the image interpretation problem, Reiter and Mackworth [Reiter and Mackworth, 1989] posed the problem within a logical representation, showing that there is a notion of correct interpretations which all applications must provide (but which is much harder to prove for less rigorously well–defined programs).

This problem is of interest as an example of the application of Dynamic Bayesian networks because of the following two questions:

1. Given several possible interpretations for objects in a sketch map, how can we choose a preferred one?
2. How can we provide a Bayesian network which can model any possible sketch map we may want to interpret?

The answer to the first question is: *Use a Bayesian network.* The answer to the second is: *Use a dynamic Bayesian network.*

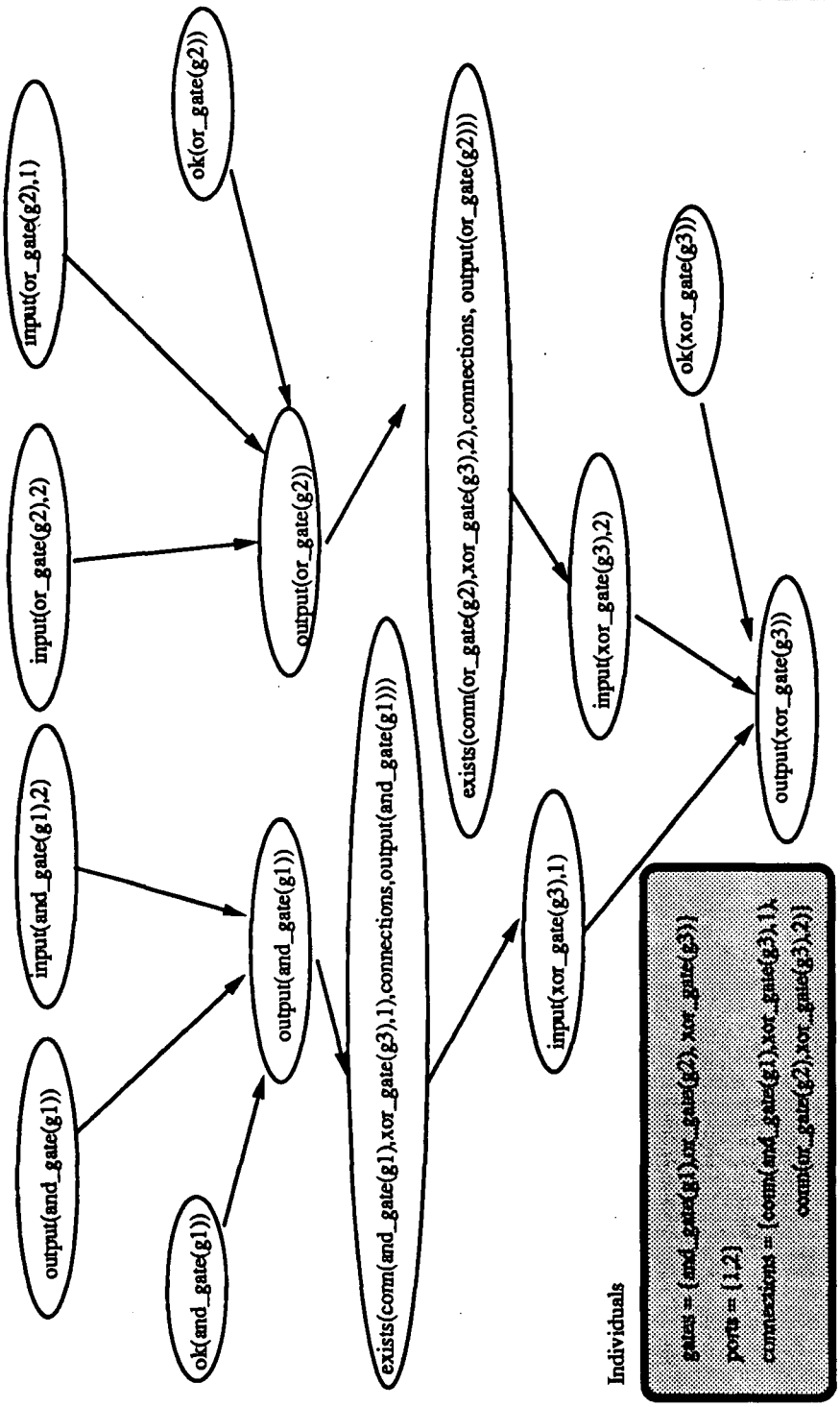

Figure 3.2: The Bayesian network constructed by our simple knowledge base for the circuit in Figure 3.1.

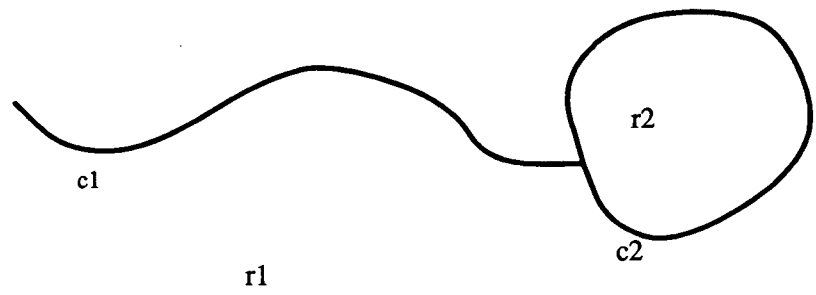

Figure 3.3: A simple sketch map.

3.3.1 Sketch maps

A sketch map is made up of chains[3] and regions, with the intention that, interpreted correctly, chains correspond to roads, rivers or shorelines, and regions correspond to either land or water. Figure 3.3 shows an example of a simple sketch map. This example is ambiguous, as chain c1 could be interpreted as either a road which meets a traffic loop (chain c2), or a river which flows into a body of water, region r2 (chain c2 being the shoreline), or a road which ends at a body of water.

3.3.2 A probabilistic knowledge base

The following schemata represents enough knowledge to provide answers to queries about sketch maps, like "What is the probability that chain c2 is a road, given that it is a chain which meets another chain?" The contingency tables use probabilities of zero for impossible configurations, such as two rivers crossing, etc. For knowledge like "Given two chains which meet in a tee, the probability that they are both rivers is p" the number p used is only a rough estimation, which is sufficient to demonstrate the dynamic qualities of schemata, but may not reflect the reality of occurrences of rivers or roads joining.

[3] A chain is a connected sequence of points.

```
%% scene objects can be linear or area objects
multi isa-linear(X) @ [road, river, shore].
multi isa-area(X) @ [land, water].

%% Image objects:
binary tee(X,Y). %% chain X meets chain Y in a tee-shape
binary chi(X,Y). %% chain X and chain Y make a chi--crossing
binary bounds(X,Y). %% chain X bounds area Y
binary interior(X,Y). %% Area Y is in the interior of chain X
binary exterior(X,Y). %% Area Y is exterior of chain X

%% Scene descriptions:
binary joins(X,Y). %%linear scene objects join in a tee shape
binary crosses(X,Y). %% linear scene objects cross
binary inside(X,Y). %% area object X is inside linear object X
binary outside(X,Y). %% area object X is outside linear object X

%% Trick implementation of equality predicate!
%% Two distinct random variables, which have their
%% arguments as values:
multi val1(X) @ [X, X].
multi val2(Y) @ [Y, Y].

%% equality is then define if the values are the same!
function equal/2.
val1(X), val2(Y) => equal(X,Y)
        {p(equal(X,Y),[val1(X)=X,val2(Y)=Y])
            = if (X=Y) then 1.0 else 0.0}.

%% Two linear scene objects can join to form a tee
isa-linear(X:chain), isa-linear(Y:chain),
joins(X:chain,Y:chain), equal(X:chain,Y:chain)
        => tee(X:chain,Y:chain)
        := [...].[4]
```

[4]There are 36 numbers in this list, and in some of the contingency tables for the remaining schemata there are even more than this, but none of them is crucial to the explanation of the dynamic nature of

```
%% two linear objects can cross to form a chi
isa-linear(X:chain), isa-linear(Y:chain),
crosses(X:chain,Y:chain), equal(X:chain,Y:chain)
        => chi(X:chain,Y:chain)
        := [...].

%% linear objects can form closed loops
isa-linear(X:chain), loop(X:chain) => closed(X:chain)
        := [...].

%% linear scene object are found beside area objects
isa-area(X:region), isa-linear(Y:chain),
beside(X:region,Y:chain) => bounds(X:region,Y:chain).
        := [...].

%% on the linear object which forms the boundary between two
%% area objects, we must constrain the objects involved
%% e.g.  A road is not a boundary between two bodies of water
isa-area(X:region), isa-linear(Y:chain), isa-area(Z:region),
inside(X:region,Y:chain), outside(Z:region,Y:chain)
        => boundary-constraint(X:region,Y:chain,Z:region).
        := [...].
```

There are several interesting features of this knowledge base. First of all, the use of multiple valued PRVs constrains the objects in the domain, *e.g.*, a linear scene object can only be one of {*river, road, shore*}, and multi-valued PRVs express this succinctly. Observe also the use of type information; since only individuals of the correct type can be used to instantiate schemata, dividing the domain into two PRVs `isa-linear(X)` and `isa-area(X)` assures that chains and regions are disjoint sets of objects. Finally, the use of `equals(X,Y)` is required to ensure that when the schemata are instantiated we only consider interpretations in which, for example, linear scene objects do not intersect themselves.

---

this example. For a complete listing, see Appendix A.

The user specifies the following individuals, and their types:

```
type(c1,chain).
type(c2,chain).
type(r1,region).
type(r2,region).
```

which when combined with the knowledge base, creates the Bayesian network in Figure 3.4. The user is free to query the state of the network, conditioning on c2 being closed, and r1 being on the outside of c2, etc. Note that this network could be used the opposite direction as well: conditioning on the scene objects allows the user to "predict" the image.

3.4 The Burglary of the House of Holmes

The following story is used by Breese [Breese, 1990] to demonstrate his network construction application, and is derived from an example used by Pearl [Pearl, 1988]:

> Mr. Holmes receives a telephone call from his neighbour, Dr. Watson, who states that he hears the sound of a burglar alarm from the direction of Holmes' home. As he is preparing to rush home, Mr. Holmes reconsiders his hasty decision. He recalls that today is April 1st, and in light of the April Fool's prank he perpetrated on his neighbour last year, he reconsiders the nature of the phone call. He also recalls that the last time the alarm sounded, it had been triggered by an earthquake. If an earthquake has occurred, it will surely be reported on the radio, so he turns on a radio. He also realizes that in the event of a burglary, the likelihood of recovery of stolen goods is much higher if the crime is reported immediately. It is therefore important, if he in fact a burglary did occur, to get home as soon as possible. On the other hand, if he rushes home he will miss an important sales meeting with clients from Big Corp. which could result in a major commission. Should Mr. Holmes rush home?

The knowledge needed to solve this problem can be expressed in the language of `Influence` as follows.

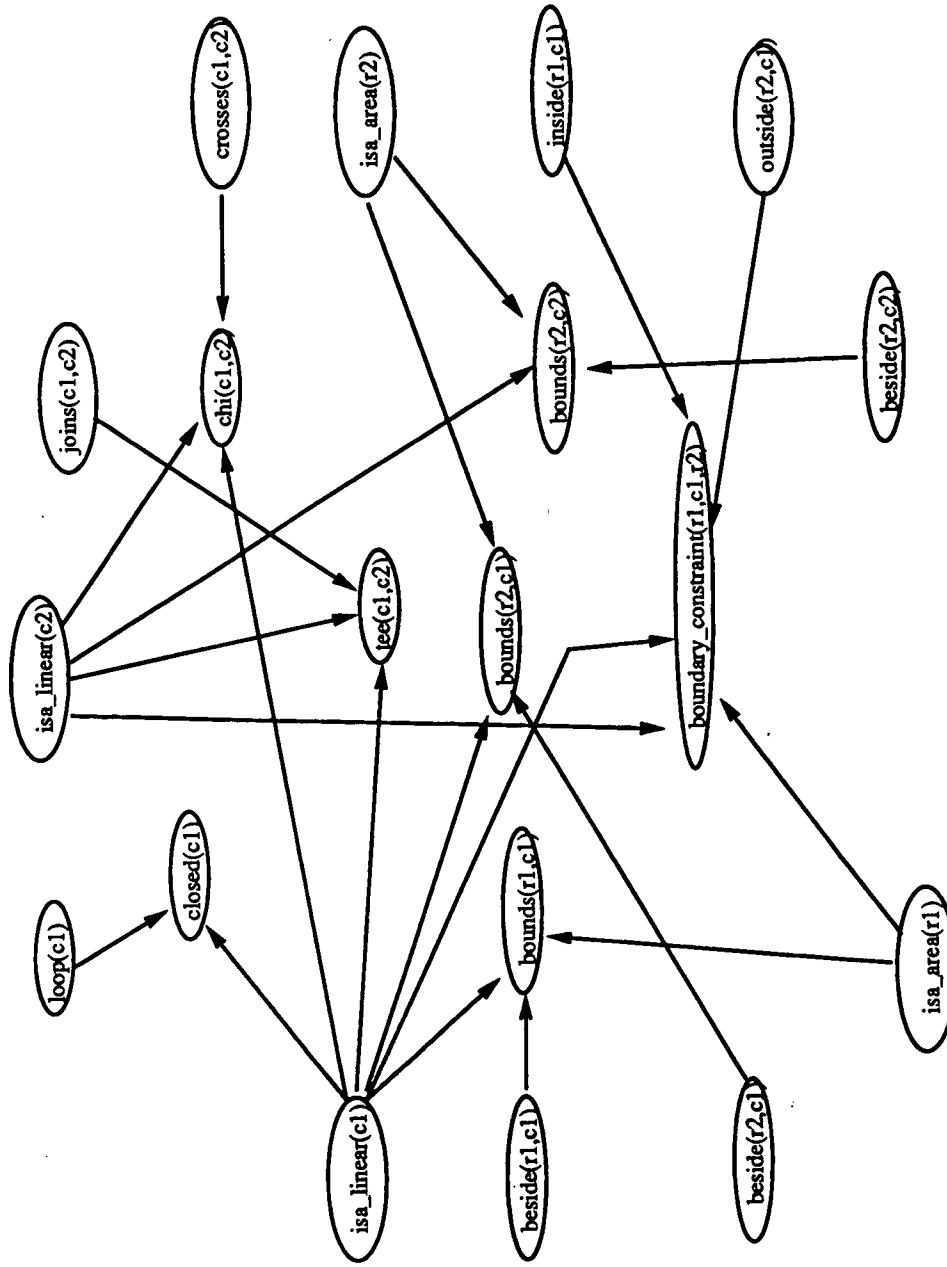

Figure 3.4: The Bayesian network constructed by our simple knowledge base for the sketch map in Figure 3.3.

```
binary sale-obtained(X). %% has a sale been obtained by X?
binary burglary(X). %% was there a burglary in X's house?
binary go-home(X). %% does (should) X go home IMMEDIATELY?
binary meeting(X,Y). %% was there a meeting between X and Y?
binary earthquake.  %% was there an earthquake?
binary radio.  %% did the radio report an earthquake ?
binary client(X). %% X has a client?

function value(X). %% what is the value of X's decision?
function losses(X). %% how much did X lose?
function income(X). %% how much does X stand to gain?

%% how much is a sale worth?
multi sale-value(X) @ [0,150,300].

%% how much of X's stolen goods were recovered?
multi goods-recovered(X) @ [0,500,5000].

%% how much were X's stolen goods worth?
multi stolen-goods-value(X) @ [0,500,5000].

%% when did X report thre burglary?
multi burglary-report(X) @ [immediate, late, none].


%% compute the difference between X's income and losses
%% call it value for X
losses(X:alarm-owner), income(X:alarm-owner)
      => value(X:alarm-owner)
          := {...}.[5]


%% X may recover some of his stolen goods depending on
%% when X reported the burglary
```

[5]The function here, as well as the contingency table for the schemata in this example are not essential for the understanding of the dynamic nature of the schemata. The actual numbers are found in the complete listing in Appendix A.

```
burglary(X:alarm-owner), burglary-report(X:alarm-owner)
        => goods-recovered(X:alarm-owner)
        := [...].

%% If X recovers stolen goods, then there is not loss
goods-recovered(X:alarm-owner),
stolen-goods-value(X:alarm-owner) => losses(X:alarm-owner)
            := {...}.

%% the income for Y depends on getting the sale from a client
client(Y:alarm-owner), sale-obtained(Y:alarm-owner),
sale-value(Y:alarm-owner) => income(Y:alarm-owner).
            := {...}.

%% - A meeting may take place between X and Y
%% if X doesn't go home
go-home(X:alarm-owner) => meeting(X:alarm-owner,Y)
        := [...].

%% X will report a burglary if one occurred, and X has gone home
%% to verify it go-home(X:alarm-owner), burglary(X:alarm-owner)
        => burglary-report(X:alarm-owner)
        := [...].

%% if there was a burglary, then goods of some value have been
%% stolen
burglary(Y:alarm-owner) => stolen-goods-value(Y:alarm-owner)
        := [...].

%% X will meet with Y to talk sales.  A sale may occur
meeting(X:alarm-owner,Y:corporation)
        => sale-obtained(X:alarm-owner,Y:corporation)
        := [...].

%% an alarm is affected by earthquakes and burglaries
%% - the alarm will almost definitely sound if both an earthquake
%% and a burglary occur, and almost definitely will not sound if
%% neither occur
```

```
burglary(Y:alarm-owner), earthquake => alarm(Y:alarm-owner)
        := [...].

%% earthquakes tend to be reported on the radio...
earthquake => radio
        := [...].

%% Phone calls from neighbours about alarms.
%% - neighbours usually only call when an alarm sounds
%% - non-neighbours don't call at all!
neighbour(X:person,Y:alarm-owner),
    alarm(Y:alarm-owner) => phone-call(X:person,Y:alarm-owner)
        := [...].
```

This example demonstrates the power of our simple network creating process. The user specifies the appropriate individuals:

```
observe type(holmes,alarm-owner).
observe type(watson,person).
observe type(big-corp, corporation).
```

and the network in Figure 3.5 is created.

In [Breese, 1990], Breese makes use of influence diagram technology, such as decision nodes, which are useful in this problem. These types of nodes are not implemented in `Influence`. However, by conditioning on whether or not Holmes should go home, the user can determine the value of each decision by querying `value(holmes)`.

The value computed by this technique is not the same as the "value of a decision node" in Breese's program; Breese uses value nodes from influence diagram technology (as in [Shachter, 1986]) which computes an expected value based on the product of the possible values by their respective probability. In `Influence` there is no equivalent functionality built–in.[6]

[6] However, this computation can be simulated by querying each value of the node to obtain the posterior probability and performing the same sum–of–products by hand.

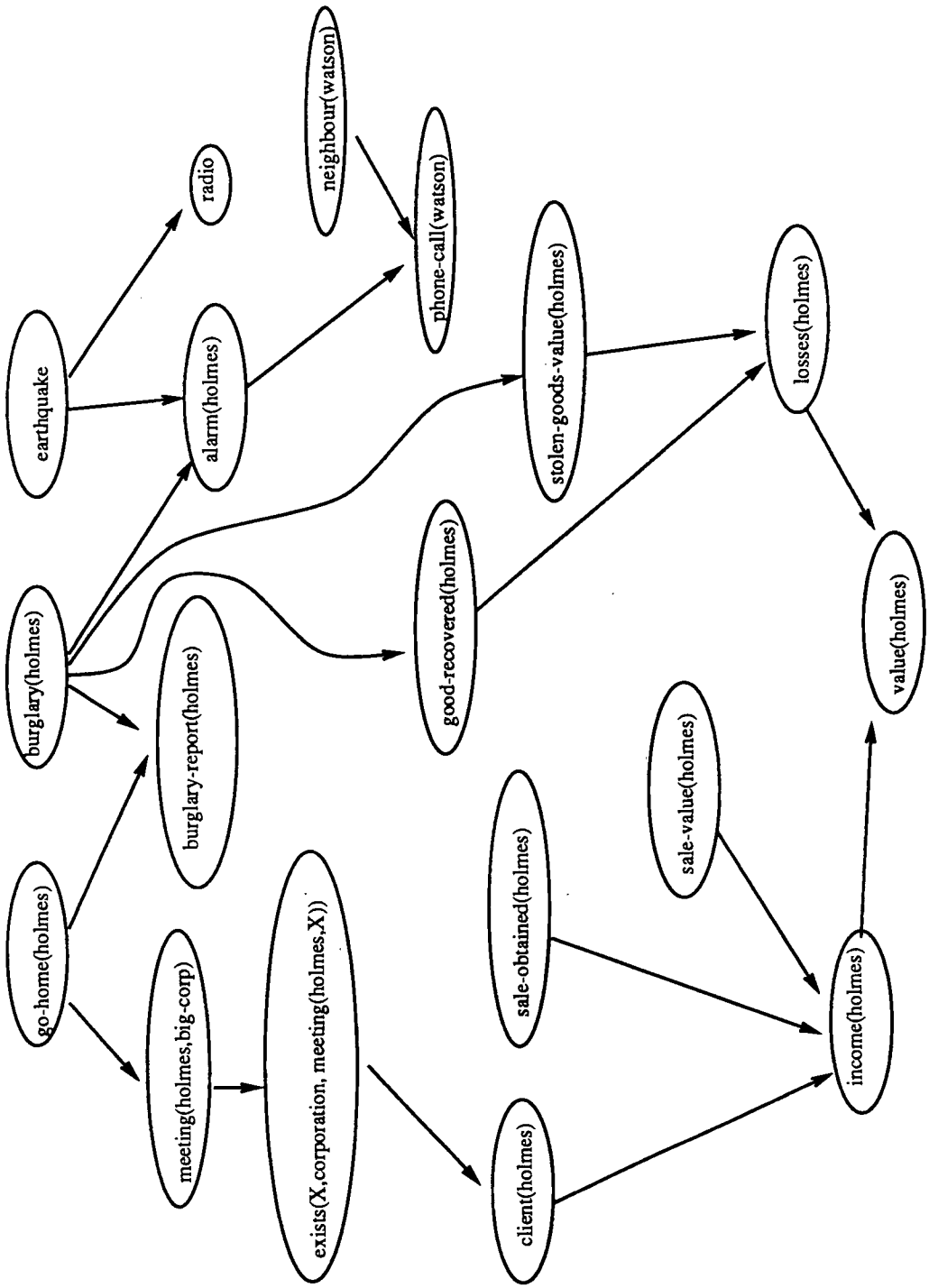

Figure 3.5: The network created for Holmes' decision.

3.5 Building knowledge bases for Dynamic Bayesian networks

In this chapter several example knowledge bases have been presented for different domains. As it may not be obvious how these knowledge bases were constructed, this section will present some ideas which have been useful in creating the knowledge bases for these domains. Some common pitfalls are also identified, and alternatives are suggested.

Separating properties from individuals

This is a fundamental technique. It should be noted that many different concepts may classify as individuals, depending on the domain, and on the intent of the application. For example, a person in a story may be an individual, and the triple (Gate1, Gate2, InputPort) as used in Section 3.2 to represent connections in electronic circuits is also an individual. It seems useful to consider the kinds of information that a user might be able to observe.

Direction

In many cases, schemata are naturally written in a causal direction. For example, the behaviour of a logic gate is quite naturally described causally, since its output depends on the inputs. The schemata for a given domain need not be in the causal direction, but should be written with a consistent direction. This is a general observation which seems to aid in keeping knowledge bases from creating directed loops in the Bayesian networks created from it. It may arise that some causal relationships are not easy to quantify, and care should be taken to avoid writing "loops" in the knowledge base.

Conditional probabilities and semantics

While it is true that a conditional probability distribution may take any values which sum to one and remain consistent mathematically, it is important to remember that a contingency table has an intended meaning. When creating the knowledge base, one should be certain that every entry in the contingency table for a schema makes sense. For example, the following schema might look appropriate, according to the direction principle

above

$$\mathsf{bird}, \mathsf{emu} \longrightarrow \mathsf{flies}$$

because both *birdness* and *emuness* have some influence on a bird's ability to fly. However, this schema requires an entry in the contingency table for the expression $p(flies|\neg bird, emu)$, which, if we know anything about emus and birds is inconsistent semantically.

Assigning this probability is context dependent; that is, if we have in addition to the schema about flying, a schema such as

$$\mathsf{emu} \longrightarrow \mathsf{bird}$$

stating that all emus are birds, the actual value of $p(flies|\neg bird, emu)$ is irrelevant. Some axiomatizations may specify a particular value for a probability with inconsistent conditioning (for example, Aleliunas [Aleliunas, 1988] assigns a value of unity). Nevertheless, even if the inconsistency may be easy to disregard in small knowledge bases, in larger ones, keeping track of all the irrelevant inconsistencies seems to be more work than redesigning the problematic schemata for consistency. In our bird example, we might rewrite the schemata as in the following:

$$\begin{aligned} \mathsf{emu} &\longrightarrow \mathsf{bird} \\ \mathsf{emu} &\longrightarrow \mathsf{ab\_flying} \\ \mathsf{bird}, \mathsf{ab\_flying} &\longrightarrow \mathsf{flies} \end{aligned}$$

where ab_flying is a random variable representing the possibility that something is abnormal with respect to flying, *i.e.*, it doesn't fly. Knowing that emus are certainly birds, and that emus tend to be exceptions to the generality that "birds fly" provides the required numbers for the contingency tables.

Keeping the contingency tables semantically consistent and informative will make modifications and extensions of the knowledge base less troublesome.

The use of combination nodes

Combination nodes, such as the *existential* and *universal* combinations discussed in Section 2.2 are used to model multiple influences assuming a simple and regular combination

of effects. They can be used in a straightforward manner, as demonstrated in Example 3.5–1, where the combination was performed over a simple set of individuals. In Section 3.2, a more complex kind of individual was used to create a separate structure for each of these individuals.

In many domains, causal or influential knowledge is best modelled by considering every cause for a particular effect (*e.g.*, every disease which causes the same symptom) independently. Further, each cause is assumed to have an enabling/disabling condition. This kind of assumption is modelled in Pearl and others [Pearl, 1988] by a structure called a *noisy–Or* gate. It has the advantage of requiring a contingency table whose size is on the order of the number of causes, and each entry in the table considers only one cause at a time.

The existential combination introduced in this thesis can be used to construct noisy–or gates. Suppose we wanted to model the causes of sneezing with a noisy–Or gate, combining nodes like flu(X) and hay-fever(X), because it is known that either disease is likely to cause sneezing and that these diseases are independent. This is stated in the following:

$$\begin{aligned} \mathsf{flu(X)} &\longrightarrow \mathsf{causes(flu(X), sneezing)} \\ \mathsf{hay-fever(Y)} &\longrightarrow \mathsf{causes(hay-fever(Y), sneezing)} \\ \exists \mathsf{Z} \in \mathsf{causes} \cdot \mathsf{causes(Z, sneezing)} &\longrightarrow \mathsf{sneezing} \end{aligned}$$

The user would then have to tell the system which causes to consider in the model, and the system would create a structure which combines the causes. This behaves exactly as a noisy–or gate, since listing the causes first is equivalent to enabling the cause, and any cause not explicitly mentioned by the user is not considered by the combination at all, as if the cause were disabled.

Avoiding too many instantiations

It is possible to write a knowledge base of schemata without regard to the process which creates the network. However, it is often helpful to keep the process in mind, to avoid writing schemata which will be instantiated too often. The most obvious way to keep the instantiations limited is to make use of the type constraint mechanism. For example, when writing the schemata for interpreting sketch maps in Section 3.3, we could have used only one type of scene object for both linear and area objects. If the following schema had been used

```
isa(X), isa(Y), joins(X,Y), equal(X,Y) => tee(X,Y)
```

(where `isa(X)` could take any value from `[road, river, shore, water, land]`) instead of

```
isa-linear(X), isa-linear(Y), joins(X,Y), equal(X,Y) => tee(X,Y)
```

as actually used in Section 3.3, the network creation process would have instantiated schemata for all scene objects. No useful information would have been obtained from instantiating them with area scene objects.

3.6 Conclusions

The example knowledge bases discussed in this chapter demonstrate some of the abilities of dynamic Bayesian networks as presented in this thesis.

It is clear that while most schemata are straight forward, some schemata make use of clever techniques which can only be learned by example, and by understanding the network building process. This seems to conflict with the simple declarative representation which seems to characterize the language. presented in Chapter 2.

The process of building networks is very simple; schemata are instantiated by every individual of the appropriate type, and the instantitations are added to the network. As we can see from the sketch map example, sometimes there are many instantiations which do not lead to useful information, and could be left out of the network completely, if the building process were more intelligent.

In work done independently of this thesis, Goldman and Charniak [Goldman and Charniak, 1990] and Breese [Breese, 1990] have presented their work on constructing Bayesian networks from knowledge bases. Both approaches address the representation issues discussed in this thesis, but emphasis is placed on more sophisticated procedures for building networks. Goldman and Charniak use a domain specific rule base to construct networks for the domain, and Breese has domain knowledge in the form of Horn clauses which are used to determine which nodes to include in the network.

Chapter 4

Implementing dynamic Bayesian networks

This chapter deals with the issue of implementing the dynamic features described in Chapter 2. I present a brief description of the algorithms of Poole and Neufeld [1989],Pearl [1988, 1986], Shachter [1986, 1988, 1989], and Lauritzen and Spiegelhalter [1988], considering how each might be adapted for dynamic Bayesian networks.

4.1 The general problem

Ideally, the dynamic constructs would be implemented so that when new individuals are added to the model, resulting in arcs and nodes being added to the network, the joint probability distribution will be correct and obtainable without recomputation of the previous evidence. That is, we want to give some preliminary observations to our system to get some results, possibly observe new individuals, and get the updated results without having to recompute the effects of the initial observations.

Take the simple example of the following schemata:

$$\begin{aligned} \mathsf{a} &\longrightarrow \mathsf{b}(\mathsf{X}) \\ \exists \mathsf{X} \in \mathsf{t} \cdot \mathsf{b}(\mathsf{X}) &\longrightarrow \mathsf{c} \end{aligned}$$

This example contains both a right–multiple schema and a combination schema, and Fig-

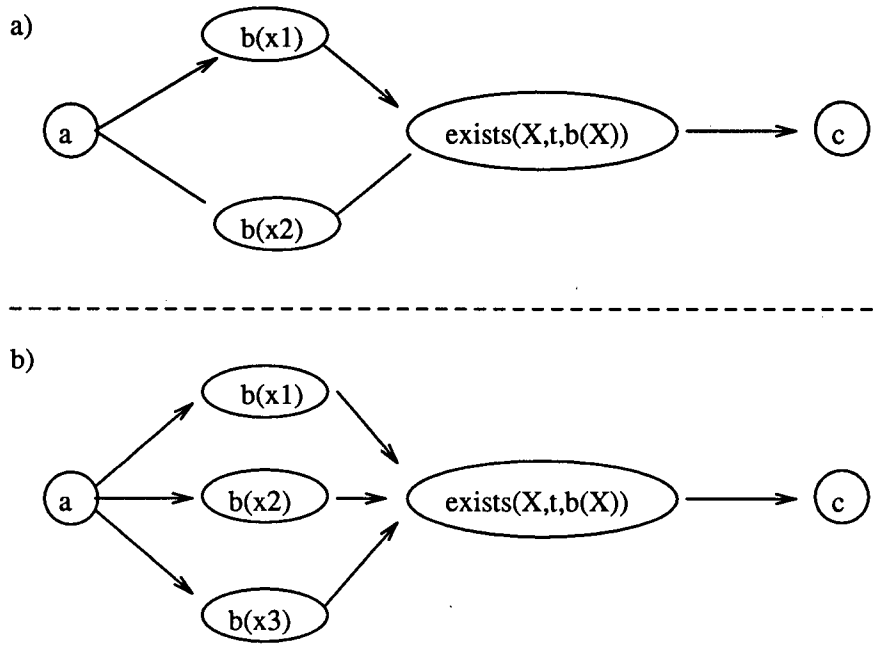

Figure 4.1: A simple example of a dynamic network, (a) showing the original network, and (b) showing the network after another individual of type t is observed. See Section 4.1.

ure 4.1a shows the result of instantiating the schemata for the individuals x_1 and x_2, which are of type t. This network represents the independence assumptions which gives the following equality:

$$\begin{aligned} & p(a, b(x_1), b(x_2), \exists X \in t \cdot b(X), c) \\ = \; & p(c|\exists X \in t \cdot b(X)) \\ & p(\exists X \in t \cdot b(X)|b(x_1)b(x_2)) \\ & p(b(x_1)|a)p(b(x_2)|a) \\ & p(a) \end{aligned}$$

Suppose a new individual x_3 is observed dynamically. The probability model changes as a result, and ideally, the algorithm used in the inference process can absorb the change in the model without too much recomputation. The network is shown in Figure 4.1b, and the joint distribution for our example becomes:

$$\begin{aligned} & p(a, b(x_1), b(x_2), b(x_3), \exists X \in t \cdot b(X), c) \\ = \; & p(c|\exists X \in t \cdot b(X)) \\ & p(\exists X \in t \cdot b(X)|b(x_1)b(x_2)b(x_3)) \\ & p(b(x_1)|a)p(b(x_2)|a)p(b(x_3)|a) \\ & p(a) \end{aligned}$$

This equation demonstrates the change to the model. One new factor has been added, and the existential combination has one more conditioning variable, $b(x_3)$. It is important to note that the factor $p(b(x_3)|a)$ uses the same contingency table as $p(b(x_3)|a)$, so no new information is required. Since the existential combination uses a contingency table which takes the *or* of all the conditions, all the system needs to know is the number of conditions.

To be able to use dynamic schemata efficiently, adding new arcs to the network should allow the algorithm which processes evidence and computes posterior distributions to modify the state of the network in a straightforward manner without recomputing the effects of evidence already submitted.

This is not always easy to do, mainly because most of the algorithms perform some direct manipulation on the structure of the network, either to create a computationally tractible representation or to perform the calculations themselves.

I will show that the addition of dynamic constructs to the work of Poole and Neufeld [Poole and Neufeld, 1989] is quite straight–forward, and that under certain conditions, Pearl's algorithm can be adapted as well. The algorithm due to Lauritzen and Spiegelhalter is not

difficult to adapt. Finally, I show that Shachter's network evaluation algorithm performs too many structural changes on the the network to allow a feasible implementation of dynamic schemata.

4.2 An Axiomatization for Probabilistic Reasoning in Prolog

The influence diagram interpreter described by Poole and Neufeld [Poole and Neufeld, 1989] provides a sound axiomatization for probabilistic reasoning. The computation is goal directed, and only the computations necessary to compute solutions to queries are performed. The `Influence` implementation of dynamic Bayesian networks is based on this interpreter.

4.2.1 The basic computational engine

A Bayesian network is specified by a database of direct dependency relations and contingency tables associated with each variable.

The basic computation is goal directed: the desired conjunction of variables which make up a query is supplied to the system together with the given evidence. Only those computations which are necessary to solve the query are performed, and no computation is performed outside the context of a query. Furthermore, each query is computed independently from any other query.

The axiomatization of probability theory is straight forward, and axioms such as the Negation Rule, the Law of Multiplication of Probabilities, and Bayes' Rule are are used to simplify and solve complex queries recursively. In particular, Bayes' Rule is used to reformulate diagnostic queries (i.e. queries which ask about causes given effects) into a causal form (i.e. queries which ask about effects given causes).

The direct dependencies of variables, made explicit in the network structure, are compiled into Prolog rules which calculate posterior distributions by "reasoning by cases" with respect to all of the variable's direct parents. The independence assumption for the parents of a variable provides some simplification of the general reasoning by cases formula, and weights the cases according to the contingency table provided for the variable.

A query can be seen as a proof of a symbolic probability expression, grounded in terms of

prior probabilities and given knowledge. This expression can be evaluated numerically to provide a quantitative result.

4.2.2 Adding the dynamic combination

Adding the dynamic constructs presented in Chapter 2 requires the addition of several axioms, recognizing the special syntax, and translating a query involving a combination node into a series of simpler queries. For existential combination nodes, the results of the simpler queries are combined using the following lemma:

Lemma 1 *If $U = \{u_1, \ldots, u_k\}$ is a set of random variables such that u_i and u_j are conditionally independent given v for all $i \neq j$, then*

$$p(u_1 \vee u_2 \vee \ldots \vee u_k|v) = 1 - \prod_{i=1}^{k}(1 - p(u_i|v))$$

$$p(u_1 \wedge u_2 \wedge \ldots \wedge u_k|v) = \prod_{i=1}^{k} p(u_i|v)$$

The first step in the combination procedure is to gather the individuals of the correct type. Each individual instantiates the parameterized random variable given in the combination node, and each instantiation is assumed to be independent, given the state of the combination.

For existential combination, the collection of instantiations are combined using Lemma 1 which combines the effects of all the instantiations using the first result. The universal combination is similar, using the second equality.

The ability to observe individuals dynamically is provided by the independence of successive queries. A probability model is based on the information provided by the query only, and therefore a new query, with new evidence in the form of individuals, builds a new probability model implicitly as the proof is generated.

4.2.3 Discussion

The simplicity of this implementation makes it very attractive, both formally and pedagogically. The use of a logical specification helped to keep the developing theory of dynamic combinations correct as well as simple.

One advantage to this approach is that computation is goal driven, and only the calculations necessary to compute the query are performed. The primary disadvantage is that, for consultations with multiple queries, many identical computations may be performed more than once.

As a general belief computation engine, the current implementation is limited, as only very small networks can be queried in reasonable time. Various improvements to the compiler and probability axioms are possible, resulting in a loss of clarity and obvious correctness. Some of these issues are explored in [Horsch, 1990]

4.3 Pearl's Distributed Propagation and Belief Updating

The work on belief propagation done by Pearl [Pearl, 1988] provides a probabilistic computational paradigm which is local in nature, and each local computation requires little or no supervision from a controlling agent. Pearl has proposed a highly parallel "fusion and propagation" algorithm for computing posterior marginal probabilities using singly connected Bayesian networks. I will describe the algorithm abstractly at first, saving the mathematical details for later.

4.3.1 Belief updating in singly connected networks

In Pearl's approach, a Bayesian network describes the channels of communication between related variables. For Bayesian networks which are singly–connected, there are simple rules which govern the way the network is affected by the submission of evidence. These rules deal with:

- How a variable updates its own marginal distribution based on messages received from its direct parents or children.
- The content of messages sent by a variable to its direct parents and children.

For singly–connected graphs, the rules for updating a node's marginal distribution and communicating changes to its parents and children guarantee that the network will converge to a stable equilibrium and the marginal distributions will be correct.

Evidence is submitted to a network by asserting a value for a variable (or a collection of values for different variables). This affects the marginal probability of that particular variable, and this variable sends evidential support values to its parent variables, and causal support values to its children. Each parent or child variable then updates its own posterior distribution based on the information gained from its parents and children, and sends out messages to its parents and children. These messages take into account that some of the parents or children have already seen the evidence, thus ensuring that no support is received more than once. When all variables in the network have updated their distributions in this manner, the propagation ceases.

One of the features of belief propagation is the natural way it can be expressed as a distributed process: the message passing and belief updating are local computations and can be performed in a highly parallel computation.

4.3.2 Adding dynamic constructs

In this section I demonstrate how the dynamic combination nodes can be added to Pearl's singly connected message passing algorithm. In general, a combination node in a singly connected network could create a second path between two nodes. Therefore, to be completely general, Pearl's algorithm must be adapted to handle general graphs. I assume for simplicity that no loops are created when an individual is observed dynamically. This assumption is valid if any of the methods of Section 4.3.3 is used to evaluate the network in the general case of multiply–connected networks.

There are three cases: right–multiple schemata, combination nodes and unique schemata. Right–multiple schemata involves the addition of a new child to a set of nodes. Combination nodes involve the addition of a new parent to a single node. Unique schemata are merely attached to either another unique schema, or at the tail of a right–multiple schema or at the head of a combination node.

The presentation follows Pearl [1988, page 183ff, Equations 4.47 to 4.53], considering the general case of adding new parents or children to the network. The special syntax for combination nodes and right–multiple schemata is important for determining where the new sections of network are to be added, but unimportant with respect to the propagation of support once added.

The dynamic addition of parents

Suppose we have a node C in the network with parents $parents(C) = \{A_1, \ldots, A_n\}$, and children $children(C) = \{B_1, \ldots, B_m\}$ as in Figure 4.2 (the dashed arc from A_{n+1} to C indicates the arc I will be adding).

Intuitively, adding a parent to a node can be seen as revising an assumption about the domain. In principle, the new parent can be treated as if it had always been connected, but having had no effect on the child variable. Adding the new node by a real process only changes the effect the assumed node has on its child.

In the case of an existential combination, we can treat the initial state of the parent as having a prior probability of zero. Adding the node can be seen as changing the message from zero to the value specified in the knowledge base for this node. This assumption effectively states that "all unknown individuals do not have the property we are looking for."

For universal combination nodes, the initial assumed message sent by the parent is unity, that is, we assume that the new node complies with the property being combined. Adding the new parent in this case changes this value from unity to the prior specified in the knowledge base.

Before the addition of A_{n+1} the internal state of C can be inferred from C's neighbours by the following:

$$\begin{aligned} BEL(c) &= p(c|\mathbf{e}) && (4.1)\\ &= p(\mathbf{e}_X^-|x)p(x|\mathbf{e}_X^+) && (4.2)\\ &= \alpha\lambda(c)\pi(c) && (4.3) \end{aligned}$$

where $\mathbf{e}$ is the evidence submitted so far, $\mathbf{e}_X^-$ are those items of the evidence connected to C through its children, $\mathbf{e}_X^+$ are those items of evidence connected to C through its parents, and α is a normalizing constant such that $\sum_c \alpha\lambda(c)\pi(c) = 1$. Now

$$\begin{aligned} \lambda(c) &= p(\mathbf{e}_X^-|x) && (4.4)\\ &= \prod_i \lambda_{B_i}(c) && (4.5)\\ \pi(c) &= p(x|\mathbf{e}_X^+) && (4.6)\\ &= \sum_{a_1,\ldots,a_n} p(c|a_1,\ldots,a_n)\prod_i \pi_C(a_i) && (4.7) \end{aligned}$$

The $\lambda_{B_i}(c)$ are messages received by C from its children, and $\pi_C(a_i)$ are messages received by C from its parents.

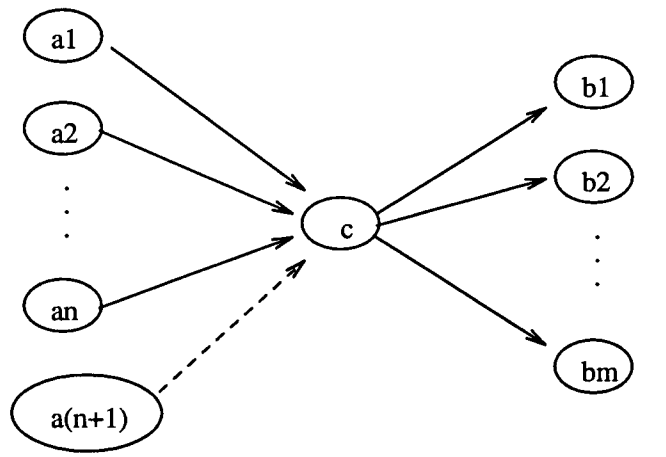

Figure 4.2: Adding a parent A_{n+1} to node C.

The probabilities $p(c|a_1,\ldots,a_n)$ are those supplied as contingency tables. In the case of combination nodes, these are functional: for existential combinations, we use a function which computes $p(c|a_1,\ldots,a_n) = \bigvee_{i=1}^{n} a_i$, and for universal combination we use $p(c|a_1,\ldots,a_n) = \bigwedge_{i=1}^{n} a_i$.

Adding a parent A_{n+1} to $parents(C)$ requires an update for the state of C. Assume that no new evidence is submitted at the time of adding the new parent. Let $BEL'(c)$ be the new state. Therefore:

$$\begin{aligned} BEL'(c) &= p(c|\mathbf{e}) && (4.8)\\ &= \alpha\lambda'(c)\pi'(c) && (4.9)\\ \lambda'(c) &= \prod_i \lambda_{B_i}(c) && (4.10)\\ &= \lambda(c) && (4.11)\\ \pi'(c) &= \sum_{a_1,\ldots,a_{n+1}} p(c|a_1,\ldots,a_{n+1}) \prod_i \pi_C(a_i) && (4.12) \end{aligned}$$

Note that $\lambda(c)$, which is the support gained from C's children doesn't change. The new $\pi'(c)$ reflects the addition of the new parent. The contingency table used before cannot in general be used in this new calculation; a new table must be supplied. However, in the case of combination nodes, these probabilities are functional and can be written to accept an arbitrary number of dependent variables.

Having updated its own internal state, C must send appropriate messages to all of its children and every parent except the new one. As well, the new parent must be made aware of the evidence previously seen by the network.

The content of the message sent by C to its parents due to the addition of A_{n+1} can be written as:

$$\lambda_C(a_i) = \beta \sum_c \lambda(c) \sum_{a_k:k\neq i}^{n+1} p(c|a_1,\ldots,a_{n+1}) \prod_k \pi_C(a_k) i \neq n+1 \qquad (4.13)$$

where β is any convenient constant. Note that each parent of C receives information from each of its mates, and about every child of C from $\lambda(c)$.

The message C sends to A_{n+1} should in one message provide information about all the evidence seen by C so far. The message sent can be written:

$$\lambda_C(a_{n+1}) = \beta \sum_c \lambda(c) \sum_{a_i:i\neq n+1} p(c|a_1,\ldots,a_{n+1}) \prod_i \pi_C(a_i) \qquad (4.14)$$

The message sent to each child B_i in $children(c)$ is:

$$\pi_{B_i}(c) = \alpha \prod_{k \neq j} \lambda_{B_k}(c) \sum_{a_1,\ldots,a_{n+1}} p(c|a_1,\ldots,a_{n+1}) \prod_i \pi_C(a_i) \quad (4.15)$$

$$= \alpha \frac{BEL'(c)}{\lambda_{B_i}(c)} \quad (4.16)$$

Finally, the added parent A_{n+1} can update its own state based on the message received from C and send messages to its children and parents in the manner originally outlined by Pearl.

The dynamic addition of children

Suppose we have node A in a network with parents $parents(A) = \{D_1,\ldots,D_l\}$ and children $children(A) = \{B_1,\ldots,B_n\}$, as in Figure 4.3 (again, the dashed arc from A to B_{n+1} indicates the one that will be added).

Intuitively, the addition of the new child is similar to the way parents were added in combination nodes. It can be assumed that the child has always been connected, but having had no initial effect on its parent. The assumed initial state gives no preference to the values of the child, that is, the assumed prior is $1/v$, where v is the number of possible outcomes of the new child. Adding the child has the effect of changing the assumed irrelevance to the value specified by the knowledge base.

Before the addition of the new child B_{n+1}, the internal state is given by:

$$BEL(a) = p(a|\mathbf{e}) \quad (4.17)$$

$$= \alpha\lambda(a)\pi(a) \quad (4.18)$$

where $\mathbf{e}$ is the evidence submitted so far, and α is a normalizing constant. Now

$$\lambda(a) = \prod_i \lambda_{B_i}(a) \quad (4.19)$$

$$\pi(a) = \sum_{d_1,\ldots,d_l} p(a|d_1,\ldots,d_l) \prod_i \pi_A(d_i) \quad (4.20)$$

Adding a child D_{n+1} to $children(A)$ requires an update for the state of A. Assume that no new evidence is submitted at the time of adding the new parent. Let $BEL'(a)$ be the

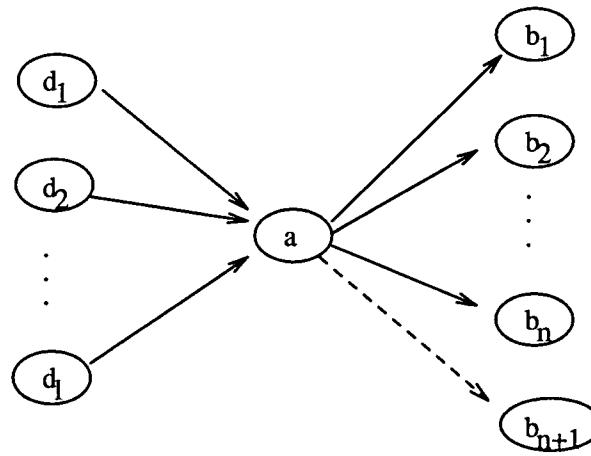

Figure 4.3: Adding a child B_{n+1} to node A.

new state. Therefore:

$$\begin{aligned} BEL'(a) &= p(a|\mathbf{e}) && (4.21)\\ &= \alpha\lambda'(a)\pi'(a) && (4.22)\\ \lambda'(a) &= \prod_i \lambda_{B_i}(a) && (4.23)\\ &= \lambda(a)\lambda_{B_{n+1}}(a) && (4.24)\\ \pi'(a) &= \sum_{d_1,\ldots,d_l} p(a|d_1,\ldots,d_l)\prod_i \pi_C(a_i) && (4.25)\\ &= \pi(a) && (4.26) \end{aligned}$$

Note that $\pi(a)$, which is the support gained from A's parents doesn't change. The new $\lambda'(a)$ reflects the addition of the new child.

Having updated its own internal state, A must send appropriate messages to all of its children except the new one, and to every parent. As well, the new child must be made aware of the evidence previously seen by the network.

The content of the message sent by A to its parents due to the addition of B_{n+1} can be written as:

$$\lambda_A(d_i) = \beta\sum_a \lambda'(a) \sum_{d_k:k\neq i} p(c|d_1,\ldots,d_l)\prod_k \pi_A(d_k) \quad (4.27)$$

where β is any convenient constant. Note that each parent of A receives information from each of its mates from $\pi_A(d_k)$ and about every child of A from $\lambda'(a)$ and

The message sent to each child $B_i, i \neq n+1$ in $children(A)$ is:

$$\begin{aligned} \pi_{B_i}(a) &= \alpha\prod_{k\neq j}\lambda_{B_k}(a)\sum_{d_1,\ldots,d_l} p(a|d_1,\ldots,a_l)\prod_i \pi_A(d_i) && (4.28)\\ &= \alpha\frac{BEL'(a)}{\lambda_{B_i}(a)} && (4.29) \end{aligned}$$

The message A sends to B_{n+1} can be written:

$$\begin{aligned} \pi_{B_{n+1}} &= \alpha\prod_k^n \lambda_{B_k}(a)\sum d_1,\ldots,d_l p(a|d_1,\ldots,a_l)\prod_i \pi_A(d_i) && (4.30)\\ &= \alpha\lambda(a)\pi(a) && (4.31)\\ &= BEL(a) && (4.32) \end{aligned}$$

which is the previous state of A.

Finally, the added child D_{n+1} can update its own state based on the message received from A and send messages to its children and parents in the manner originally outlined by Pearl.

Adding unique schemata dynamically

Adding unique schemata requires no special handling. This is because they can be attached only to nodes which were added dynamically as part of a right–multiple or combination structure. Their internal state is inferred from their parents and children, which are also new to the network. The standard computation applies to these nodes.

When the message due to the change in network structure reaches the nodes of the unique schemata, the updating and subsequent message passing occurs in the manner described by Pearl.

4.3.3 Propagation in general Bayesian networks

For multiply–connected networks, Pearl's message passing scheme does not apply. The problem is that networks which have multiple causal chains between variables lose the conditional independence assumption which permitted the local propagation algorithm to distribute messages along each arc. Furthermore, then network can get caught in an infinite stream of messages. Solutions to this problem include:

- Keeping a list of variables in the history of the propagation. This requires message lengths exponential in the number of variables in the network.
- Clustering sibling nodes along the multiple paths, creating a single chain. This requires possibly exponential time to identify loops and form the clusters.
- Disconnecting the loop at the head (i.e. at the variable which begins the multiple chains). The variable at the head is instantiated to each value it can attain, and the effects of this instantiation are averaged. This requires computations exponential in the number of values the head of a loop can attain.
- Stochastic simulation, in which the variables take random values according to the probability distribution derived from the current state of its direct parents. The

posterior distribution is then taken as the ratio of assigned values to the number of trial runs made. The drawback to this approach is that convergence to the correct distribution is guaranteed under certain conditions, but the number of trials necessary for a particular accuracy is undetermined.

4.3.4 Discussion

I have shown how Pearl's fusion and propagation algorithm can be modified to handle dynamic constructs in a way which correctly reflects the change in the probability model.

The adaptation is at a cost of locality: the purely local nature must be enhanced by a controlling program which adds nodes and arcs to the net and starts the appropriate message propagation.

Finally, I have only treated the special case where the dynamic constructs do not create loops in previously singly–connected networks. A dynamic scheme to handle this general case, using the ideas presented in Section 4.3.3, would be another interesting project.

4.4 Lauritzen and Spiegelhalter's computation of belief

Lauritzen and Spiegelhalter [Lauritzen and Spiegelhalter, 1988] address the issue of performing local computations of beliefs using Bayesian networks as expert system inference engines. In particular, they are concerned with computing beliefs on large, sparse networks which are not necessarily singly connected.

Briefly, their approach first modifies the structure of the Bayesian network representing the domain knowledge, creating a *full moral graph.* Cliques are identified, and the belief computation uses them to propagate support values throughout the network.

4.4.1 Belief Propagation on Full Moral Graphs

Their algorithm can be described as consisting of three steps.[1] First, the Bayesian network is triangulated (and referred to as a full moral graph). Second, cliques in the triangularized network are ordered using a maximal cardinality search (i.e. starting from an arbitrary node, of all neighbour nodes yet to be labelled, the node having the most already labelled neighbours is labelled first), and each clique is treated as a compound variable in a tree structure connecting these cliques. Finally, the ordering of the maximal cardinality search is used to direct the arcs in the tree, and the method of propagation in singly–connected networks is used to update probabilities.

4.4.2 Adding dynamic constructs

Briefly, Lauritzen and Spiegelhalter's algorithm can be adapted to include the dynamic structures in the following manner. The new section of the network is added (dropping the direction of the arcs) to the extant triangularized structure. The network is retriangularized, and following the above procedure, cliques are identified and a tree of compound nodes is created. At this point the propagation procedes on the new structure in the familiar manner.

The complication of adding the new nodes to the triangularized structure can be approached by using the observation from Section 4.3.2, namely treating the arcs as if they had always been in the network, but having had no effect on the rest of the network. This observation can be used to change the marginal distribution of each clique which has a new node in it or is a neighbour to a such a clique.

4.5 Probabilistic inference using Influence diagrams

In Operations Research, influence diagrams have been used as a common analysis tool for decision analysis under uncertainty. The approach of Shachter is to provide a normative axiomatic framework for this domain, which has often used *ad hoc* methods [Shachter, 1986, Shachter, 1988, Shachter, 1989].

An influence diagram is a network similar to a Bayesian network, relating random vari-

[1]This observation is due to Pearl, from his commentary on [Lauritzen and Spiegelhalter, 1988], and also noted by Shachter in [Shachter, 1989].

ables and identifying probability distributions graphically. In addition to random variables (called *chance nodes* in Operations Research), influence diagrams also have nodes to represent expected outcomes and decisions. Influence diagrams without these special nodes are Bayesian networks, and in this overview, these will only be considered in passing.

For purposes of decision analysis, all probabilistic nodes are removed from the network, transferring their effect on decisions to a single table which ranks the decision options based on expected cost. For less specific applications, Shacter has shown [Shachter, 1989] that his algorithm can be used for evidence absorption and propagation by removing evidence nodes from the network.

Network evaluation is performed using two techniques: arc reversal and node removal. A barren node, i.e. a node with no children, can be removed from a network without affecting the underlying distribution. An arc can be reversed, that is, the arc from a node A to a node B can be tranformed into an arc from B to A, if there is no other directed path from A to B. This transformation reverses the conditional probabilities as well, and both A and B are "adopted" by each other's direct parents. When only probabilistic nodes are used in the diagram, the procedure of arc reversal corresponds to Bayes' theorem. Arc reversals are used to change nodes having children into barren nodes so they can be removed from the network.

Network evaluation can be seen as the sequence of node removals and arc reversals necessary to remove all nodes from the network. Queries to the network can be performed by the sequence of arc reversals necessary to remove the nodes which condition the query.

4.5.1 Adding dynamic constructs

Given the very dynamic network evaluation already inherent in Shachter's algorithm, it is not especially helpful to add the kind of dynamic structures I have presented in Chapter 2.

As an example of the difficulties involved, consider the network in Figure 4.4a. Following Shachter's algorithm, evaluating the network with a series of node removals and arc reversals results in the network of Figure 4.4b. If a node B_3 is added to the network in this state, perhaps because a user has just discovered a new individual in the model, the system must perform the evaluation again from the original network.

In general, an influence diagram could be built by instantiating a collection of schemata, and Breese demonstrates this by taking a similar approach to building influence diagrams [Breese, 1990]. Shachter's evaluation could be adapted to use combination nodes (*i.e.*, to

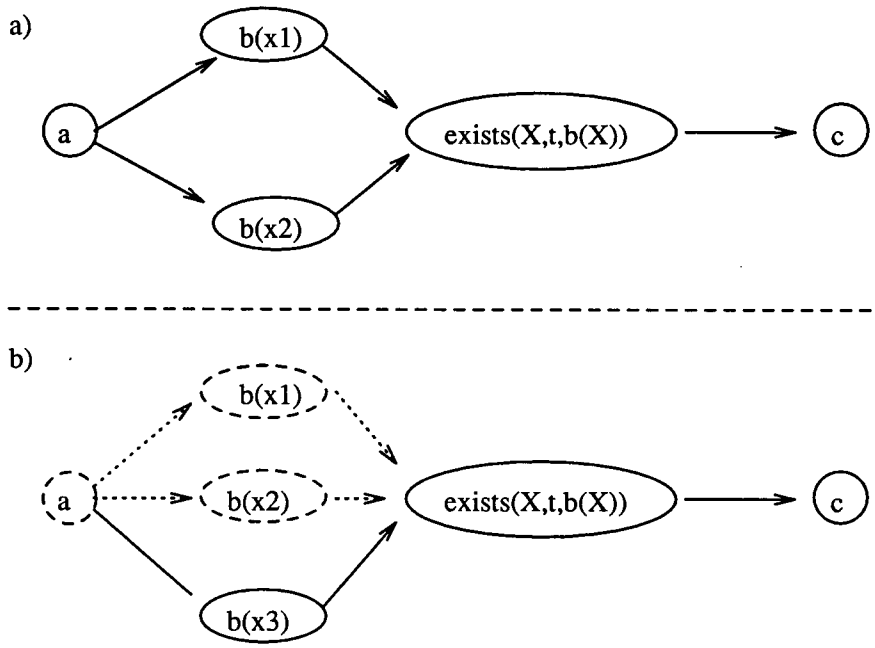

Figure 4.4: The problem of adding a node after arc reversals have been performed.

be used as a noisy–Or gate as in Section 3.5, for example), but if individuals are observed after a network has already been created, and after some evaluation has been performed, a new network must be created, and all of the previous arc reversals and node removals must be repeated for the new network.

4.6 Conclusions

In this chapter I have presented several Bayesian network evaluation techniques, and discussed how the dynamic techniques of Chapter 2 might be implemented.

Some of these algorithms lend themselves well to dynamic schemata, and they benefit from the observation that the parts of the network which are added dynamically can be treated as if they had always been present in the network; the addition merely has the effect of changing how these nodes affect the rest of the network. There are other network evaluation algorithms in use or being developed which have not been discussed in this chapter. These may also be adaptable to dynamic constructs and may benefit from this same observation.

The question of whether an evaluation algorithm can be modified to use dynamic schemata says more about the intended application than about the algorithm itself. For expert system use, a Bayesian network is a batch process, and the computation is designed to be autonomous and inflexible. These constraints are quite exploitable for implementing dynamic constructs for these algorithms. The fact that Shachter's algorithm is typically used in a very interactive consultation with a user, who is quite likely an expert in the domain she is modelling, means that the dynamic constructs are essentially unnecessary. The user herself can add to the network interactively should the necessity arise.

Chapter 5

Conclusions

5.1 What dynamic Bayesian networks can't do

In this section, we consider the limitations and disadvantages of dynamic Bayesian networks as presented in this thesis. Some of these issues could be addressed as part of future research springing from this thesis.

The most obvious disadvantage to the process of instantiating parameterized schemata to create Bayesian networks is the problem of finding individuals: the user of the system must be fully aware of the individual types, and the criteria for deciding types. Furthermore, the correctness of the model constructed by the process is dependent on the competence of the user who supplies the individuals.

In a similar vein, the combination mechanisms presented in Chapter 2 take only known individuals into account when performing the combination. A facility for hypothesizing some unknown individual, perhaps based on the condition that all causes due to known individuals have been dismissed, seems to be a valuable addition.

Another related issue is the type hierarchy itself. As implemented, the network creating process requires that the user identify the individual's type, and if the individual belongs to more than one type (perhaps because private detectives are people too), the user must identify all the types to which an individual belongs. Of immediate benefit to the ideas presented in this thesis would be a study on hierarchical types, with an eye towards efficiency considerations and representational adequacy.

The fact that the knowledge base of schemata implicitly represents possibly infinitely many,

possibly infinitely large directed networks, the possibility that some of those networks are not acyclic, *i.e.,* not Bayesian networks, is cause for concern. Adding a loop check to the network creation process implementation is a simple matter, but it only identifies directed loops after the schemata are instantiated. Some utility or theorem which assures that a knowledge base will not result in cyclic networks would be a desirable result. Currently, the possibility of loops is treated in the same way left–recursive logic programs are treated by the logic programming community.

5.2 Other future possibilities

One of the fundamental ideas of my approach was to keep the network construction process simple, so that the issue of representation could be more fully addressed. One obvious extension of this thesis would be to design a more sophisticated construction process which would maintain the consistency of the probability distribution and the precision of the mapping between the knowledge base and the networks which can be created from it.

The existential and universal combination mechanisms presented in Chapter 2 are only two of many possible ways to combine information. It seems useful to consider how to implement a mechanism in which some fraction of a set of parent random variables must be true in order for an effect to be triggered.

5.3 Some final remarks

This thesis has presented a dynamic approach to the use of Bayesian networks. The approach was motivated by the need to model domains in which it would be difficult to anticipate some of the details. In particular, for domains which can be modelled by general knowledge of properties and their interactions, this general knowledge can be used in specific instances to create a probabilistic model. Changing the instances changes the model.

A simple language of parameterized probabilistic knowledge, augmented with structures which dynamically combine the effects of several causes, was described. A procedure to create networks by instantiating statements in this language with particular individuals which have been submitted to the system, was outlined as well.

An implementation of this language was described briefly, and several examples were pre-

sented using this implementation. The language seems to be versatile, but requires some careful programming for effective use.

Several algorithms were discussed, due to Pearl [1988], Poole and Neufeld [1989], Lauritzen and Spiegelhalter [1988], and Shachter [1986], currently used to evaluate Bayesian networks, and some modifications were suggested which would implement dynamic Bayesian networks in these systems.

This thesis focussed on providing a precise framework for representing parameterized probabilistic knowledge, and a firm basis for building networks from this knowledge. Because the network construction process presented here was kept quite simple, the issue of representation could be more fully addressed. The result is a language for representing parameterized probabilistic dependencies which precisely defines the mapping between the knowledge and the Bayesian networks created from it, as well as ensuring a consistent probability distribution which is based on all available knowledge.

Bibliography

[Aho *et al.*, 1974] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.

[Aleliunas, 1988] R. Aleliunas. A new normative theory of probabilistic logic. In *Proc. Seventh Conf. of the CSCSI*, pages 67–74, 1988.

[Andreassen *et al.*, 1987] S Andreassen, M. Woldbye, B. Falck, and S.K. Andersen. Munin—a causal probabilistic network for interpretation of electromyographic findings. In *Procedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 366–372, 1987.

[Breese, 1990] Jack Breese. Construction of belief and decision networks. Rockwell Internation Science Center, forthcoming, 1990.

[Cheeseman, 1988] P. Cheeseman. In defense of probability. *Computational Intelligence*, 4(1):58–66, 1988.

[Cooper, 1990] G. F. Cooper. The computational complexity of probabilistc inference using bayesian belief networks (research note). *Artificial Intelligence*, 42(2–3):393–405, 1990.

[D'Ambrosio, 1989] Bruce D'Ambrosio. Incremental goal–directed probabilistic inference. Dept. of Computer Science, Oregon State University (draft), 1989.

[de Kleer and Williams, 1987] J. de Kleer and B.C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[Gardenfors, 1988] Peter Gardenfors. *Knowledge in Flux.* MIT Press, Cambridge, Mass., 1988.

[Geiger, 1990] Dan Geiger. Graphoids: A qualitative framework for probabiolistic inference. Technical Report R-142, Cognitive Systems Laboratory, Dept. of Computer Science, UCLA, 1990.

[Goldman and Charniak, 1990] Robert P. Goldman and Eugene Charniak. Dynamic construction of belief networks. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence*, pages 90–97, 1990.

[Horsch and Poole, 1990] Michael C. Horsch and David Poole. A dynamic approach to probabilistic inference using bayesian networks. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence*, pages 155–161, 1990.

[Horsch, 1990] Michael C. Horsch. Influence: An interpreter for dynamic bayesian networks. Dept. of Computer Science, University of British Columbia. In preparation., 1990.

[Jensen *et al.*, 1988] F.V. Jensen, K.G. Olesen, and S.K. Andersen. An algebra of bayesian belief universes for knowledge based systems. Technical Report R-88-25, Institute of Electronic Systems, Aalborg University, 1988.

[Jensen *et al.*, 1990] F.V. Jensen, S.L Lauritzen, and K.G Olesen. Bayesian updating in recursive graphical models by local computations. To appear in: Networks, 1990.

[Laskey, 1990] Kathryn Blackmond Laskey. A probabilistic reasoning environment. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence*, pages 415–422, 1990.

[Lauritzen and Spiegelhalter, 1988] S. L. Lauritzen and D. J. Spiegelhalter. Local computation with probabilities on graphical structures and their application to expert systems. *J. R. Statist Soc B*, 50(2):157–224, 1988.

[Lindley, 1965] D. V. Lindley. *Introduction to Probability & Statistics from a Bayesian viewpoint. Part 1. Probability.* Cambridge University Press, 1965.

[Mulder *et al.*, 1978] J. A. Mulder, A. K. Mackworth, and W. S. Havens. Knowledge structuring and constraint satisfaction: The mapsee approach. Technical Report 87-21, Dept. of Computer Science, University of British Columbia, Vancouver., 1978.

[Neufeld and Poole, 1987] E. Neufeld and D. Poole. Towards solving the multiple extension problem: Combining defaults with probability. In *Procedings of the Third Workshop on Uncertainty in Artificial Intelligence*, pages 305–312, Seattle, 1987.

[Pearl, 1986] Judea Pearl. Fusion, propagation and structuring in belief networks. *Artificial Intelligence*, 29(3):241–288, 1986.

[Pearl, 1988] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Reasoning.* Morgan Kauffman Publishers, Los Altos, 1988.

[Poole and Neufeld, 1989] David Poole and Eric Neufeld. Sound probabilistic inference in prolog: An executible specification of bayesian networks. 1989.

[Poole and Provan, 1990] David Poole and Gregory Provan. What is an optimal diagnosis. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence*, pages 46–53, 1990.

[Poole, 1989] David Poole. A methodology for using a default and abductive reasoning system. Technical Report 89–20, Dept. of Computer Science, University of British Columbia, Vancouver, 1989.

[Reiter and Mackworth, 1989] R. Reiter and A.K. Mackworth. A logical framework for depiction and image interpretation. *Artificial Intelligence*, 41(2):125–155, 1989.

[Reiter, 1987] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[Schubert, 1988] J. K. Schubert. Cheeseman: A travesty of truth. *Computational Intelligence*, 4(1):118–121, 1988.

[Shachter, 1986] Ross D. Shachter. Evaluating influence diagrams. *Opns Rsch*, 34(6):871–882, 1986.

[Shachter, 1988] Ross D. Shachter. Probabilistic inference and influence diagrams. *Opns Rsch*, 36(4):589–604, 1988.

[Shachter, 1989] Ross D. Shachter. Evidence absorption and propagation through evidence reversals. In *Proceedings of the 9th Annual Workshop on Uncertainty in Artificial Intelligence*, pages 303–308, 1989.

Appendix A

Influence **code for the examples**

A.1 Diagnosis of multiple faults for digital circuits

The following is a complete listing of the knowledge base for modelling circuits. See Section 3.2.

```
binary ok(Gate:gate).
binary output(Gate:gate).
binary input(Gate:gate, Port:port).
binary exists((G1, G2, Port), connections, output(G1)).

/* and--gates */
ok(and_gate(G):gates),
input(and_gate(G):gates,1),
input(and_gate(G):gates,2) => output(and_gate(G):gates)
                := [1,0,0,0,0.5,0.5,0.5,0.5].

/* or--gates */
ok(or_gate(G):gates),
input(or_gate(G):gates,1),
input(or_gate(G):gates,2) => output(or_gate(G):gates)
                := [1,1,1,0,0.5,0.5,0.5,0.5].

/* xor--gates */
ok(xor_gate(G):gates),
input(xor_gate(G):gates,1),
input(xor_gate(G):gates,2) => output(xor_gate(G):gates)
                := [0,1,1,0,0.5,0.5,0.5,0.5].
```

```
exists((G1,G2,Port), connections, output(G1))
            => input(G2:gates,Port:ports).
                := [1,0].


observe type(and_gate(g1),gates).
observe type(or_gate(g2),gates).
observe type(xor_gate(g3),gates).
observe type((and_gate(g1),xor_gate(g3),1) connections).
observe type((or_gate(g2),xor_gate(g3),2) connections).
```

A.2 Interpreting sketch maps

The following Influence code creates dynamic Bayesian networks for the domain discussed in Section 3.3

```
%% scene objects can be linear or area objects
multi isa-linear(X) @ [road, river, shore].
multi isa-area(X) @ [land, water].

%% Image objects:
binary tee(X,Y). %% chain X meets chain Y in a tee-shape
binary chi(X,Y). %% chain X and chain Y make a chi--crossing
binary bounds(X,Y). %% chain X bounds area Y
binary interior(X,Y). %% Area Y is in the interior of chain X
binary exterior(X,Y). %% Area Y is exterior of chain X

%% Scene descriptions:
binary joins(X,Y). %%linear scene objects join in a tee shape
binary crosses(X,Y). %% linear scene objects cross
binary inside(X,Y). %% area object X is inside linear object X
binary outside(X,Y). %% area object X is outside linear object X

%% Trick implementation of equality predicate!
%% Two distinct random variables, which have their
%% arguments as values:
multi val1(X) @ [X, X].
multi val2(Y) @ [Y, Y].

%% equality is then define if the values are the same!
function equal/2.
```

```
val1(X), val2(Y) => equal(X,Y)
        {p(equal(X,Y),[val1(X)=X,val2(Y)=Y])
            = if (X=Y) then 1.0 else 0.0}.

%% Two linear scene objects can join to form a tee
isa-linear(X:chain), isa-linear(Y:chain),
joins(X:chain,Y:chain), equal(X:chain,Y:chain)
        => tee(X:chain,Y:chain)
        := [1, %% a road can meet itself in a tee, but nothing else
            0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, %% equal objects...
            1,1,0, %% a road or shore can join a road, a shore cannot
            1,1,0, %% a road or a river can join a river, a shore cannot
            1,1,0, %% a road or a river can join a shore, a shore cannot
            0,0,0,0,0,0,0,0,0]. %% join is false

%% two linear objects can cross to form a chi
isa-linear(X:chain), isa-linear(Y:chain),
crosses(X:chain,Y:chain), equal(X:chain,Y:chain)
        => chi(X:chain,Y:chain)
%% no scene objects cross themselves
        := [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, %% (equal is true)
            1,1,0, %% roads and rivers can cross a road, not shores
            1,0,0, %% roads can cross a river, not rivers or shores
            0,0,0, %% shores cannot cross anything
            0,0,0,0,0,0,0,0,0]. %% cross is false, so chi is not possible

%% linear objects can form closed loops
isa-linear(X:chain), loop(X:chain) => closed(X:chain)
        := [1,0,1,0,0,0]. %% only roads and shores form loops

%% linear scene object are found beside area objects
isa-area(X:region), isa-linear(Y:chain), beside(X:region,Y:chain)
        => bounds(X:region,Y:chain).
        := [1,0, %% only land can be beside roads
            1,0, %% only land can be beside rivers
            1,1, %% land and water can be beside shores
            0,0,0,0,0,0]. %% beside is false

%% on the linear object which forms the boundary between two
%% area objects, we must constrain the objects involved
%% e.g. A road is not a boundary between two bodies of water
isa-area(X:region), isa-linear(Y:chain), isa-area(Z:region),
inside(X:region,Y:chain), outside(Z:region,Y:chain)
        => boundary-constraint(X:region,Y:chain,Z:region).
        := [1,0, %% only land can be inside a road boundary
            0,0, %% river boundaries are not allowed
```

```
0,1, %% shores have water inside, land outside
0,0,0,0, %% when water is outside, roads, rivers can't bound
1,0, %% water outside, shore boundary then only land inside
0,0,0,0,0,0,0,0,0,0,0,0, %% inside and outside are false
0,0,0,0,0,0,0,0,0,0,0,0,0].
```

A.3 The Burglary of the House of Holmes

The following Influence code models the problem of Section 3.4.

```
binary sale-obtained(X). %% has a sale been obtained by X?
binary burglary(X). %% was there a burglary in X's house?
binary go-home(X). %% does (should) X go home IMMEDIATELY?
binary meeting(X,Y). %% was there a meeting between X and Y?
binary earthquake.  %% was there an earthquake?
binary radio.  %% did the radio report an earthquake ?
binary client(X). %% X has a client?

function value(X). %% what is the value of X's decision?
function losses(X). %% how much did X lose?
function income(X). %% how much does X stand to gain?

%% how much is a sale worth?
multi sale-value(X) @ [0,150,300].

%% how much of X's stolen goods were recovered?
multi goods-recovered(X) @ [0,500,5000].

%% how much were X's stolen goods worth?
multi stolen-goods-value(X) @ [0,500,5000].

%% when did X report thre burglary?
multi burglary-report(X) @ [immediate, late, none].


%% compute the difference between X's income and losses
%% call it value for X
losses(X:alarm-owner), income(X:alarm-owner) => value(X:alarm-owner)
            := {p(value(X)=V,[losses(X)=V1,income(X)=V2])
                if (V is V2 - V1) then 1.0 else 0.0}.
%% X may recover some of his stolen goods depending on
%% when X reported the burglary
```

```
burglary(X:alarm-owner), burglary-report(X:alarm-owner)
        => goods-recovered(X:alarm-owner)
        := [0.01, 1, 0.5, 1, 0.1, 1].

%% If X recovers stolen goods, then there is not loss
goods-recovered(X:alarm-owner),
stolen-goods-value(X:alarm-owner) => losses(X:alarm-owner)
        := {p(losses(X)=V,[goods-recovered(X)=V1,stolen-goods-value(X)=V2])
            if (V is V2 - V1) then 1.0 else 0.0}.
%% the income for Y depends on getting the sale from a client
client(Y:alarm-owner), sale-obtained(Y:alarm-owner),
sale-value(Y:alarm-owner) => income(Y:alarm-owner).
        := {p(income(Y)=V,[client(Y), sale-obtained(Y),
                    sale-value(Y)=V1])
            if (V is V1) then 1.0 else 0.0}.
%% - A meeting may take place between X and Y
%% if X doesn't go home
go-home(X:alarm-owner) => meeting(X:alarm-owner,Y:corporation)
        := [0, 0.75].
%% X may get a client if X has a meeting with a corporation
exists(X,corporation,meeting(Y:alarm-owner,X))
        => client(Y:alarm-owner)
        := [0.75,0.0].

%% X will report a burglary if one occurred, and X has gone home
%% to verify it go-home(X:alarm-owner), burglary(X:alarm-owner)
        => burglary-report(X:alarm-owner)
        := [1.0, %% X went home immediately and reported a burglary
            0.0, %% so the report isn't late...
            0.0, %% ...or not reported at all
            0.0, %% didn't go home, so report can't be immediate
            1.0, %% assume X reports a burglary eventually...
            0.0, %% (X reported the burglary LATE, not NONE)
%% no burglary, no report!
            0.0, 0.0, 1.0, 0.0, 0.0, 1.0].

%% if there was a burglary, then goods of some value have been
%% stolen
burglary(Y:alarm-owner) => stolen-goods-value(Y:alarm-owner)
        := [0.05, %% burglary, but s-g-v=0
            0.55, %% burglary, but s-g-v=500
            0.4, %% burglary, but s-g-v=5000
            1.0, %% no burglary, and s-g-v=0
            0.0, %% no burglary, and s-g-v=500
            0.0].%% no burglary, and s-g-v=5000
```

```
%% X will meet with Y to talk sales.  A sale may occur
meeting(X:alarm-owner,Y:corporation)
        => sale-obtained(X:alarm-owner,Y:corporation)
        := [0.5, 0.1].

%% an alarm is affected by earthquakes and burglaries
%% - the alarm will almost definitely sound if both an earthquake
%% and a burglary occur, and almost definitely will not sound if
%% neither occur
burglary(Y:alarm-owner), earthquake => alarm(Y:alarm-owner)
            := [0.99, 0.1, 0.8, 0.001].
%% earthquakes tend to be reported on the radio...
earthquake => radio
        := [0.98, 0.0].

%% Phone calls from neighbours about alarms.
%% - neighbours usually only call when an alarm sounds
%% - non-neighbours don't call at all!
neighbour(X:person,Y:alarm-owner),
    alarm(Y:alarm-owner) => phone-call(X:person,Y:alarm-owner)
        := [0.6, 0.0, 0.2, 0.0].
```

Appendix B

Source code to Influence

B.1 probability.pl

```
/*
 * probability.pl
 * The set of probability axioms from which we can compute
 * many probabilities without much effort. The real
 * work is in the last axiom, which is Bayes' theorem,
 * and the call to dyn_p, where we reason by cases.
 *
 * (The Multiplication Rule for probabilities used here
 * is really stupid. Improvements can be made here!)
 */

%% The first two rules allow the user to query conveniently.
%% And displays the answer nicely, to boot...

p(A)
  :- p(A,[]).

%% First, check to see if the conditioning knowledge is consistent...
p(A,B) :-
  given_inconsistent(B),
  write_p(A,B,'undefined'),nl.

p(A,B)
  :- p(A,B,P),
     write_p(A,B,P),nl.
```

```
%% p(+A,+Given,-Probability)
%% - calculate the probability of logical expression +A conditioned
%%   by +Given, to be returned in -Probability
%% - if +A is a list, then this list is a conjunct of propositions

%% Check to see if it is a prior that we know in the database.
%% p(X,A,P)
  %% :- pr(X,A,P,_).

%% If X is a predefined node type, then we do it here.
p(X,A,P)
  :- built_in_p(X,A,P).

%% Otherwise, we have to do some calculations...through axiom_p
p(X,A,P)
  :- axiom_p(X,A,P).

%% If X is a quantification node...
p(X,A,P)
  :- quant_p(X,A,P).

%% If none of the axioms above can calculate a value, then
%%   we must resort to reasoning by cases, which is performed
%%   by the rules created during compilation, under the
%%   head of 'dyn_p'.
p(X,A,P)
  :- dyn_p(X,A,P).

quant_p(exists(Parameter,Type,Variable),Cond,P)
  :- get_indivs_of_type(Type,Cond,Indivs),
     instantiate(Variable, Parameter, Indivs, Instances),
     or_p(Instances,Cond,P).

quant_p(forall(Parameter,Type,Variable),Cond,P)
  :- get_indivs_of_type(Type,Cond,Indivs),
     instantiate(Variable, Parameter, Indivs, Instances),
     and_p(Instances,Cond,P).

%% get_indivs_of_type(Type,Cond,Indivs)
get_indivs_of_type(_,[],[]).
get_indivs_of_type(Type,[type(Indiv,Type)|Conds],[Indiv|Indivs])
  :- !,
     get_indivs_of_type(Type,Conds,Indivs).
get_indivs_of_type(Type,[_|Conds],Indivs)
  :- get_indivs_of_type(Type,Conds,Indivs).
```

```
instantiate(_, _, [], []).
instantiate(Variable, Parameter, Indivs, Instances)
  :- bagof(Variable,member(Parameter,Indivs),Instances).

built_in_p(X,L,P)
  :- and_node(X),
     !,
     parents(X,PL),
     and_p(PL,L,P).

built_in_p(X,L,P)
  :- or_node(X),
     !,
     parents(X,PL),
     or_p(PL,L,P).

/* theorem :
   if a node X is a model of 'and', and
     it has parents U={u1,..,un} then
     p(X|C) = p(u1,...,un|C).

   Proof: trivial. :-)
*/

and_p(L,C,P)
  :- p(L,C,P).

/* theorem:
   if a node X is a model of 'or' and
     it has parents U={u1,..,un} then
     p(X|C) = p(u1;...;un|C)
            = p(u1|C) + p(u2;...;un|~u1 C)*p(~u1|C)
            = p(u1|C) + p(u2;...;un|~u1 C)*(1- p(u1|C))

    which is recursive.

   Proof: trivial. :-)
 */

or_p([],_,0).

or_p([A],C,P)
  :- p(A,C,P).

or_p([A|L],C,P)
```

```
  :- p(A,C,P1),
     or_p(L,[~A|C],P2),
     P = P1 + P2*(1-P1).

%% axiom_p(+A,+Given,-Prob)
%% - some axioms for calculating probabilities.
%% - variables the same as in p(A,Given,Prob)

%% Multiplication rule for probabilities
%% - base case: one conjunct should be computed simply.

axiom_p([A],B,P) :- !,
   p(A,B,P).

%% Multiplication rule for probabilities
%% - recursive step: find the probability of the first
%%   conjunct, and if this is non-zero, find the probability
%%   of the remainder of the conjuncts conditioned on the first
%%   by multiplication.
%% - this could be a lot smarter, I think.
%%   (by making use of conditional independence, for example)
%%   (or by using the fact that some variables may have no
%%   parents)

axiom_p([B|A],C,P) :- !,
   p(B,C,PEval),
   eval(PEval,P2),
   (P2 =:= 0, P = 0;
    p(A,[B|C],P1),
    P = P1*PEval).

%% Negation rule
%% - p(~A|C) = 1 - p(A|C)

axiom_p(~A,B,P) :- !,
   p(A,B,P1),
   P = 1-P1.

%% Definiteness rule #1
%% - p(A|A) = 1

axiom_p(A,B,p(A,B,1)) :-
   member(A,B),!.

%% Definiteness rule #2
%% - p(A|~A) = 0
```

```
axiom_p(A,B,p(A,B,0)) :-
   member(~A,B),!.


%% Definiteness rule #2 for multivalued variables

axiom_p(A=X,B,p(A,B,0)) :-
   member(A=Y,B),X \== Y,!.

%% Incompleteness rule for multivalued variables
%% - if X is not in the list of known values for
%%   A, then sum all the probabilities for all the values
%%   which A can take, and subtract this from 1.
%% - assumes sum_p(A,L,B,P) <= 1
%% - only useful if we allow incomplete specification of
%%   value lists.

axiom_p(A=X,B,P1) :-
   multivalued_node(A,L),
   \+ member(X,L),
   sum_p(A,L,B,P),
   P1 = 1 -P.

axiom_p(A=X,B,P1) :-
   multivalued_node(A,_),
   remove(~A=Y,B,BB),
   p(~A=Y,BB,P2),
   p(A=X,BB,P3),
   P1 = P3 / P2.

%% Bayes' Rule
%% - if there is evidence in the given list which influences
%%   the proposition H directly, then do bayes' rule...
%% - Why? Because we have knowledge about p(E|H) which
%%   is found in the priors.

axiom_p(H,L,P) :-
   remove(E,L,K),
   influences(H,E),
   !,
   p(E,[H|K],PEval),
   eval(PEval,P1),
   (P1 =:= 0, P = 0
   ;p(H,K,P2),
    p(E,K,P3),
```

```
    (P3 = 0,
            !, format('Error: Division by zero: ~w.~n',[p(E,K,P3)]),fail
    ;P = PEval * P2 / P3)).


%% Auxiliary predicate sum_p used in Indefiniteness rule for
%%   multivalued variables.

%*%sum_p(_,[],_,0).

sum_p(A,[V],B,P) :-
   p(A=V,B,P).

sum_p(A,[V|Values],B,P) :-
   sum_p(A,Values,B,P1),
   p(A=V,B,P2),
   P = P1 + P2.
```

B.2 expand.pl

```
/*
 * expand -- expand for multi-valued and binary variables.
 *        no independence or any other clever trick.
 *        expands propositional variables in the base manner
 *        expands multivalued variables as if X=x1 were a proposition
 */

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% expand( +A, +T, +L, +CV, -Body, -P) %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% expansion of binary proposition A on parents in T
%% Variables:
%%   +A is the variable around which we expand
%%   +T is a truth table entry for A; initially empty list!
%%   +L is a list of immediate parents of A, from which T is
%%   constructed recursively
%%   +CV is a list of conditioning variables, not necessarily
%%   parents of A
%%   -Body is the body of the rule which computes
%%   subcalculations for p(A|CV)
%%   -P is the algebraic expression which 'is' must use
%%   to compute p(A|CV) from the subcalcs done in -Body
%%
```

```
%%    p(A|T CV) = p(A|Pred T CV)p(Pred|T CV) + p(A|~Pred T CV)p(~Pred|T CV)

%% Method 1: (base case)
%% - the parents of A which were in L have been transferred to
%%   the truth table list T, leaving L = [].
%% - in this case, we need the prior for this configuration T
%% - ask Prolog for the prior, which in turn may ask the user,
%%   if there is no such prior in the database.
%% - returns the prior, as well as the Body 'true', meaning
%%   that there is no associated calculation necessary to
%%   compute the prior (it's a given quantity)

expand(A,T,[],_,pr(A,T,P,_),P).
%%expand(A,T,[],_,true,pr(A,T,P)) :-
       %%pr(A,T,P,_).

%% Method 2: (multiple valued variables)
%% - if the first variable in the truth table is a multivalued variable,
%%   handle it separately.

expand(A,T,[Z|Pred],CV,Body ,P) :-
multivalued_node(Z,Values),
        !,
expand_multi(Z,Values,A,T,Pred,CV,Body,P).

%% Method 3:
%% - to expand a binary variable, construct the truth table
%%   for it by recursively calling expand, once with the
%%   first remaining parent Z, and once with Z negated.
%% - return as the Body the sub-calculation of P(Z|CV), and
%%   the reasoning-by-cases formula which uses this value

expand(A,T,[Z|Pred],CV,Body ,(E1*PZ+E2*(1-PZ)) ) :-
expand(A,[Z|T],Pred,[Z|CV],C1,E1),
expand(A,[~Z|T],Pred,[~Z|CV],C2,E2),
simp_body((C1,C2,p(Z,CV,PZ)),Body) .

%% expand_multi(+Multi, +Values, +A, +T, +L, +CV,  -Body, -P) :-
%%   similar to expand but for multivalued variables
%%   +Multi is the multiply valued variable in question
%%   +Values is the list of values for Multi
%%   +A is the variable we are expanding around (as in expand)
%%   +T, +L, +CV, -Body, -P are all exactly as in expand
%% Basic description: for each value in +Values we find
%%   the expansion of Multi=value; kinda breadth-wise
%%   (we have to use each value here, whereas in expand
```

```
%%   it sufficed to use p(Z|CV) and calculate p(~Z|CV)
%%   as (1 - p(Z|CV)). When Z is multivalued, this doesn't
%%   work.

%% Method 1: (base case)
%% - if we have only one value left in the list of values, then
%%   call expand to continue the expansion to full depth
%% - return as Body the subcalculation call to p(Multi=Value|CV),
%%   and as the numerical expression the product of this and
%%   the expression returned by expand.

expand_multi(Multi, [Value], A, T, Pred, CV,  Body, (PM * E)) :-
       expand(A,[Multi=Value|T],Pred,[Multi=Value|CV],B,E),
simp_body((p(Multi=Value,CV,PM),B),Body).

%% Method 2:
%% - recursivelly expand A using each valuie of Multi in the Values
%%   list
%% - the body returned is the conjunction of the subcalcs
%%   and the expression is a simple addition.

expand_multi(Multi, [Value|Values], A, T, Pred, CV, Body, (E1 + E2)) :-
expand_multi(Multi,Values,A,T,Pred,CV,B1,E1),
expand_multi(Multi,[Value],A,T,Pred,CV,B2,E2),
simp_body((B1,B2), Body).

%% Rulify(+H,+B,-Rule)
%% - creates a Rule to calculate p(H|Anything) by
%%   using reasoning by cases.
%%   +H is the variable which is to be rulified
%%   +B is the list of immediate parents of H
%%   -Rule is a prolog rule which  should be asserted.

%% Method 1: multiple variables.
%% - creates a rule which handles only one value
%%   of the variable.
%% - using backtracking, all the rules will be generated
%%   (so  be sure to 'fail' after a call to rulify to get
%%   all the appropriate rules).
%% Issue: we treat each 'H=value' the same as a binary variable

rulify(H,B,(dyn_p(H=V,C,P) :- Body)) :-
multivalued_node(H,_Values),!,
%*% remove(V,Values,_),
       expand(H=V,[],B,C,R,E),
       simp_body((R, P = E), Body).
```

```
%% Method 2: Binary values
%% - creates a rule for binary variables
%% - doesn't succeed on backtracking

%*%rulify(H,B,((dyn_p(H,C,Combine) :- Body))) :-
        %*%binary_node(H),
        %*%bagof(type(Param,Type),remove(type(Param,Type),B,_),TList),
        %*%remove_1(TList,B,NewB),
        %*%expand(H,[],NewB,C,R,E),
        %*%simp_body((R, P = E), Prebody),
        %*%Body = (bagof(combine_p(Prebody,P,M),mapmember(TList,C,M),Combine),
                %*%combine_p(H,C,Combine)),
        %*%!.

rulify(H,B,((dyn_p(H,C,P) :- Body))) :-
binary_node(H),
       expand(H,[],B,C,R,E),
       simp_body((R, P = E), Body).

%% simp_body(+Old, -New)
%% - does some simplemided simplification
%%   on the rules generated by expand, or rulify.
%% - easy

simp_body((true,P),PS) :-
    !,
    simp_body(P,PS).

simp_body((P,true),PS) :-
    !,
    simp_body(P,PS).

simp_body((A;B),(AS;BS)) :-
    !,
    simp_body(A,AS),
    simp_body(B,BS).

simp_body((A,P),(AS,PS)) :-
    !,
    simp_body(A,AS),
    simp_body(P,PS).

simp_body(P,P).
```

B.3 network.pl

```
%% add_to_net(Parents,Child)
add_to_net(Parents,Child)
  :- listify(Parents,L),
     add_parents(Child,L),
     add_children(L,Child).
     %%(member(exists(X,T,V),L)
       %%-> add_parents(exists(X,T,V), [V])
       %%;  true),
     %%(member(forall(X,T,V),L)
       %%-> add_parents(forall(X,T,V), [V])
       %%;  true) .

add_parents(A,P)
  :- (retract(st_node(A,S,P1,C))
       -> append(P,P1,Pall),
          assert(st_node(A,S,Pall,C))
       ;  assert(st_node(A,[bin],P,[]))),
     assert_check(chgd(A)).

add_children([],_).
add_children([P|P1],Child)
  :- add_child(P,Child),
     add_children(P1,Child),!.

add_child(A,C)
  :- (retract(st_node(A,S,P,C1))
       -> assert(st_node(A,S,P,[C|C1]))
       ;  assert(st_node(A,[bin],[],[C]))).
     %%assert_check(chgd(A)).

remove_from_net(Parent,Child)
  :- remove_child(Parent,Child),
     remove_parent(Parent,Child).

remove_child(P,C)
  :- children(P,L),
     remove(C,L,NewL)
       -> retract(st_node(P,S,GP,L)),
          assert(st_node(P,S,GP,NewL))
       ;  format("Can't delete ~w from child list of ~w!~n", [C,P]).

remove_parent(P,C)
  :- parents(C,L),
```

```
    remove(P,L,NewL)
      -> retract(st_node(C,S,L,GC)),
         assert(st_node(C,S,NewL,GC)),
         assert_check(chgd(C))
      ;  format("Can't delete ~w from parent list of ~w!~n", [P,C]).

children(A,L)
  :- st_node(A,_,_,L).

parents(A,L)
  :- st_node(A,_,L,_).

node(A)
  :- st_node(A,_,_,_).
```

B.4 priors.pl

```
all_priors(_,[],L)
  :- asserta_list(L).

all_priors(Var,[new_prior(A,B,P)|L],L2)
  :-
     write_p(A,B,P), nl,
     write('Change this value? '),
     read(Ans),
     (prior(A,B,P),retract(prior(A,B,P))
     ;new_prior(A,B,P),retract(new_prior(A,B,P))),
     (Ans == y,
         assert_check(chgd(Var)),
     ask_prior(A,B,P1),
     all_priors(Var,L,[new_prior(A,B,P1)|L2])
     ;all_priors(Var,L,[new_prior(A,B,P)|L2])).

ask_prior(Prop,L,P)
  :- new_pr(Prop,L,P,L2),!,
     retract(new_prior(Prop,L2,P)).

ask_prior(Prop,L,P)
  :- \+ \+ write_probability(Prop,L),
     read(P).

get_priors(Prop,_,_)
```

```
  :- hidden_node(Prop),
     !.

get_priors(Prop,[],L)
  :- multivalued_node(Prop,Values),
     !,
     each((
       remove(V,Values,_),
       ask_prior(Prop=V,L,P),
       %% unnumbervars(prior(Prop=V,L,P),NP),
       asserta(prior(Prop=V,L,P)))).

get_priors(Prop,[],L) :-
     ask_prior(Prop,L,P),
     %% unnumbervars(prior(Prop,L,P),NP),
     asserta(prior(Prop,L,P)).

get_priors(Prop,[A|List],List2)
  :- multivalued_node(A,Values),
     !,
     each((remove(V,Values,_),
           get_priors(Prop,List,[A=V|List2]))).

%*%get_priors(Prop,[A|List],List2)
  %*%:- blocks(A,B2,Prop),
     %*%remove(B2,List,NewList),
     %*%get_priors(Prop,List,[A|List2]),
     %*%get_priors(Prop,NewList,[~A|List2]).

%*%get_priors(Prop,[type(_,_)|List],List2)
  %*%:- !,get_priors(Prop,List,List2).

get_priors(Prop,[A|List],List2)
  :- get_priors(Prop,List,[A|List2]),
     get_priors(Prop,List,[~A|List2]).


change_priors(A)
   :- multivalued_node(A,_),
     (setof(new_prior(A=V,B,P),prior(A=V,B,P),Bag)
     ;setof(new_prior(A=V,B,P1),new_prior(A=V,B,P1),Bag)),
     all_priors(A,Bag,[]).

change_priors(A)
  :- (setof(new_prior(A,B,P),prior(A,B,P),Bag)
     ;setof(new_prior(A,B,P1),new_prior(A,B,P1),Bag)),
```

```
    all_priors(A,Bag,[]).

assign_priors(A=V,L,P)
  :- multivalued_node(A,Values),
     member(V,Values),
     (prior(A=V,_,_),format('Priors exist for ~w. Use "change ~w".~n',[A,A])
     ;assert(new_prior(A=V,L,P))),
     !.

assign_priors(A,L,P)
  :- binary_node(A),
     (prior(A,_,_),format('Priors exist for ~w. Use "change ~w".~n',[A,A])
     ;assert(new_prior(A,L,P))),
     !.

list_priors :-
    prior(A,L,P),
    write_p(A,L,P),
    fail.

list_priors :-
    new_prior(A,L,P),
    write_p(A,L,P),
    fail.

list_priors.

pr(A,L,pr(A,L,P),L)
  :- prior(A,L2,P),
     match(L,L2),!.

new_pr(A,L,P,L2)
  :- new_prior(A,L2,P),
     match(L,L2),!.
```

B.5 combine.pl

```
p(A,C,P)
  :- ground(A,C,GroundedA),
     combination_model(A,Model),
     combine_p(Model,GroundedA,C,P).
```

```
ground([],[]).
ground([A|L],L1)
  :- ground(A,NewA),
     ground(L,NewL),
     append(NewA,NewL,L1).

ground(A,[A])
  :- atom(A).

ground(A,[A])
  :- functor(A,Name,Args),
     all_grounded(A,Args).

ground(A,GA)
  :- functor(A,Name,Args),
     first_not_grounded(A,Args,N),
     collect_individuals(A,N,New_A),
     ground(New_A,GA).

collect_individuals(Prop,Arg,List)
  :-

all_grounded(A,N)
  :- first_not_grounded(A,N,0).

first_not_grounded(A,0,0).
first_not_grounded(A,N,N)
  :- arg(N,A,X),
     var(X).
first_not_grounded(A,N,M)
  :- arg(N,A,X),
     \+ var(X),
     N1 is N-1,
     first_not_grounded(A,N1,M).
```

B.6 combined.pl

```
%% From preprocess.pl

dir_infl(A,B)
```

```
  :- consistent(A,B),
     list(A,P),
     collect_free(P,NewP,Params),
     combine(NewP,B,Params).

combine(P,B,[])
  :- !,add_to_net(P,B).

combine(P,B,Params)
  :- functor(B,Name,Args),
     gensym(Name,NewName),
     length(Params,L),
     NewArgs is Args + L,
     functor(NewB,NewName,NewArgs),
     declare_combined(Name/Args),
     declare_bin(NewName/NewArgs),
     declare_hidden(NewName/NewArgs),
     unify_args(B,Args,NewB,NewArgs,Params),
     add_to_net(P,NewB),
     add_to_net([NewB|Params],B).

unify_args(_,0,_,_,[]).

unify_args(Old,0,New,NewArgs,[type(X,_)|OtherArgs])
  :- arg(NewArgs,New,X),
     NextArg is NewArgs - 1,
     unify_args(Old,0,New,NextArg,OtherArgs).

unify_args(Old,Args,New,NewArgs,OtherArgs)
  :- arg(Args,Old,X),
     arg(Args,New,X),
     NextArg is Args - 1,
     unify_args(Old,NextArg,New,NewArgs,OtherArgs).

collect_free([],[],[])
  :- !.
collect_free(P,New,[type(X,Type)|Params])
  :- remove(type(X,Type),P,NewP),
     !,
     collect_free(NewP,New,Params).
collect_free(P,P,[]).

%% ******************
%% From probability.pl
%% ******************
```

```
%% If the node is parameterized and has 'free' parameters, then
%% we have to deal wiht it specially...
p(A,B,C)
  :- combined_p(A,B,C).

combined_p(X,L,P)
  :- combined_node(X),
     !,
     parents(X,[Hidden|Types]),
     collect_types(Hidden,Types,L,Combine),
     or_p(Combine,L,P).

collect_all_types(L,[],_,L).
collect_all_types([],_,_,[]).
collect_all_types([Hidden|Others],Types,Given,Out)
  :- collect_types(Hidden,Types,Given,Out1),
     collect_all_types(Others,Types,Given,Out2),
     append(Out1,Out2,Out).

collect_types(Hidden,_,_,[Hidden])
  :- instantiated(Hidden).

collect_types(Hidden,Types,Given,Out)
  :- parents(Hidden,P),
     mapmember(P,Given,Out1),
     remove(Out1,Given,NGiven),
     collect_types(Hidden,Types,NGiven,Out).

collect_types(Hidden,[type(Z,Type)|Types],Given,Out)
  :- bagof((type(Z,Type),Hid),(member(type(Z,Type),Given),
                Hidden = Hid),Comb),
     extract_hid(Comb,Extract),
     collect_all_types(Extract,Types,Given,Out).

instantiated(F)
  :- functor(F,_,Arity),
     instantiated(F,Arity).
instantiated(_,0).
instantiated(F,Arity)
  :- arg(Arity,F,X),
     \+ var(X),
     AA is Arity - 1,
     instantiated(F,AA).
```

B.7 consistent.pl

```
%% consistent(+A,+B)
%% - complain if the user tries any kind of stupidity
%% - +A, which may be a list, are intended to be direct parents of +B
%% - we don't want directed cycles in our network!

consistent((A,L),B)
  :- !,
     consistent(A,B),
     consistent(L,B).

consistent(A,B)
  :- \+ \+ (numbervars((A,B),0,_), A = B),
     !, format('Error: ~w cannot influence itself.~n',[A]), fail.

consistent(A,B)
  :- parents(B,L),
     member(A,L),
     !,
     format('Error: Already know ~w => ~w.~n',[A,B]),
     fail.

consistent(A,B)
  :- children(B,L),
     member(A,L),
     !,
     format('Error:Already know ~w => ~w.~n',[B,A]),
     fail.

consistent(A,B)
  :- infl(B,A),
     !,
     format('Error: ~w influences ~w already.~n',[B,A]),
     fail.

consistent(_,_).

given_inconsistent([_]) :- fail.
given_inconsistent([~A|L]) :-
  member(A,L).
given_inconsistent([A|L]) :-
  member(~A,L).
given_inconsistent([_|L]) :-
  given_inconsistent(L).
```

B.8 done.pl

```
process_and_compile
  :- each(( chgd(A),
            (multivalued_node(A,_)
               -> retract(prior(A=_,_,_)),
                  retract((dyn_p(A=_,_,_) :- _))
               ;  retract(prior(A,_,_)),
                  retract((dyn_p(A,_,_) :- _))) )),

     each(  retract(st_infl(A,B))),

     each(( st_node(A,_,_,_),
            st_node(B,_,_,_),
            infl(A,B),
            assert(st_infl(A,B)) )),

     each(( chgd(A),
            parents(A,L),
            get_priors(A,L,[]) )),
            %%build_prior_str(A,PS),
            %%assert(PS) )),

     each(( chgd(A),
            %% (binary_node(A);multivalued_node(A,_)),
            parents(A,L),
            reverse(L,RL),
            rulify(A,RL,R),
            assert(R) )),

     each(  retract(chgd(_))),

     each(  retract(new_prior(_,_,_))).
```

B.9 infl.pl

```
%% dir_infl(A,(B,C))
  %% :- !,
     %% dir_infl(A,B),
     %% dir_infl(A,C).
```

```
%% dir_infl(A,B)
  %% :- consistent(A,B),
     %% !,add_to_net(A,B).

dir_infl(A,B)
  :- consistent(A,B),
     !,add_to_net(A,B).

delete_influences([],_)
  :- !.

delete_influences(_,[])
  :- !.

delete_influences([A|L],B)
  :- delete_infl(A,B),
     delete_influences(L,B).

delete_influences(A,[B|L])
  :- delete_infl(A,B),
     delete_influences(A,L).

delete_infl(A,B)
  :- node(A),
     node(B),
     remove_from_net(A,B),
     assert_check(chgd(B)).

infl(A,~B) :-
   infl(A,B).

infl(A=_,B) :-
   infl(A,B).

infl(A,B=_) :-
   infl(A,B).

infl(A,B) :-
   children(A,L),
   member(B,L).

infl(A,B) :-
   children(A,L),
   member(C,L),
   infl(C,B).
```

```
influences(A,~B) :-
   influences(A,B).

influences(A=_,B) :-
   influences(A,B).

influences(A,B=_) :-
   influences(A,B).

%%influences(A,B) :-
   %%children(A,L),
   %%member(B,L).

%%influences(A,B) :-
   %%children(A,L),
   %%member(C,L),
   %%influences(C,B).

influences(A,B) :-
   st_infl(A,B).
```