A logical Framework for Model Based Diagnosis with Probabilistic Search

by

Peter Macdonald

M.Sc., McMaster University, 1979
B.Sc., Queens University, 1976

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in the School
of
Computing Science

SIMON FRASER UNIVERSITY
November 1992

Approval

Name : Peter Macdonald

Degree : Master of Science

Title of Thesis : A Logical Framework for Model Based Diagnosis with Probabilistic Search

Examining Committee : Dr. Veronica Dahl, Chairperson

Dr. William S. Havens
Senior Supervisor

Dr. John D. Jones
Supervisor

Dr. Jaiwei Han
Supervisor

Dr. David Poole
External Examiner

Nov. 16, 1992
Date Approved

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

A Logical Framework for Model Based Diagnosis with Probabilistic Search.

Author: _____
(signature)

Peter D. Macdonald
(name)

November 16, 1992
(date)

Abstract

Diagnosis involves finding explanations for the observed behavior of an often complex physical system. Historically, approaches to diagnosis have differed, depending upon the existence, or not, of a theoretical model of system behavior. Problems in medical diagnosis are characterized by a highly incomplete knowledge of how the human body operates. In contrast, the behavior of engineered systems is typically described by a sophisticated theory based on established scientific principles.

Early expert systems developed for medical diagnosis model the reasoning strategies of human experts, rather than human physiology. These systems feature ad-hoc uncertainty calculi, and shallow, heuristic domain knowledge. Alternatively, recent medical diagnosis systems utilize Bayesian Belief Networks to represent statistical, causal models of the human body and disease.

Model-based diagnosis systems address engineering domains and incorporate engineering models of system behavior. The lack of a generalized uncertainty calculus results in difficulties in ranking diagnoses, and in diagnosing faults which are not covered by the theoretical model.

We propose a generalized model-based approach to diagnosis which reconciles statistical and deterministic modeling. We argue that even with an engineered system, particularly one which is faulty, there are aspects of system behavior which are incompletely understood. We require a representation which is equally good at representing both well-understood and partially-understood aspects of system behavior.

We present a way of representing Bayesian Belief networks as logic programs with extra-logic probability annotations. In doing so we extend the dual procedural and declarative semantics of the annotated logic programs. We also present an architecture for diagnostic inferencing. The proposed architecture features a

combination of best-first search and intelligent backtracking and generates diagnoses in order of decreasing likelihood. The architecture explicitly supports the comparison of multiple diagnoses with incremental observations, a process typical of diagnostic problem solving.

Acknowledgements

I would like to thank my wife, Robin, for her love and support during my stint back at school. I also wish to thank my three children Graham, Hugh and Ian for the constant delight and encouragement they have been to me during this period. I would like especially to thank my senior supervisor, Dr. Bill Havens, for his encouragement, guidance and many fruitful discussions. I am also grateful to Dr. John Jones for his curiosity, his always open ear and fresh viewpoints.

I wish to thank Charlie Hunter of the diagnosis group for his perspective on the practical difficulties in modeling real complex engineering systems. I am grateful to Miron Cuperman of the Expert Systems Lab for his patience and cooperation in answering my many questions on Echidna.

I would like to acknowledge the financial support which I obtained from MacDonald Dettwiler, the Science Council of British Columbia, and the Center for Systems Science. I would like especially to thank Dr. Bruce Sharpe, Dr. John MacDonald and Mr. Pat Brownsword of MacDonald Dettwiler for their ongoing interest and support of my research.

Table of Contents

List of Figures

Chapter 1
Introduction

Humans are often called upon to diagnose a complex physical system exhibiting abnormal behavior. Such problems are inherently difficult. Complex systems fail in many different ways, not all of which are well understood. There may be several possible explanations for an observed set of symptoms, with some explanations being much more likely than others. In spite of these difficulties, human experts such as physicians and mechanics are remarkably adept at diagnosing problems in diverse application domains.

In recent years diagnosis has been studied as a problem in Artificial Intelligence. Diagnostic Expert Systems have been developed which model the knowledge intensive reasoning of human experts in solving diagnosis problems. MYCIN [Buchanan,84] is an early example of an Expert System for medical diagnosis. While relatively successful, the limitations of these early systems are readily apparent. MYCIN models knowledge as if-then rules of the form *if symptom then disease*. The uncertain nature of diagnostic reasoning is modeled by associating ad-hoc *certainty factors* with these rules which are combined according to the way in which the rules are combined in a line of reasoning. The ad-hoc, procedural nature of MYCIN certainty factors can lead to incorrect, non-intuitive results as well as problem solving inflexibilities [Pearl,88]. For example, MYCIN's certainty factors do not combine properly for correlated sources of evidence. Also, MYCIN only supports diagnostic queries. It can reason from symptom to disease but not vice-versa.

The *if symptom then disease* form of MYCIN's rules leads to a *shallow* knowledge base. Rather than reflecting an in-depth domain understanding, these rules model how experts solve diagnosis problems in a particular application domain. This perspective results

in shallow, surface knowledge which is immodular, and difficult to extend. Domain knowledge which, in principle, can be applied to many different domain problems, is instead tied to a particular usage.

In recent years Bayesian Belief Networks [Pearl,88] have been applied to medical diagnosis. Bayesian Belief Networks remedy many of the problems associated with MYCIN. Bayesian Belief Networks model causal (cause → symptom) relationships between diseases and their symptoms in a modular and intuitively appealing way. Moreover, this is accomplished without sacrificing the formal semantics of Bayesian probabilities. However, Bayesian Belief Networks are essentially propositional representations. This represents a modeling limitation, particularly in Engineering domains, where we often have generative theories capable of deriving complex system behaviors. Bayesian Belief Networks are well suited to medical diagnosis where we do not have a well understood theory of the human body and disease.

Recent research in Model Based Diagnosis has focused on domains where the physical system is an engineered artifact such as a circuit, or a mechanical system. Model Based Diagnosis incorporates an elaborate model of how the physical system ought to work. A complex system is represented as many interconnected components with localized, well understood modes of operation. Each component includes a model of its normal healthy behavior, as well as one or more faulty behavioral states. The global behavior of the system can be derived from the net effect of local behavioral interactions between connected components. The problem of enumerating global disease/symptom pairs has been replaced with the problem of modeling the localized behavior of components, and their inter-connection.

Formalizations of model based diagnosis [Reiter,87a], [de Kleer,90a], [Selman,90] are based on predicate calculus and are closely related to formalizations of default logic. Like Bayesian Belief Network applications, the model based approach is based on causal,

modular, and intuitively appealing representations. However, unlike Bayesian Belief Networks they do not incorporate a generalized uncertainty calculus. As a result, model based diagnosis systems often resort to implicit and explicit assumptions in order to distinguish between likely and unlikely diagnoses. Often, for example, it is assumed that single component failures are more likely than multiple component failures. De Kleer's SHERLOCK system [de Kleer,89a] assumes that components fail independently. Perhaps more seriously, model based systems implicitly assume that the inter-connectivity of components is as given by the model. This rules out the possibility of shorted circuits in a circuit diagnosis application. As a result of such assumptions, model based diagnosis systems loose modelling completeness. There are some states of the actual physical system which are not represented.

In this thesis we posit that the partitioning of diagnosis problems according to whether the application domain is well understood or not is superficial. Diagnosis domains do not either have a well developed theory or none at all. Rather most applications contain aspects of both. An Engineering system may have a well understood theory provided that certain conditions hold. If these conditions are not met, then the system may exhibit behavior which can only be modeled statistically.

We propose a way of representing Bayesian Belief Networks as Horn clause logic programs which are annotated with extra-logic probabilities. We refer to such programs as Bayesian programs. This integrated representation offers both the benefits of Bayesian Belief Networks and predicate-calculus-based formulations of model based diagnosis. It offers generalized probabilistic ranking of diagnoses, together with the expressivity of predicate calculus for those parts of the model which have a well founded theory.

We also propose an architecture for problem solving with Bayesian programs. At first glance Bayesian programs look very much like Prolog programs. However, we argue that the

requirements of diagnostic problem solving make for very different architectural tradeoffs.

The adoption of causal rules strengthens the importance of search. Whereas MYCIN reasons directly from symptoms to disease, we search among a space of possible causes to find those that generate the observed symptoms. In complex systems containing many components, the search space becomes very large. It is essential that we adopt a strategy for moving through this search space efficiently.

Recent intelligent backtracking implementations for Prolog [Bruynooghe,84], [Cox,84], [Drakos,88], [Havens,91], [You,89] use Reason Maintenance System (RMS) techniques to improve the efficiency of search. These techniques are based on the early identification of search space branches which cannot possibly lead to a solution. In the case of diagnosis problems, it is not sufficient to find *any* diagnosis. Rather, it is important that we find the most likely diagnosis first. We therefore propose an integration of best-first probabilistic search with intelligent backtracking.

Another characteristic of diagnostic problem solving is that typically, diagnosis systems are called upon to recommend a course of action based on a set of highly likely diagnoses. A diagnosis system may recommend that a particular measurement be taken in order to discriminate between several highly likely diagnostic candidates. Typically, depending upon the result of this measurement, several candidates are eliminated from further consideration. Here we see the need to maintain multiple solutions to a diagnostic query, and to support incremental, interactive comparisons involving these solutions. These capabilities are in marked contrast to most Prolog-like systems which generate the next solution by destroying the first.

This thesis is organized as follows. Chapter 2 discusses diagnosis as an example of the more general form of reasoning known as abduction. We present a survey of recent research related to the above objectives. Chapter 3 describes a new knowledge

representation for abductive problem solving. The declarative and procedural semantics of this representation are developed as an extension to conventional logic programs. In chapter 4 we discuss an inferencing architecture which addresses the special requirements of abductive problem solving. We finish with chapter 5 which draws conclusions.

Chapter 2
Recent Research

In this chapter we view diagnosis as an example of the more general class of problems which are solved through abductive reasoning. We discuss what abduction is and survey related research.

2.1. What is Abduction

Abduction is the reasoning process by which one generates explanations for observations. In medical diagnosis we may observe that John has a sore throat. Abductive reasoning enables one to propose the hypothetical explanation that John has a cold which is causing his sore throat. We can think of abduction as an unsound rule of inference which resembles a sort of inverse deduction.

$$\begin{array}{l} a \rightarrow b \\ \underline{b\ \ } \\ a \end{array}$$

Whereas deduction leads to unequivocal assertions, abduction leads to equivocal assertions. That John has a cold is essentially a guess. It is a hypothesis which, if it *were* unequivocally true, would deductively lead to the observed sore throat. Here we see that abductive and deductive reasoning often work together with deduction acting as a partial justification for abductive guesses.

Of course there are many other possible reasons for John's sore throat, some of which are more likely than others. Typically, in proposing that John has a cold, we have implicitly selected this explanation on the basis that it is *more likely* than other equally possible explanations. The ranking and comparison of multiple possible explanations is a common feature of problems which require abduction.

The equivocal nature of assertions reached through abduction leads to an inherent non-monotonicity that does not occur in deductive problem solving. We may have to retract previous assertions in the light of new evidence. If we were to discover that John has just undergone an operation to remove his tonsils, we would typically retract the previous explanation in favour of a now much more likely explanation, namely, that John's sore throat was caused by the operation which lead to swelling and damaged tissue.

Charniak and McDermott point out that abduction is properly related to causation [Charniak,87]. They cite the example:

patient in ward 5 → patient has cancer
patient has cancer
patient in ward 5

If abduction generates explanations which explain observations then our unsound rule of inference falls short as a definition of abduction. The fact that a patient is in ward 5, does not explain why he or she has cancer. Charniak and McDermott offer an alternative rule of inference - one which is based on causality rather than logical implication.

a *causes* b
b
a

The difficulty with this definition is that whereas logical implication is well understood, the human notion of causality has proven difficult to define precisely, in spite of its intuitive appeal. Model based reasoning adopts causality as a primitive notion and incorporates it into the way in which knowledge is modeled. Model based systems use rules of the form *cause* → *effect*. which makes the two rules of inference proposed above, equivalent.

2.2. Approaches to Abduction

AI researchers have developed different approaches to the problem of abductive reasoning. Several researchers have proposed nonmonotonic extensions to classical logic. These logics are referred to as nonmonotonic logics. Reiter's Default Logic [Reiter,80], McDermott and Doyle's *nonmonotonic logic* [McDermott,80], and Poole's framework for default reasoning [Poole,88] are based on the notion of logical consistency. The essential idea is to interpret the pattern "in the absence of information to the contrary, assume A", as "if A can be consistently assumed, then do so". Recent work in model based diagnosis exploits this idea. Components are assumed to operate normally unless this leads to predictions which conflict with observations. If necessary, assumptions are retracted until inconsistencies between observed and predicted behavior is removed. This forms the basis for several recent model based diagnosis systems [de Kleer,87], [Genesereth,84], [Reiter,87a], [Poole,87], [Davis,84], [de Kleer, 89a].

McCarthy's circumscription [McCarthy,80] is a form of non-monotonic reasoning based on the notion of selecting interpretations which minimize the number of accepted abnormalities or exceptions.

There are computational difficulties associated with both circumscription and default logics. However, for our purposes, a more important deficiency is the lack of any ranking criteria. While default logics enumerate a space of possible abductive explanations, the logics are not able to rank which explanations are better than others. This is critical as some abductive explanations are intuitively far superior to others which are equally well supported by the nonmonotonic logic. Diagnostic systems deal with this requirement in different ways. Reiter [Reiter,87a] generates diagnoses in order of the increasing cardinality of the set of faulty components. This is a generalization of the single-fault assumption often used in conventional MYCIN-like diagnosis systems [Charniak,87]. De Kleer [de Kleer,87], [de Kleer, 89a], [de Kleer,90b] uses Bayesian analysis together with simplifying independence assumptions. This leads to

the more general question as to whether we are looking in the wrong place in our quest for an abductive reasoning formalism. Perhaps probabilistic reasoning should be given explicit consideration, rather than treated as an afterthought.

Bayesian probability theory enables us to reason with uncertainty in a very general and consistent way. Consider the joint probability distribution, P(D1,D2,S1,S2). From a knowledge representation point of view, it has a well defined denotational semantics. The frequency with which D1=d1,D2=d2,S1=s1,S2=s2 occurs is P(d1,d2,s1,s2). Another property of joint probability distributions is global coherence. Global coherence refers to the property that individual pieces of knowledge making up a knowledge base sum to a complete understanding of a whole system. In this case our pieces of knowledge are probability values. The complete set of these pieces of knowledge supports generalized global queries. If we consider D1,D2 to represent diseases, and S1,S2 symptoms, then we can perform abductive calculations such as:

$$P(D1|\,S1) = \frac{\sum_{S2,D2} P(D1,D2,S1,S2)}{\sum_{D1,D2,S2} P(D1,D2,S1,S2)}$$

We are equally able to calculate deductive conditional probabilities such as P(S1=s1|D1=yes).

Unfortunately, the joint probability distribution representation is extremely immodular. Each piece of knowledge is a statement about the entire system. We essentially have a single system with a very large set of attributes. If we change any probability value, we must reexamine the entire system in order to ensure that the distribution remains normalized. As a knowledge representation, this deficiency is fatal. Such a representation is extremely difficult to build and to maintain.

In the 70's and early 80's researchers developed Expert Systems for medical diagnosis of which the much written about MYCIN is a notable example [Buchanan,84], [Charniak,87]. These systems incorporate ad-hoc uncertainty calculi so as to avoid the modularity difficulties associated with probability representations. The basic strategy is to exploit the syntactic modularity of rule based representations, namely, the property whereby rules can be invoked incrementally without regard for each other. Unfortunately, in doing so, desirable properties of Bayesian probabilities, such as global coherence, and denotational semantics are sacrificed.

In [Pearl,88], Pearl refers to systems which use MYCIN-like uncertainty calculi as *extensional* systems. He contrasts the procedural semantics of extensional systems with the denotational semantics of *intensional* systems such as Bayesian probability theory. In his words, the propositional rule $A \rightarrow B$, with associated certainty factor m means: "If you see the certainty of A undergoing a change DA, then regardless of what other things the knowledge base contains and regardless of how DA was triggered, you are given an unqualified license to modify the current certainty of B by some amount DB, which may depend on m, on DA, and on the current certainty of B". In contrast, in the Bayesian formalism the rule can be associated with the conditional probability $P(B|A) = m$ which states that of all world's for which A is true, those for which B is also true constitute an m percent majority. Extensional systems mean what they allow one to do. Intensional systems make statements about a domain.

While extensional semantics results in localized, syntactic manipulations, it creates other difficulties. Consider the rules $S1 \rightarrow D$ with associated certainty factor m1, and $S2 \rightarrow D$ with associated certainty factor m2. In MYCIN certainty factors are real numbers between -1 and 1. A certainty factor of -1 indicates a complete lack of believe; a value of 1 represents complete certainty. If both m1 and m2 are positive, and both rules are executed, MYCIN calculates the certainty factor for D as $CF(D) = m1 + m2(1-m1)$. The effect of the

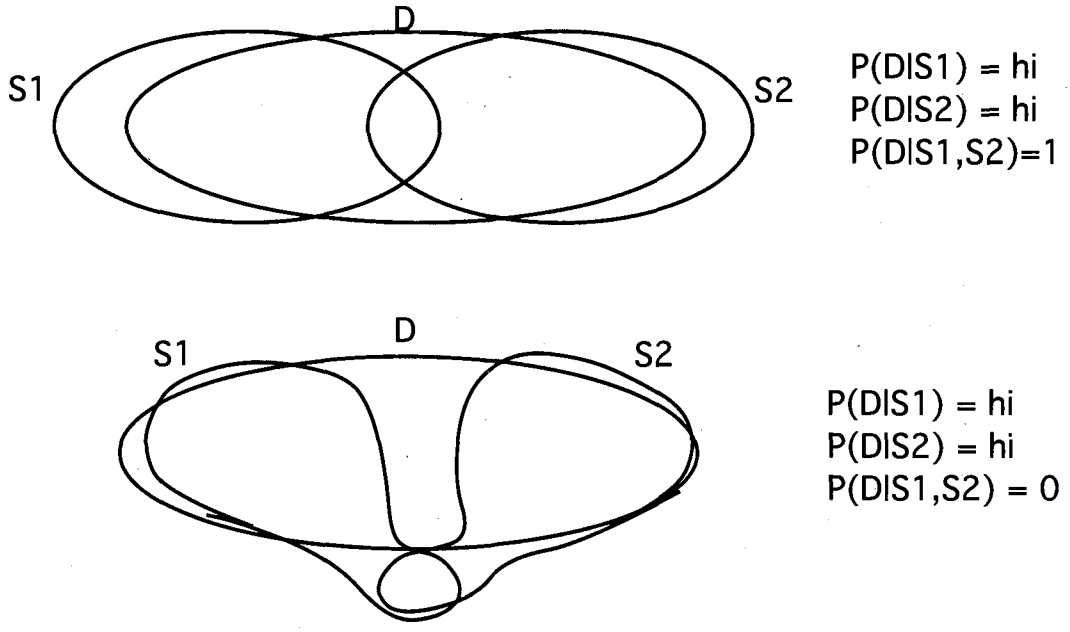


Figure 2.1. P(D|S1,S2) is not constrained by P(D|S1) and P(D|S2)

second rule invocation is always to increase the certainty of D resulting from the first rule invocation.

Under a Bayesian formalism the assertions P(D|S1)=m1 and P(D|S2)=m2 do not constitute sufficient information to determine P(D|S1,S2). This can be seen in the Venn diagram representations of figure 2.1. In both Venn diagrams P(D|S1) and P(D|S2) are close to one, yet in the first case P(D|S1,S2) = 1, and in the second case P(D|S1,S2) = 0. The value of P(D|S1,S2) is not constrained in any way by a knowledge of P(D|S1) and P(D|S2). Knowledge about the relationship between sets S1 and D, and between S2 and D, is inadequate when it comes to making further statements about the nature of $S1 \cap S2$ and its relationship with D.

Extensional semantics provides a certainty factor calculation mechanism which parallels the conventional, highly localized rule execution mechanism. Unfortunately, certainty factors do not combine in such a way as to reflect global interrelationships in our models. Global coherence has been sacrificed for syntactic

modularity. Improper treatment of correlated sources of evidence [Pearl,88] and other difficulties can be attributed to this sacrifice.

The earlier definition of extensional rule semantics is explicitly tied to the direction of reasoning. Extensional semantics propagates certainty factors from rule antecedent to rule consequent. It does not provide a similar license to reason in reverse. As a result, MYCIN-like systems support only a single direction of reasoning. Since MYCIN reasons from symptoms to disease, it uses rules of the form *effect* → *cause*. Moreover, in order to prevent certainty factor amplification cycles, it must not have *any* rules of the form *cause* → *effect*. This is at odds with our treatment of abduction as a rule of inference. As such, a single rule of the form *cause* → *effect* should support either forward reasoning (deduction) or backward reasoning (abduction).

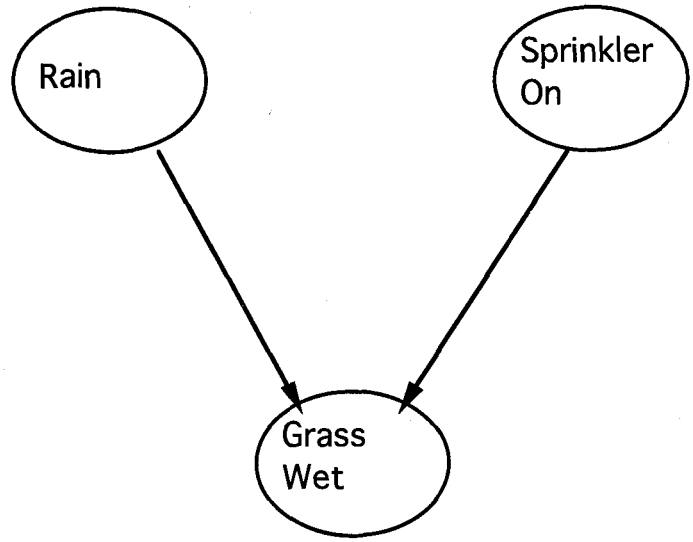
The procedural nature of MYCIN-like reasoning binds the form of its knowledge representation to the particular problem being solved. Extensional systems can compute a diagnosis, but cannot predict additional symptoms which might arise from that diagnosis. From a purely syntactic point of view, MYCIN appears to be doing deduction. From a semantic point of view it is clearly a form of abduction. It is probably best described as procedural abduction with no concomitant deduction at all. Apart from preventing the same knowledge base from being used for more than one purpose, the unidirectional limitation of extensional systems compromises diagnostic problem solving. In the case of a suspected disease, it is natural to predict additional symptoms likely to be present if that disease were in fact true. Whether or not these predicted symptoms are actually present can be used to corroborate or discredit the original explanation.

We have seen the sacrifices that extensional systems make in order to achieve the syntactic modularity of rule based systems. However, modularity can also be looked from a semantic point of view. From this perspective we are concerned with examining whether rules tie together remotely related concepts and entities. In

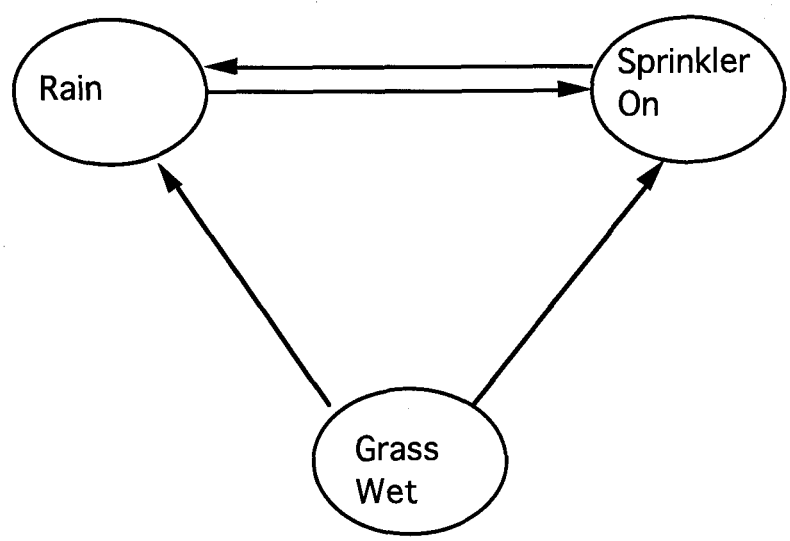
this regard it is prehaps ironic to discover that while MYCIN-like expert systems provide syntactic modularity, they suffer from poor semantic level modularity. Figure 2.2 illustrates the problem. In the example, either *rain* or *sprinkler_on* can result in *grass_wet*. In a causal representation where we represent rules in the form *cause* → *effect*, this situation can be represented quite naturally as *rain* → *grass_wet* and *sprinkler* → *grass_wet*. In a MYCIN-like system we are required to choose a rule form which corresponds to the direction of our reasoning. In this case our reasoning is abductive, that is, from effect to cause. If we adopt a non-causal representation of the form *effect* → *cause*, then we must use rules such as *grass_wet*, ¬ *sprinkler_on* → *rain* and *grass_wet*, ¬*rain* → *sprinkler_on*, These rules resort to exceptions to tie together concepts like *rain* and *sprinkler_on* which are only remotely related to each other.

A precise definition of causality has proven elusive [Iwasaki,86a], [Iwasaki,86b], [de Kleer,86e], [Forbus,88b]. Pearl [Pearl,88] speculates that prehaps the human notion of causality is tied to an unconscious selection of inherently parsimonious memory representations as a way of structuring and simplifying our understanding of the world within which we live. The unidirectional reasoning of MYCIN-like systems rules out causal knowledge representations, thereby eliminating from consideration, intuitively appealing and modular representations.

Semantic level modularity is crucial to complex model building. From this perspective, non-causal representations represent a serious modeling handicap. Expert systems such as MYCIN are often referred to as having *shallow* knowledge bases. They reflect associative rather than causal relationships, are incomplete, and are based on the heuristic experience of domain experts rather than on a problem independent understanding of the domain itself. The knowledge bases consist of highly inter-related problem dependent rules and are difficult to maintain and extend.



(a) Causal Form



(b) Non-causal Form

Figure 2.2. Causal representations link related concepts only

In contrast, Model Based Diagnosis systems adopt the causal, object-based representations. These systems are referred to as having *deep* knowledge. The resulting modularity and parsimony enables relatively complex systems to be incrementally developed

and maintained. These systems allow for both diagnostic and predictive reasoning.

In recent years Bayesian Belief Networks (from here on we refer to Bayesian Belief Networks simply as Bayesian Networks) have been proposed [Pearl,88] as an alternate knowledge representation for uncertain reasoning. Bayesian Networks are intensional systems and are based on Bayesian probability theory. The motivation is for a representation which is modular, as well as having global coherence and denotational semantics.

Consider the joint probability distribution P(A,B,C,D,E). We can use the chain rule of probability theory, and a total ordering of the variables, to express the joint probability distribution as a chain of multiplications. In this example the variables are ordered alphabetically:

$$P(A,B,C,D,E) = P(A)\ P(B|A)\ P(C|A,B)\ P(D|A,B,C)\ P(E|A,B,C,D)$$

Expressing the joint probability distribution in this form enables one to exploit independence relationships in the semantic domain in simplifying this expression. For example, the fact that B does not depend on A is expressed as $P(B|A) = P(B)$. Reducing the worlds under consideration to those for which A is true, does not change the percentage for which B is also true. Typically, there are many such independence relationships. For each conditional probability in the above expression, we determine the smallest set of conditioning variables which directly influences the conditioned variable. Continuing our example, we assume the following independence relationships:

$$P(B|A) = P(B)$$
$$P(D|A,B,C) = P(D|C)$$
$$P(E|A,B,C,D) = P(E|A)$$

This simplifies the expression for the joint probability distribution:

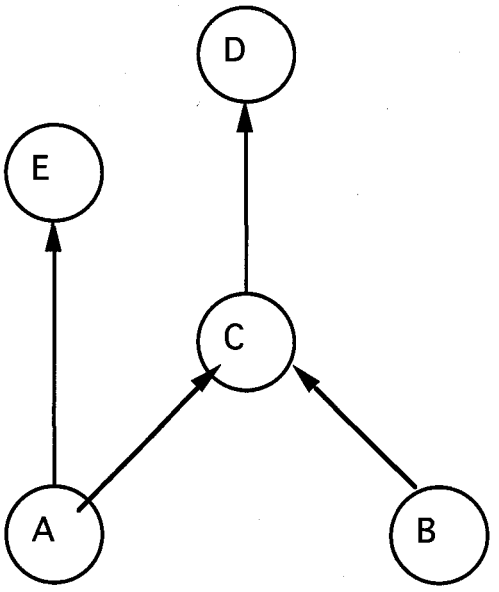


Figure 2.3. A Bayesian Network

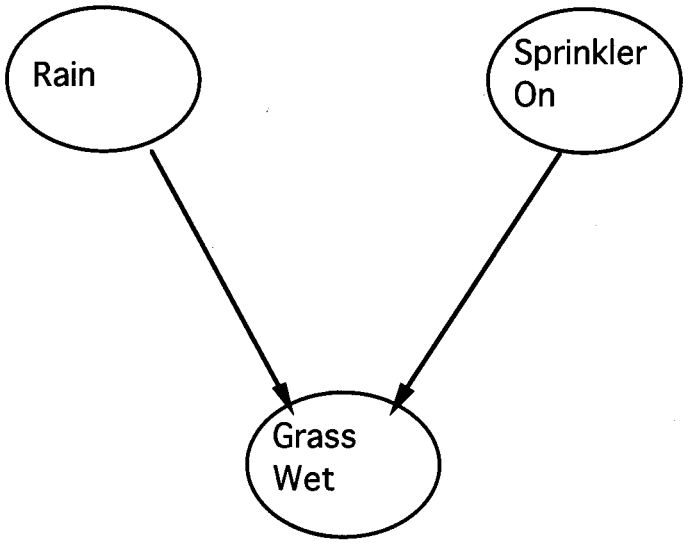
$$P(A,B,C,D,E) = P(A)\ P(B)\ P(C \mid A,B)\ P(D \mid C)\ P(E \mid A)$$

A Bayesian Network represents expressions of this sort as a Directed Acyclic Graph (DAG). Figure 2.3 shows a Bayesian Network representation of the above expression. Each variable is represented as a node. Terms such as P(C|A,B) can be interpreted qualitatively as "C depends on A and B." We represent this as directed edges from A and B to C. There are directed edges from each conditioning variable to the conditioned variable for each conditional probability in the above expression. Bayesian Networks are given quantitative meaning through the association of a set of probabilities at each node. Each probability term in the original expression is associated with the variable node for its conditioned variable. For example, we associate P(C|A,B) with variable node C. If A,B,C have domains D_A, D_B, D_C respectively, then node C stores $|D_A| \times |D_B| \times |D_C|$ conditional probabilities. Root nodes, for which there are no incoming edges, store unconditional probabilities.

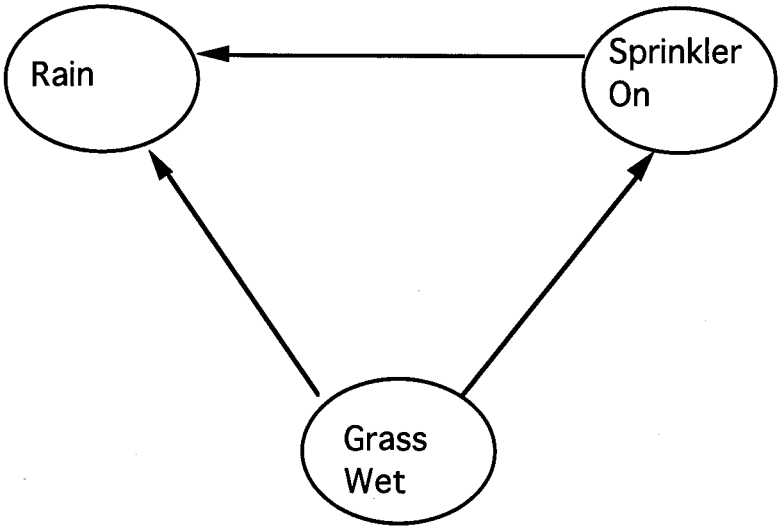
The Bayesian Network DAG represents a partial ordering of the variables which is consistent with the original total ordering. After the Bayesian Network has been formed, the original total ordering can be disregarded. Any total ordering consistent with the final partial ordering would have lead to the same Bayesian Network [Pearl,88].

Although a Bayesian Network Representation can be formed regardless of which total variable ordering is initially adopted, the modularity of the final DAG is very different for different orderings. For example, figure 2.4 illustrates Bayesian Network representations for the earlier example of figure 2.2. The first case represents a causal ordering of the variables. The second case reverses the ordering used in the first case. As in our earlier discussion, the relatively better modularity of the second case suggests that we adopt causal Bayesian Network representations. This is reinforced by the intuitive nature of causal representations, and the availability of conditional probabilities. We are far more likely to have probabilities for the first case than for the second. In fact, although the preceding discussion has focused on decomposing an existing joint probability distribution into a Bayesian Network, in practise we are more likely to incrementally build Bayesian Networks from an understanding of causal dependencies among variables. The global distribution is synthesized from an understanding of local causation.

Bayesian Networks achieve semantic level modularity by making explicit use of independence relationships in the underlying probability distribution. Unlike extensional systems, this is not achieved at the expense of denotational semantics and global cohesiveness. Bayesian Networks explicitly define the way in which each variable directly depends on other variables. The way in which variables indirectly depend on each other under different evidential conditions, can be derived through graph traversal algorithms [Pearl,88]. Such procedures enables a Bayesian Network to act as a graph traversing Inference Engine capable of answering generalized queries, based on the underlying probability distribution.



(a) Causal Form



(b) Non-causal Form

Figure 2.4. Causal Bayesian Networks are modular

Pearl [Pearl,88] presents a distributed message-based approach to belief updating and revision. In doing so he achieves syntactic modularity not unlike that of rule based systems. Unfortunately, his approach is limited to *singly connected* networks. A singly connected

network is a network for which no more than one path exists between any two nodes. As we encountered earlier with extensional systems, local syntactic reasoning mechanisms limit our ability to maintain global correctness under all circumstances. In the case of Bayesian Networks, however, a well defined semantics provides a basis for which to develop more general meaning preserving algorithms. One approach results in approximate answers through the use of stochastic simulation techniques [Henrion,88]. Other approaches use search and graph traversal based reasoning steps to derive correct answers to Bayesian queries. Shachter [Shachter,86], [Shachter,88] describes an approach based on transforming Bayesian networks through a series of arc-reversal and node removal reductions. Jensen et. al. [Jensen,89a], [Jensen,89b] outline a scheme which makes use of a secondary structure known as a junction tree for propagating belief information in multiply connected networks. D'Ambrosio et. al. [D'Ambrosio,90] describe a goal-driven approach which is incremental with respect to both queries and observations. Shimony and Charniak [Shimony,90] describe an algorithm for finding maximum a-posteriori (MAP) assignments of node values through the use of an intermediate boolean network and best-first search techniques. Henrion [Henrion,90], [Henrion,91] investigates the use of admissability heuristics for pruning best-first search trees.

There are strong similarities between Model Based Diagnosis, and recent Bayesian Network based diagnosis which, out of convenience, we refer to as Network Based Diagnosis. Both approaches represent knowledge in the form *cause* $\rightarrow$ *effect*. Predicate calculus based formalizations of Model Based Diagnosis [Reiter,87a], [de Kleer,90a] provide well defined denotational semantics although this semantics does not extend to the ranking of diagnoses. Both Model Based Diagnosis and Network Based Diagnosis offer a well defined enumeration of possible system states. Under Model Based Diagnosis, global system behavior arises out of the way in which components encapsulating local behavior, are physically connected. Under Network Based Diagnosis, global assignments arise out of the way in

which nodes encapsulating local conditional probabilities, are connected.

At first glance, the *object* ontology which underlies Model Based Diagnosis would seem to represent a fundamental difference. The Model Based world consists of objects or components which interact according to how they are physically connected. Whereas components encapsulate deterministic behavior, Bayesian Network nodes encapsulate statistical behavior. Whereas components are physically connected, Bayesian Network nodes are causally connected. However, recent research [Dechter,91] suggests that object abstractions arise naturally out of causal interpretations of the world, particularly in those cases where we have deterministic models of behavior. From this perspective, causal representations represent a generalization of object-based representations. The object ontology of Model Based Diagnosis simply reflects the relatively more complete theories associated with engineering domains as compared with medical diagnosis.

The are however, important distinctions between the two approaches. As mentioned above Model Based Diagnosis formalisms do not include a general uncertainty calculus. As a result, diagnoses cannot be ranked except through the use of simplified probabilistic assumptions or heuristics. On the other hand, Model Based formalisms offer the expressivity of predicate calculus over the propositional nature of Bayesian Networks.

We argue that the partitioning of diagnosis problems into different problem classes, according to whether the problem domain is well understood or not, is superficial. In the general case we can expect to encounter problem domains for which some aspects are well understood and other aspects are not well understood. In automobile diagnosis, while many properties can be derived from a well understood theory, there may be other symptoms such as "rough-idle" which can only be statistically modelled. This point is related to the issue of completeness in Model Based Diagnosis. Davis and Hamscher [Davis,88], [Hamscher,90b] point out that a purely

Model Based System cannot account for all possible failure modes of a device since it is based on the assumption that the model correctly reflects the physical structure of the artifact undergoing diagnosis. Indeed, there are implicit assumptions in any abstract model of reality that may or may not be met in the case of a particular artifact. Whereas predicate logic representations are good at representing theoretical component behavior given a particular framework of assumptions, probabilities are needed to summarize possibilities which lie outside the assumptions sanctioning the theory.

Recent research interest focuses on the possibility of overcoming technical difficulties found in logic through the use of probabilities. McDermott [McDermott,87] and Watanabe [Watanabe,87] discuss the limitations of AI systems based fundamentally on deductive logic, in a world where abduction and induction play dominant roles in commonsense and scientific reasoning. Cheeseman [Cheeseman,88] suggests that Bayesian inferencing can overcome many of logic's technical difficulties. Bacchus [Bacchus,90] argues that it is important to distinguish between statistical and subjective degree of belief interpretations for probability. His logic LP is an extension of first-order logic for reasoning with statistical knowledge. Statistical statements in LP specify probability distributions over a domain of discourse. LP includes random designators which are treated as random variables, rather than as conventional universally quantified variables.

An alternative approach to integrating logic and probability is to extend the possible worlds semantics of first-order logic to a probability distribution over a set of possible worlds. Probabilities assigned to sentences are treated as part of each sentence's assertion, and must hold for universally quantified variables of the sentence. There is recent research along these lines. Charniak and Shimony [Charniak,90] define a boolean belief network representation for a propositional rule system and assign probabilistic semantics to cost-based abduction. In [Poole,91] Poole proposes extensions to Horn

clause logic and represents Bayesian Networks as extended logic programs. In this thesis we propose a similar representational mechanism. The motivation is to combine the representational advantages of predicate logic with those of Bayesian Networks, thereby supporting a generalized approach to diagnostic problem solving.

2.3. Architectures for Abduction

In [Nii,89] Nii distinguishes between two problem solving paradigms, one based on *search* and the other on *recognition*. In the search-based paradigm a problem solver makes each problem solving choice in the context of a well enumerated set of possible choices. The search paradigm is referred to as a weak method since virtually all programs that employ search use weak knowledge to evaluate which of the available choices is best. In the recognition paradigm a problem solver *matches* the current problem state with a piece of knowledge that can be applied in this situation. As Nii puts it:

"At any particular computational state, instead of generating and evaluating the possible next states, a recognition system simply knows what the next state should be."

The recognition paradigm is referred to as knowledge rich. It relies heavily on task specific criteria to *know* what the next state should be. Most current expert systems are recognition systems.

Whereas MYCIN is a recognition system, Model and Network systems are search-based. As a result we are interested in architectures which efficiently support search, particularly abductive, or nonmonotonic search. This leads to an examination of Reason Maintenance Systems, and the architectures which use them.

A Reason Maintenance System (RMS) and a Problem Solver are two components which together form an overall reasoning system.

The role of the RMS is to keep a record of propositional assertions arrived at by the Problem Solver during problem solving. These assertions are referred to as *datums*. The RMS records both the datums themselves, and the way in which these datums are justified by other datums and/or assumptions. By recording datum dependencies the RMS supports the Problem Solver in its use of retractable assumptions. Should a line of reasoning lead to an inconsistency, the datum dependencies identify which set of assumptions lead to this inconsistency. Conjunctive sets of inconsistent assumptions are referred to as *nogoods*. In order to proceed, at least one of a newly discovered nogood's assumptions must be retracted. As assumptions are retracted and/or asserted, datum dependencies propagate resulting changes in datum belief status. The complete set of nogoods encountered thus far serves to prune the assumption search space of other, as yet unvisited, points which subsume nogoods. Such search space points do not need to be investigated as they subsume a previously discovered inconsistency.

It is the Problem Solver's responsibility to ensure the semantic correctness of information which it passes to the RMS. The RMS treats these expressions purely syntactically. This separates the design of non-monotonic belief revision mechanisms from issues concerning knowledge representation and inferencing, which differ from one reasoning system to the next.

As well as supporting non-monotonic reasoning, and search space reduction, the RMS acts as a cache of inferencing results. Search space points typically share many common inferences. The RMS stores these results and makes them available to the Problem Solver, thereby eliminating the need for executing inferencing steps more than once.

One of the early RMS systems was that of Doyle in the late seventies [Doyle,79]. Doyle's RMS (which he called a TMS) is often referred to as a justification based RMS, by way of distinguishing it from de Kleer's more recent [de Kleer,86a], [de Kleer,86b], [de Kleer,86c], [de Kleer,88] assumption-based RMS (which he refers

to as an ATMS). The distinguishing feature is the way in which the relationship between datums and their entailing assumptions are maintained. In Doyle's TMS this relationship is implicitly represented by datum justification dependencies. Whenever an inconsistency is identified, datum dependencies are followed in reverse, in order to explicitly retrieve the responsible assumptions. In de Kleer's ATMS, as well as recording justifications, assumptions responsible directly or indirectly for a datum are explicitly and incrementally maintained as information stored with each datum. He refers to these conjunctive assumption sets as *environments*. Since each datum can be inferred under more than one environment, each datum stores a *label* which is a disjunctive set of environments. As each new datum justification is added, its label environments are propagated forward through the new justification.

By explicitly maintaining datum labels, the ATMS is able to efficiently respond to Problem Solver queries concerning more than one problem solving *extension*.[1]. Here we use *extension* to refer to the set of datums which follow from a maximal set of consistent assumptions referred to as an *interpretation*. All datums whose label contains an environment which is a subset of an interpretation, are held to be true in that extension. In contrast, Doyle's TMS supports queries regarding only one problem solving extension at a time. Each datum is labeled as being either *in* the current extension or *out* of the current extension. In principle, queries concerning other extensions can be derived from existing datum dependencies; to do so, however, is prohibitively computationally expensive. The ATMS maintains datum labels in anticipation of Problem Solver queries for more than just the current extension under active consideration. As a result, the Problem Solver can switch its attention between very different search space points at relatively little computational expense. This is particularly attractive in abductive problem solving,

[1]More generally an ATMS supports multiple contexts. A context refers to the set of datums which follow from a characterizing set of consistent assumptions. An extension refers to the set of datums which follow from a characterizing *maximal* set of consistent assumptions.

as we are often lead to compare several equally possible, mutually inconsistent explanations.

Historically, the control interface between the Problem Solver and its RMS has seen many variations. Doyle's TMS took responsibility for selecting which assumption to retract from a nogood set, and for selecting which alternative assumption to assert. In Doyle's *dependency directed backtracking* these choices are made arbitrarily, thereby denying the Problem Solver a role in guiding the direction of problem solving. In [de Kleer,86c] de Kleer points out a Problem Solver-RMS synchronization difficulty which he refers to as the *unouting* problem. This problem concerns how a Problem Solver keeps track of inferences which it has left on its agenda. Some inferences are common to several contexts. When a Problem Solver switches to a new context, it cannot always tell whether a shared inference has previously been executed or not. Often, in order to be sure, inferences are reexecuted unnecessarily.

De Kleer's Consumer Architecture [de Kleer,86c] presents a Problem Solver-ATMS interface which solves the unouting problem. In this architecture, the Problem Solver keeps track of its agenda items, called *consumers*, by associating them with the RMS datums that must be true in order for the consumer to be executed. By attaching consumers to datums, they are assigned labels much like regular datums. These labels can be used to determine whether a consumer is executable in a particular interpretation. Consumers disappear from the agenda immediately after execution. In this way, only consumers which are executable in a selected interpretation, and which have never been executed before, are scheduled for execution.

Unfortunately, de Kleer's original Consumer Architecture, is based on a breadth-first search of *all* possible interpretations. He argues that this approach is warranted for problems with many solutions all of which are required. In more recent work [de Kleer,86d], [Forbus,88a] de Kleer et. al. propose modifications to the original Consumer Architecture which place focusing control in the hands of

the Problem Solver, where it belongs. The Problem Solver defines a problem solving focus of attention by specifying one or more *focus environments*. As before, the Problem Solver associates consumers with RMS datums. In this case, however, only consumers which are executable in one of the focus environments are scheduled for execution. Consumer execution is interrupted whenever a focus environment becomes inconsistent. When this occurs the Problem Solver establishes an alternative set of focus environments. This focusing mechanism makes the architecture attractive for problems requiring a limited number of *highly ranked* solutions. Unfortunately, the view persists that the ATMS is specifically aimed at problems with many solutions, all of which are required.

The Consumer Architecture can be viewed as a kind of Blackboard Architecture. Blackboard Architectures are based on a paradigm of multiple specialists working together to solve a common problem. At any time, the state of the problem solving is represented on a global blackboard accessible to each specialist. Specialists view the emerging state of the blackboard solution and indicate when they are able to make a problem solving contribution. When authorized by a controlling monitor, specialists make their contributions by updating the blackboard with newly inferred information. When viewed as a Blackboard Architecture, the consumers are specialists, the RMS database is a blackboard, and the Problem Solver is the monitor. There are numerous examples of Blackboard Architectures in the literature [Nii,89]. Many of these systems solve problems best viewed as abduction. Protean determines possible 3-D protein molecule structures from evidence such as nuclear magnetic resonance (NMR) data obtained from the protein in solution. Other systems involve applications such as signal interpretation, and speech recognition.

Recent research has seen the incorporation of RMS techniques into Prolog implementations. Although resolution is based on deduction, its goal directed or backward chaining implementation in Prolog is syntactically indistinguishable from abduction. In both cases we

follow rules "backwards" from consequent to antecedent. Goebel [Goebel,90] distinguishes the two by the equivocal nature of the assumptions we reason back to in abduction, as compared to the unequivocal facts of Prolog. With abduction we reason back to assumptions which, *if they were* true, would explain our observations. In backward chaining we reason back to facts which *are* true. From the point of view of a Prolog Inference Engine, this difference is relatively minor. When faced with a non-deterministic clause choice during backward chaining, an Inference Engine does not know *at that time* whether a particular clause choice will successfully terminate in known facts or not. It can only proceed by making optimistic assumptions until inconsistencies force retractions. Thus the Inference Engine resorts to non-monotonic reasoning. This naturally leads to consideration of an RMS-based architecture.

Under Prolog's SLD-refutation procedure [Lloyd,84], clause choices result in resolvant transformations. High level resolvant goals are replaced by lower level subgoals; variables acquire bindings. From an RMS point of view, clause choices are assumptions. The assertion that a subgoal must be proven, or that a variable has a particular binding, are datums. By establishing justifications for each new datum, subgoals and their arguments are associated with the clause choices responsible for their state. As a result, unification failures can be analyzed and associated with nogoods as part of a dependency directed backtracking mechanism. This approach forms the basis for recent work in *Intelligent Backtracking* for Prolog [Bruynooghe,84], [Cox,84], [Drakos,88], [Havens,91], [You,89].

In practise intelligent backtracking schemes for Prolog are faced with a need to make practical tradeoffs. An RMS-based approach avoids needless rule execution at the expense of additional unification overhead as well as additional memory. Often it is better to execute some rules needlessly rather than attempt to eliminate all such needless executions. Many of the tradeoffs can be seen from the point of view of preserving the popular, efficient stack-based memory allocation scheme used in Prolog implementations

[Warren,77]. In these architectures a single problem solving context is incrementally extended and rolled back with the depth first pre-order traversal of the SLD search space. Likewise, intelligent backtracking schemes maintain a single context, and link the storage of datum dependency information to the stack-based problem solving state storage mechanism. Whenever an assumption is retracted, the problem solving context is rolled back to the point where that assumption was first made. In the process datums are discarded which could, in principle, be reused. Nogoods are also discarded prematurely.

Echidna [Havens,90] is a Constraint Logic Programming (CLP) language with dependency directed backtracking. Havens [Havens,91] refers to Echidna's unique integration of dependency backtracking with constraint propagation as *dataflow dependency backtracking*. The CLP paradigm [Jaffar,87a] provides natural support for the constraint relaxation techniques found in model based diagnosis [Davis,84], [de Kleer,87]. Havens argues against the use of a stack-based architecture for Echidna's dataflow dependency backtracking. As a result, when an assumption is retracted, rather than roll the problem solving state back to the state it had when the retracted assumption was first made, Echidna is able to keep datums that can be immediately reused.

Earlier we discussed the close relationship between the goal directed reasoning of Logic Programming languages and abductive reasoning. This similarity is reflected in the design of RMS-based systems for diagnosis [Hamscher,90a], [Struss,89], [de Kleer,86c], [de Kleer,87], [Hamscher,90b] and the incorporation of RMS techniques into intelligent backtracking techniques for Prolog. There are, however, important differences that affect the design of an efficient reasoning architecture. Abductive applications involve a comparison of multiple, ranked explanations. In purely deductive domains like theorem proving, only one solution is typically required. When more than one solution is required, they are not ranked in comparison to each other. Whereas intelligent backtracking schemes are based on

finding a single solution, diagnosis systems require multiple, highly-ranked diagnoses.

D'Ambrosio [D'Ambrosio,90a], [D'Ambrosio,90b], [D'Ambrosio,87] uses an ATMS as a basis for generalized probabilistic reasoning. D'Ambrosio shows that by associating assumptions with certainty values, label propagation computes symbolic certainty expressions for ATMS datums. His concern is not diagnosis, but the support of generalized probabilistic queries for arbitrary propositions, as conditioned by an incrementally maintained set of observations. D'Ambrosio's earlier work [D'Ambrosio,87] uses a certainty calculus based on the Dempster/Shafer theory of evidence. Subsequent work uses Bayesian probabilities and an ATMS representation of Bayesian networks. The ATMS is unfocused and maintains all possible contexts. This results in an exponential growth in the size of datum labels. To circumvent this problem, D'Ambrosio and Edwards [D'Ambrosio,91] decompose problems into a subproblem abstraction hierarchy, and use a partitioned ATMS which summarizes labels into simpler form, for datums exchanged between partitions.

De Kleer and Williams incorporate Bayesian analysis in SHERLOCK, an ATMS-based system for circuit diagnosis [de Kleer,89a]. Unlike D'Ambrosio's system, SHERLOCK sacrifices generality in favour of a particular application, namely, model based diagnosis. It assumes, for example, that circuit component states are statistically independent. SHERLOCK uses a best-first search to find diagnoses in order of decreasing probability. It uses focusing heuristics to limit the size of the set of competing diagnosis to only those which are highly likely. As a result, SHERLOCK avoids the exponential label growth problem, as well as the computational cost of deriving many relatively unlikely diagnoses.

2.4. What is needed

In distinguishing between model based diagnosis and medical diagnosis, we focus on the fact that whereas model based diagnosis is

concerned with physical systems like circuits or engines, medical diagnosis is concerned with the human body. In the former case we have a precise theory of how the system should behave; in the latter case we have only an empirical understanding of how the human body works. We argue that this distinction is not fundamental. Rather than two distinctly different kinds of problems, we have a continuum of abductive problems ranging from theoretically well understood systems, to systems for which we have only an empirical understanding.

Even for the case where we have a precise theory, completeness considerations can force us to consider the possibility that our theory is based on false assumptions. For example, in model based diagnosis we assume that the physical connectivity between components is as described by our model. Completeness forces us to consider the possibility of structural faults such as electrical shorts. Such problems are outside the theory represented by our model. Theories represent abstractions of the real world artifact under diagnosis. They are formed by distinguishing between those artifact properties that are important and those that are not. These distinctions are necessary to prevent us from having to model the artifact in infinite detail. Unfortunately, these same distinctions amount to contextual assumptions which can only be empirically justified. In the actual case of a faulty artifact, almost anything could be the cause, even that which we considered to be not worth modeling. It would seem that we can never actually have a purely model based diagnostic reasoner which offers diagnostic completeness. Even at this end of the spectrum we require a hybrid system. Whereas some parts of our model can be precisely defined, other parts can only be statistically defined.

Generalized abductive reasoning requires a knowledge representation capable of supporting both model-based and network-based diagnosis. Whereas deterministic behavior of healthy circuit components is well represented by rules or procedures, explicitly enumerated conditional probability distributions are

required in the case of medical diagnosis. A generalized representation must support theoretical representations of well understood behaviors as well as probabilistic representations of statistical behavior. In this thesis we propose a way of representing Bayesian networks in logic programs so as to accomplish this. In order to be useful, the integrated representation must preserve the desirable features of both logic and Bayesian network representations.

While we adopt a Prolog-like framework for our reasoning system, it is important to reconsider architectural issues in the light of abductive problem solving. We propose an architecture which utilizes a focused ATMS for Horn clause reasoning. We incorporate Bayesian techniques to ensure that more likely explanations are found first. The implications of these decisions changes the nature of many of the practical tradeoffs normally associated with intelligent backtracking. In particular, a best-first search of the assumption search space visits possible solutions in a very different order from the pre-order traversal of a stack-based Prolog implementation. This precludes the efficient stack-based architecture upon which many intelligent backtracking tradeoffs are based, but raises the possibility of adopting probabilistic heuristics to control datum caching.

Chapter 3
A Knowledge Representation for Abduction

In this chapter we present a knowledge representation based on Horn clause logic and Bayesian networks which is suitable for general problems in abductive reasoning. Horn clause programs are commonly referred to as *definite programs*. The notion of a definite program is extended to support complete representations of Bayesian networks. Under these extensions, programs used to represent Bayesian networks are referred to as *Bayesian programs*.

3.1. Syntax

In the following we provide syntactic definitions for definite programs, Bayesian networks and Bayesian programs.

3.1.1. Definite Programs

The reference textbook on the standard theory of logic programming is [Lloyd,84]. Our discussion of definite programs is based on a summary of this material. The interested reader is referred to the original text for a more complete presentation.

A first order theory consists of an alphabet, a first order language, a set of axioms and a set of inference rules. We are interested in first order theories where the set of axioms is a definite program, and resolution is the only inferencing rule.

An *alphabet* consists of seven classes of symbols:
(a) variables
(b) constants
(c) function symbols or *functor*
(d) predicate symbols

(e) connectives
(f) quantifiers
(g) punctuation symbols.

The connectives are $\neg,\wedge,\vee,\rightarrow$ and $\leftrightarrow$. The quantifiers are $\exists$ and $\forall$. The punctuation symbols are "(", ")" and "," .

The following definitions are needed for the definition of a first order language.

A *term* is defined inductively as:
(a) A variable is a term.
(b) A constant is a term
(c) If f is an n-ary function symbol and t1,...,tn are terms, then f(t1,...,tn) is a term.

A *(well-formed) formula* is defined inductively as:
(a) If p is an n-ary predicate symbol and t1,...,tn are terms, then p(t1,...tn) is a formula (called an *atomic formula* or, more simply, an *atom*).
(b) If F and G are formulas, then so are (~F),(F$\wedge$G),(F$\vee$G),(F$\rightarrow$G) and (F$\leftrightarrow$G).
(c) If F is a formula and x is a variable, then ($\forall$x F) and ($\exists$x F) are formulas.

The formula (F$\rightarrow$G) is often written as (G :- F).

The *first order language* given by an alphabet consists of the set of all formulas constructed from the symbols of the alphabet.

The following definitions are needed to support the definition of a definite program.

The *scope* of $\forall$x (resp. $\exists$x) in $\forall$x F (resp. $\exists$x F) is F. A *bound* occurrence of a variable in a formula is an occurrence immediately following a quantifier, or an occurrence, within the scope of a quantifier, of the same variable as immediately follows the quantifier. Any other occurrence of a variable is *free*.

A *closed formula* is a formula with no free occurrences of any variable.

If F is a formula, then $\forall(F)$ denotes the universal closure of F, which is the closed formula obtained by adding a universal quantifier for every variable having a free occurrence in F.

A *ground formula* is a formula with no variables.

A *literal* is an atom or the negation of an atom. A *positive literal* is an atom. A *negative literal* is the negation of an atom.

A *clause* is a closed formula of the form $\forall x_1 ... \forall x_s \ (L_1 \vee ... \vee L_m)$ where each L_i is a literal and $x_1,...,x_s$ are all the variables occurring in $L_1 \vee ... \vee L_m$.

It is easy to show that the clause $\forall x_1 ... \forall x_s \ (A_1 \vee ... \vee A_m \vee \neg B_1 \vee ... \vee \neg B_n)$, where $A_1,...,A_k,B_1,...,B_n$ are atoms and $x_1,...,x_s$ are all the variables appearing in these atoms, is equivalent to $\forall x_1 ... \forall x_s \ (A_1 \vee ... \vee A_m$:- $B_1 \wedge ... \wedge B_n)$ which, by convention, is written $A_1,...,A_m$:- $B_1,...,B_n$. Under this convention, all variables are assumed to be universally quantified, the commas in the antecedent $B_1,...B_n$ denote conjunction and the commas in the consequent $A_1,...A_k$ denote disjunction.

A *definite program clause* is a clause of the form A_1 :- $B_1,...,B_n$ which contains precisely one atom in its consequent. A is called the *head* and B1,...Bn is called the *body* of the program clause. The informal semantics of a clause of this form is that "for each assignment of each variable, if B1,...,Bn are all true, then A is true".

A *unit clause* is a definite program clause with an empty body. It has the form A :- .The informal semantics of a unit clause is that "A is unconditionally true for each assignment of each variable."

We often denote the unit clause, 'A :- ' as simply 'A.' .

A *definite program* is a finite set of definite program clauses.

A definite program for which:
(a) all unit clauses are ground
(b) no clause head includes variables which do not appear in the clause body is said to be a *ground definite program.*

In a definite program, the set of clauses with the same predicate symbol p in the head is called the *definition* of predicate p.

A *definite goal* is a clause which has an empty consequent. If $y_1,...,y_r$ are the variables of the goal :- $B_1,...,B_n$ then this clausal notation is shorthand for $\forall y_1...\forall y_r(\neg B_1 \vee ... \vee \neg B_n)$ or, equivalently, $\neg \exists y_1...\exists y_r(B_1 \wedge ... \wedge B_n)$. Each B_i (i=1,n) is called a *subgoal* of the goal.

We often denote the goal, :- $B_1,...,B_n$, as ?- $B_1,...,B_n$.

The *empty clause*, denoted $\circ$, is the clause with empty consequent and empty antecedent. This clause is to be understood as a contradiction.

A *Horn clause* is a clause which is either a definite program clause or a definite goal. Alternatively, a Horn clause is a clause which contains either one atom or no atoms in its consequent.

3.1.2. Bayesian Networks

The following sequence of definitions leads to the definition of a Bayesian network.

A *directed graph* G = (V,E) consists of:
(a) a finite, nonempty set of vertices V
(b) set of directed edges E

A *directed edge* is an ordered pair of vertices; v is called the *tail* and w the *head* of edge (v,w). We write tail(e) to refer to the tail of edge e. Similarly, head(e) refers to the head of edge e.

A *path* is a sequence of edges (v_1,v_2), (v_2,v_3) ... (v_{n-1},v_n). We say that the path is from v_1 to v_n and is of length n-1.

A path is *simple* if all edges and all vertices on the path except possibly the first and last vertices, are distinct.

A *cycle* is a simple path of length at least 1 which begins and ends at the same vertex.

A *directed acyclic graph* (DAG) is a directed graph with no cycles.

If (v,w) is an edge in a directed graph, we say that v is a *parent* of w and that w is a *child* of v.

The set of parents of a vertex x is designated as π_x.

If v is a vertex in a directed graph, we refer to the set consisting of v and all of its parents as the *family* of v.

If v and w are two vertices in a directed graph, and there is a path from v to w, then we say that v is an *ancestor* of w. We write ancestor(w) to refer to the set of all of w's ancestors.

If v and w are two vertices in a directed graph we say that v is a *weak ancestor* of w if v is an ancestor of w *or* v = w. We write prev(w) to refer to the set of all of w's weak ancestors.

If v and w are two vertices in a directed graph, then c is a *common weak ancestor* of v and w if:

(a) c is an element of prev(v)

(b) c is an element of prev(w)

If v and w are two vertices in a directed graph, then r is a *recent weak ancestor* of v and w if:

(a) r is a common weak ancestor of v and w.

(b) no direct successor of r is a common weak ancestor of v and w.

A vertex with no parents is referred to as a *terminal vertex*.

A *Bayesian network* is a DAG whose vertices are *nodes*. In the following we lead up to precise definitions for the terminal and non-terminal nodes of a Bayesian network.

An *unconditional probability expression* for *random variable* V is a sentence of the form:

P(V = c) = x
where x is a real number such that $0 \leq x \leq 1.0$,
c is a constant,
and "P", "(" , ")" , and "=" are punctuation symbols.

An *unconditional probability distribution* on variable V, denoted by P(V), is a set of unconditional probability expressions for V such that:

(a)All constants contained in the expressions of P(V) are distinct.

(b) $\sum_{i \in P(V)} x_i = 1.0$. Note that in order for this normalization condition to hold, P(V) must contain at least one element.

A *terminal node* is a pair (V,P(V)). In a Bayesian network a terminal node is designated by the name of its variable V.

For the terminal node (V,P(V)), the *domain* of V is defined as the set of those constants which appear in one of the node's associated probability expressions:

$$\text{domain}(V) = \{ c \mid \exists x \, (P(V=c) = x \in P(V)) \}$$

A *conditional probability expression* for *conditioned variable* V and *conditioning variables* $B_1,...,B_n$, is a sentence of the form:

$P(V=c \mid B_1=b_1,...,B_n=b_n) = x$
where x is a real number such that $0 \leq x \leq 1.0$,
c is a constant,
"P","(",")" and "=" are punctuation symbols,
and $b_1 \in \text{domain}(B_1)$,..., $b_n \in \text{domain}(B_n)$.

For the conditional probability expression $P(V=c|B_1=b_1,...,B_n=b_n) = x$, the sentence $B_1=b_1,...,B_n=b_n$ is referred to as the *condition* of the expression; the sentence V=c is called the *proposition* of the expression.

A *conditional probability distribution* on variable V is denoted by $P(V|B_1=b_1,...,B_n=b_n)$ where $B_1,...,B_n$ are conditioning variables. It is a set of conditional probability expressions such that:

(a)The same condition appears in each conditional probability expression.

(b)All constants contained in the propositions of $P(V|B_1=b_1,...,B_n=b_n)$ are distinct.

(c) $P(V|B_1=b_1,...,B_n=b_n)$ is the empty set or $\sum_{i \in P(V|B_1=b_1,...,B_n=b_n)} x_i = 1.0$.

The informal semantics of the empty set is that the specified condition does not occur in the associated semantic domain. If the condition does occur, then $P(V|B_1=b_1,...,B_n=b_n)$ contains at least one element since V must assume some value.

$P(V|B_1,...,B_n)$ denotes a set of conditional probability distributions:

$$P(V|B_1,...,B_n) = \{ P(V|B_1=b_1,...,B_n=b_n) \mid b_1 \in \text{domain}(B_1),..., b_n \in \text{domain}(B_n) \}$$

A *non-terminal node* is a pair $(V,P(V|B_1,...,B_n))$ where $B_1,...,B_n$ are the parents of node V. In a Bayesian network a non-terminal node is designated by the name of its variable V.

For the non-terminal node $(V,P(V|B_1,...,B_n))$, the *domain* of V is defined as the set of those constants which appear in the propositional part of one of the node's associated conditional probability expressions :

$$\text{domain}(V) = \{ c \mid \exists b_1,...,\exists b_n,\exists x\ P(V=c|B_1=b_1,...B_n=b_n) = x \in P(V|B_1=b_1,...,B_n=b_n) \in P(V|B_1,...,B_n) \}$$

This completes the definition of a Bayesian network. The following terms are needed in the next section where a Bayesian program is defined.

The set of variable assignments, such that each variable of a Bayesian network B, is assigned one of its domain values, is called an *extension* of B

From the definition of a Bayesian network, if N is a node in a Bayesian network B, then N together with its ancestors also constitutes a Bayesian network. We refer to an extension of this network as a *tail extension* of node N.

If N1 and N2 are nodes of a Bayesian network B, and N1 is an ancestor of N2, then each tail extension E2, of node N2, includes a tail extension E1 of node N1. We say that E1 is a *sub-tail extension* of E2.

3.1.3. Bayesian Programs

A Bayesian program is a definite program which is structured after a Bayesian network, and where some program clauses are annotated with probabilities. In the following we define how a Bayesian program is constructed from a Bayesian network.

An *annotated clause* is of the form { definite program clause, x } where x is a real number and $0 < x < 1.0$.

A *Bayesian clause* is either a definite program clause or an annotated definite program clause.

Following convention we adopt "." as the *list* functor. A list may be defined inductively as:

(a) [] which is a symbol for the nil list
(b) .(X, list) where X stands for any term

Also following convention we adopt the more convenient Prolog syntax for representing lists. The difference between the two representations is illustrated by the following table:

Formal Representation	**Prolog Representation**
.(a,[])	[a]
.(a,.(b,[]))	[a,b]
.(a,.(b,.(c,[])))	[a,b,c]
.(a,_X)	[a|_X]
.(a,.(b,_X))	[a,b|_X]

We adopt the Prolog representation purely a matter of convenience. The Prolog syntax can always be replaced with its equivalent first order language syntax.

Let B be a Bayesian network, and let N be the set of all variable names for B, then a *node term* is defined as:

> /(Nodename, Nodevalue) where Nodename represents an element of N and Nodevalue, an element of domain(Nodename).

For syntactic convenience we represent the "/" functor as an infix operator. The term Nodename/Nodevalue is equivalent to the term /(Nodename,Nodevalue). By convention we use capital letters for Nodenames and small letters for Nodevalues. Informally, the node term A/a represents the assignment A=a for random variable A. We use the "_" character to distinguish between constants and variables. The node term _A/_a consists of the variables _A and _a. The node term A/a consists of the constants A and a.

An *extension term* is defined as:

> ←(X,Es) where X stands for a node term, and Es is a list of extension terms. Es may be the nil list.

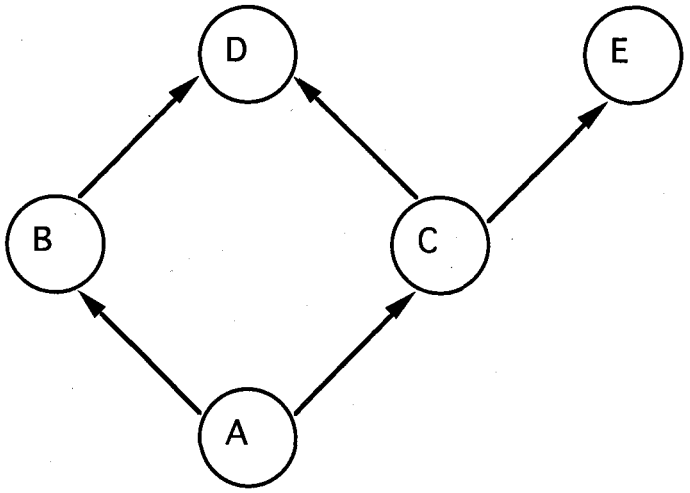


Figure 3.1. A Bayesian Network.

Informally, extension terms are used to represent the tail extensions of a Bayesian network node. The tail extension of a node consists of the union of the tail extensions of its parents, Es, together with its own assigned node value, X. Again, as a matter of syntactic convenience, we represent the ← functor as an infix operator. The term X ← [Y,Z] is syntactically equivalent to the term ←(X,[Y,Z]). A tail extension of node D of the Bayesian network shown in figure 3.1 is represented as D/d ← [B/b ← [A/a ← []], C/c ← [A/a ← []]] .

In a Bayesian network a terminal node V is represented as (V,P(V)). We convert this to a set of annotated definite program clauses:

(a) associate a unary predicate with the node. For the purpose of discussion let this predicate be referred to as "node_V". Informally, this predicate enumerates the possible tail extensions of terminal node V.

(b) If P(V) contains the single probability expression P(V=c) = 1.0 create the unannotated clause: node_V(V/c ← []).

(c) for each probability expression P(V=c) = x ∈ P(V), where x is neither zero nor one, create the annotated clause: {node_V(V/c ← []),x}.

In a Bayesian network a non-terminal node V is represented as $(V,P(V|B_1,...,B_n))$. We convert this to a set of annotated definite program clauses:

(a) let F be the family of V. Associate a k-ary predicate with V where k is the cardinality of F. For the purpose of discussion let this predicate be referred to as "family_V".
(b) for each probability expression $P(V=c|B_1=b_1,...,B_n=b_n)=1.0$ associated with $P(V|B_1,...,B_n)$, create the clause: family_V(V/c,B1/b1,...,Bn/bn).
(c) for each probability expression $P(V=c|B_1=b_1,...,B_n=b_n) = x$ associated with $P(V|B_1,...,B_n)$ where x is neither zero nor one, create the annotated clause { family_V(V/c,B1/b1,...,Bn/bn), x }.
(d) associate a unary predicate with the node. For the purpose of discussion let this predicate be referred to as "node_V". Informally, this predicate enumerates the tail extensions of non-terminal node V.
(e) let P be the set $\{B_1,...B_n\}$, the parents of V. Let R be the set $\{r \mid \exists B_i\ \exists B_j\ B_i \in P \wedge B_j \in P \wedge r \in$ recent weak ancestor of B_i and $B_j\}$
(f) if R is the empty set create the *node clause*: node_V(_V←[_B1←_X1,...,_Bn←_Xn]):- node_B1(_B1←_X1) ,...,node_Bn(_Bn←_Xn),family_V(_V,_B1,...,_Bn). Informally, node_V builds tail extensions of V from the tail extensions of its parent nodes provided that the node terms of V and its parents form a consistent family of values. In this case the variables _X1,...,_Xn simply abstract away unnecessary detail.
(g) if R is not the empty set, we elaborate variables X1,...,Xn so as to identify sub-tail extensions which parent tail extensions are required to have in common. We do this by

replacing the variables X1,...,Xn with extension representations for which each tail extension of a recent weak ancestor $\in$ R is named with a distinct variable. In this way shared sub-tail extensions are represented as variables which are shared between subgoals in the body of the node clause. A node clause for which shared node subgoal variables correctly represent shared sub-tail extensions is said to be *network consistent.*

Ignoring variables for a moment we see that the A :- B1,...,Bn,family_A structure of a node clause represents the local network relationships of node A with its immediate parents. This is similar to the syntactic modularity of MYCIN rules described earlier. In contrast, the shared variables of node subgoals represent those global network relationships which are required in order to maintain the global coherency which was lacking in MYCIN's representation. By representing only *recent* weak ancestors as shared variables, we keep global information to a minimum in each node clause.

We refer to node_A predicates, where A is a node name, as *node predicates.* Similarly family_A is referred to as a *family predicate.*

If we apply the above transformations to the set of nodes of a Bayesian network B, then the set of annotated and unannotated clauses which result constitute a *Bayesian program* based on B.

A *Bayesian clause* is an element of a Bayesian program.

Let BP be a Bayesian Program. Let BP' be the definite program formed by stripping each annotated clause in BP of its annotation. We refer to BP' as the *base program* for BP.

The following is an example of a Bayesian program based on the Bayesian network shown in figure 3.1.

```
node_D( _D←[_B←[_A],_C←[_A]]) :-   node_B(_B←[_A]),
                                   node_C(_C←[_A]),
                                   family_D(_D,_B,_C).
```

```
{family_D(D/d1,B/b1,C/c1)., .7}
{family_D(D/d2,B/b1,C/c1)., .3}
family_D(D/d1,B/b1,C/c2).
{family_D(D/d1,B/b2,C/c1)., .45}
{family_D(D/d2,B/b2,C/c1)., .55}
{family_D(D/d1,B/b2,C/c2)., .9}
{family_D(D/d2,B/b2,C/c2)., .1}

node_B(_B←[_A←_X1]) :-  node_A(_A←_X1),
                        family_B(_B,_A).

{family_B(B/b1,A/a1)., .65}
{family_B(B/b2,A/a1)., .35}
{family_B(B/b1,A/a2)., .8 }
{family_B(B/b2,A/a2)., .2}

node_C(_C←[_A←_X1]) :-  node_A(_A←_X1),
                        family_C(_C,_A).

{family_C(C/c1,A/a1)., .85}
{family_C(C/c2,A/a1)., .15}
{family_C(C/c1,A/a2)., .6 }
{family_C(C/c2,A/a2)., .4}

{node_A(A/a1←[]).,.8}
{node_A(A/a2←[]).,.2}

node_E(_E←[_C←_X1]) :-  node_C(_C←_X1),
                        family_E(_E,_C).

{family_E(E/e1,C/c1)., .3}
{family_E(E/e2,C/c1)., .4}
{family_E(E/e3,C/c1)., .3}
{family_E(E/e1,C/c2)., .67}
{family_E(E/e2,C/c2)., .33}
```

Thus far, Bayesian program clauses correspond 1:1 with Bayesian network probability sentences. We now describe equivalent Bayesian

program representations which can offer more succinct, generative representations.

Consider the node A of a Bayesian network B with domain(A) = $\{a_1,a_2,...,a_n\}$. Let BP be a Bayesian Program constructed from B as described above. We have a set S of n clauses of the form { node_A(A/$a_i \leftarrow$ []) , x_i } where each clause i corresponds to the distinct A value a_i.

Since several clauses may have the same x value, we have m distinct x values where m ≤ n. Consider the case where:

(1) n is large

(2) m is small

In this case we may wish to consider the following equivalent representation. Let S_x be an S subset of k clauses which share the same probability x. We can replace these k clauses with:

(a)the annotated clause { node_A(A/_a ← []) :- node_ax(_a), x }

(b) the clauses of a ground definite program Nax defined such that:

(1) Nax includes a definition of node_ax

(2) node_ax(a) ∈ M_H, the least Herbrand model for Nax, iff

{ node_A(A/a← []), x } ∈ S_x

M_H represents the declarative meaning of definite program Nax, as will be discussed in a subsequent section.

We may replace any number of unit node clauses in this way. If node_A is defined by a complete set of ground unit clauses we say that these clauses form an *explicit definition* for node_A. Alternatively, if we do one or more of the substitutions described above, we refer to the final set of clauses as a *generative definition* for node_A.

We can make similar replacements for family clauses. We have a set S of n = | domain(A) | x | domain(B_1) | x ...x | domain(B_n) |

clauses of the form { family_A(A/a_i, B1/$b1_i$, ..., Bn/bn_i) , x_i } where the parents of node A are B_1,...,B_n and each clause i corresponds to a distinct set of A,B_1,...,B_n values.

We have m distinct x values where $m \leq n$. In the case where n is large and m is small we may wish to consider a generative representation.

Let S_x be an S subset of k clauses which share the same probability x. We can replace the k clauses with the annotated clause:

(a) {family_A(A/_a,B1/_b1,...,Bn/_bn) :-
family_ax(_a,_b1,...,_bn), x}
(b) the clauses of a ground definite program Fax defined such that:
(1) Fax includes a definition of family_ax
(2) family_ax(a,b1,...,bn) $\in M_H$, the least Herbrand model for Fax, iff {family_A(A/a,B1/b1,...,Bn/bn), x } $\in S_x$

Unannotated family clauses F are treated as equivalent to {F,1.0}. We may follow the same replacement procedure for sets of unannotated family clauses.

As with terminal nodes we refer to family_A definitions as being either explicit or generative.

3.2. Declarative Semantics

In the following we define the declarative semantics of definite programs, Bayesian networks and Bayesian programs.

3.2.1. Definite Programs

A *pre-interpretation* of a first order language L consists of the following:

(a) A non-empty set D, called the *domain* of the pre-interpretation.
(b) For each constant in L, the assignment of an element in D.
(c) For each n-ary function symbol in L, the assignment of a mapping from D^n to D.

Informally, a pre-interpretation of a language defines a semantic domain, and a way of referring to elements of that domain.

An *interpretation* I of a first order language L consists of a pre-interpretation J with domain D together with the following: For each n-ary predicate symbol in L, the assignment of a mapping from D^n into {true,false} (or, equivalently, a relation on D^n). We say that I is based on J.

Informally, an interpretation of a language defines a set of atomic assertions that we can make about the elements of a semantic domain. There are many possible interpretations for any particular pre-interpretation.

Let J be a pre-interpretation of a first order language L. A *variable assignment* (wrt J) is an assignment to each variable in L of an element in the domain of J.

Let J be a pre-interpretation with domain D of a first order language L and let V be a variable assignment. The *term assignment* (wrt J and V) of the terms in L is defined as follows:
(a) Each variable is given its assignment according to V.
(b) Each constant is given its assignment according to J.
(c) If $t'_1,\ldots,t'_n$ are the term assignments of $t_1,\ldots,t_n$ and f' is the assignment of the n-ary function symbol f, then $f'(t'_1,\ldots,t'_n) \in D$ is the term assignment of $f(t_1,\ldots,t_n)$.

Let I be an interpretation with domain D of a first order language L and let V be a variable assignment. Then a formula in L can be given a *truth value*, true or false, (wrt I and V) as follows:

(a) If the formula is an atom p(t1,...,tn), then the truth value is obtained by calculating the value of $p'(t'_1,...,t'_n)$ where p' is the mapping assigned to p by I and $t'_1,...,t'_n$ are the term assignments of $t_1,...,t_n$ wrt I and V.

(b) If the formula has the form $\neg F$, $F \wedge G$, $F \vee G$, $F \rightarrow G$ or $F \leftrightarrow G$, then the truth value of the formula is given by the following table:

F	G	$\neg$ F	F$\wedge$G	F$\vee$G	F$\rightarrow$G	F$\leftrightarrow$G
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false
false	false	true	false	false	true	true

(c) If the formula has the form $\exists x$ F, then the truth value of the formula is true if there exists $d \in D$ such that F has truth value wrt I and V(x/d), where V(x/d) is V except that x is assigned d; otherwise, its truth value is false.

(d) If the formula has the form $\forall x$ F, then the truth value of the formula is true if, for all $d \in D$, we have that F has truth value true wrt I and V(x/d); otherwise, its truth value is false.

It is clear from the above that the truth value of a closed formula does not depend on the variable assignment V.

If the truth value of a closed formula with respect to an interpretation is true (resp., false), we say the formula is true(resp., false) with respect to the interpretation.

Let I be an interpretation of a first order language L and let F be a closed formula of L. We say that I is a *model* for F if F is true wrt I.

Let S be a set of closed formulas of a first order language L and let I be an interpretation of L. We say I is a *model* for S if I is a model for each formula of S.

Let P be a definite program defined on a language L and let I be an interpretation of L. As a special case of the above statement, I is a *model* for P if I is a model for each clause of P.

Let S be a set of closed formulas of a first order language L. We say that S is *unsatisfiable* if no interpretation of L is a model for S.

Typically, we define a definite program with a particular semantic domain and interpretation in mind. While the above procedure enables us to determine whether or not an intended interpretation is a model for a particular definite program, we are also interested in whether or not the intended interpretation follows from the definite program and the intended semantic domain. Does our definite program when interpreted with respect to a particular pre-interpretation necessarily lead to the intended interpretation? In order to answer such questions we define the notion of logical consequence.

Let P be a definite program of first order language L and F be a closed formula of L. Let F have the form $\exists y_1,...,\exists y_r\ (B_1,...,B_n)$ where $y_1,...,y_r$ represent the complete set of variables appearing in atoms $B_1,...,B_n$. We say that F is a *logical consequence* of P if, for every interpretation I of L, I is a model for P implies that I is a model for F.

It is not difficult to show that F is a logical consequence of P iff $P \cup \{\neg F\}$ is unsatisfiable. Since the formula $\neg F$ is equivalent to the definite goal $G = :- B_1,B_2,...,B_n$, F is a logical consequence of P is equivalent to showing that the set of Horn clauses $P \cup \{G\}$ is unsatisfiable.

To show that $P \cup \{G\}$ is unsatisfiable, we must show that no interpretation of L is a model for $P \cup \{G\}$. At first glance this seems difficult as there are any number of semantic domains upon which we can base an interpretation. Fortunately, as it turns out, we can restrict our attention to a particular domain called the Herbrand universe.

Let L be a first order language. The *Herbrand universe* U_L for L is the set of all ground terms, which can be formed out of the constants and function symbols appearing in L.

Informally, the Herbrand universe for L is the set of all names for 'things' supported L.

Let L be a first order language. The *Herbrand base* B_L for L is the set of all ground atoms which can be formed by using predicate symbols from L with ground terms from the Herbrand universe as arguments.

Let L be a first order language. The *Herbrand pre-interpretation* for L is the pre-interpretation given by the following:

(a) The domain of the pre-interpretation is the Herbrand universe U_L.

(b) Constants in L are assigned themselves in U_L.

(c) If f is an n-ary function symbol in L, then the mapping from $(U_L)^n$ into U_L defined by $(t_1,...,t_n) \rightarrow f(t_1,...,t_n)$ is assigned to f.

In effect, the Herbrand pre-interpretation lets each name for a 'thing' supported by the language L, stand for itself in the set of all such names.

A *Herbrand interpretation* for L is any interpretation based on the Herbrand pre-interpretation for L.

Following convention we identify each Herbrand interpretation with a particular subset of the Herbrand base. For each Herbrand interpretation, its associated Herbrand base subset consists of only those ground atoms which are made true by the interpretation. Conversely, for any subset of the Herbrand base, its associated Herbrand interpretation is defined by specifying that the mapping defined by predicate symbols maps arguments to true precisely when the atom made up of the predicate symbol and these same arguments is in the Herbrand base subset. This identification enables

us to talk of Herbrand interpretations as equivalent to Herbrand base subsets.

Let L be a first order language and S a set of closed formulas of L. An *Herbrand model* for S is an Herbrand interpretation for L which is a model for S.

We can now associate each interpretation I of a language L with a corresponding Herbrand interpretation H for L.

$$H = \{ p(t1,...,tn) \mid p(t1,...,tn) \in B_L \wedge p(t1,...,tn) \text{ is true wrt I} \}$$

It is readily shown that if I is a model for S, a set of clauses defined on L, then H is also a model for S.

The implication is that if S has any model then it must have a Herbrand model. Put differently, S is unsatisfiable iff S has no Herbrand models.

Let P be a definite program of L and F a closed formula of L of the form $\exists y_1,...,\exists y_r\ (B_1,...,B_n)$. We said that F is a logical consequence of P iff $P \cup \{G\}$ is unsatisfiable where $G = :- B_1,B_2,...,B_n$. From the above results, since $P \cup \{G\}$ is a set of clauses (Horn clauses), it is only necessary to consider Herbrand Interpretations in order to establish the unsatisfiability of $P \cup \{G\}$.

The declarative semantics of a definite program can now be established. It is straightforward to show that for P, a definite program of L, and a non-empty set of Herbrand models for P, $\{M_i\}$, then $\bigcap_i M_i$ is also a model for P. Since every definite program P has B_P as an Herbrand model, the set of all Herbrand models for P is non-empty. Thus the intersection of all Herbrand models for P is again a model, called the *least Herbrand model*, for P. We denote this model by M_P. M_P is the declarative meaning of a definite program.

It can be shown that for a definite program P, $M_P = \{A \mid A \in B_P \wedge A \text{ is a logical consequence of } P\}$. The ground atoms contained in M_P are precisely those that are logical consequences of P.

It is also readily shown that the least Herbrand model M_P of a definite program P grows monotonically as clauses are added to P. The addition of clauses to P can only add new atoms to M_P; it cannot remove previously included atoms. We say that Horn clause logic is *monotonic*.

We can also give a fixpoint characterization of M_P. First we define some notation. Let f be a mapping defined on domain d, f: d $\rightarrow$ d. We define $f^n(x)$ inductively, where n is a positive non-zero integer and $x \in d$:

(a) $f^1(x) = f(x)$
(b) $f^n(x) = f(f^{n-1}(x))$

We use $f^\infty(x)$ to refer to $f^n(x)$ where n is infinity (the least upper bound of the set of non-negative integers).

We say that x is a *fixpoint* of f if f(x) = x.

A fixpoint characterization of M_P requires the notion of a substitution.

A *substitution* θ is a finite set of the form $\{v_1/t_1,...,v_n/t_n\}$, where each v_i is a variable, each t_i is a term distinct from v_i and the variables $v_1,...,v_n$ are distinct. Each element v_i/t_i is called a *binding* for v_i. θ is called a *ground substitution* if the t_i are all ground terms.

An *expression* is either a term, a literal or a conjunction or disjunction of literals.

Let $\theta = \{v_1/t_1,...,v_n/t_n\}$ be a substitution and E be an expression. Then $E\theta$, the *instance* of E by θ, is the expression obtained from E by

simultaneously replacing each occurrence of the variable v_i in E by the term t_i (i=1,...,n). If Eθ is ground, then Eθ is called a *ground instance* of E.

The substitution given by the empty set is called the *identity substitution.*

Let P be a definite program of the first order language L. Let 2^{B_P} represent the set of all Herbrand interpretations of L including the empty set interpretation. The mapping T_P: $2^{B_P} \rightarrow 2^{B_P}$ is defined as follows. Let I be an Herbrand interpretation. Then:

> $T_P(I) = \{A \mid A \in B_P \wedge A :- A_1,...,A_n$ is a ground instance of a clause in P and $I \supseteq \{A_1,...,A_n\}\ \}$.

It can be shown that M_P is a fixpoint[1] of T_P, and that it is equal to $T_P^{\infty}(\varnothing)$.

We refer to the function T_P for P as the *fixpoint function* for P.

If M_P is a finite set then there will exist a smallest finite positive integer k such that $T_P^{n>k}(\varnothing) = T_P^{k}(\varnothing) = M_P$.

It remains to relate M_P to the intended interpretation of a definite program P defined on a first order language L. Let J be the intended pre-interpretation defined for the intended semantic domain. We can associate M_P, and J with an interpretation M based on J.

> Let $M = \{ p(t'_1,...,t'_n) \mid p(t_1,...,t_n) \in M_P \wedge t'_1,...,t'_n$ are the term assignments under J of the ground terms $t_1,...,t_n \}$

In the above we refer to $p(t'_1,...,t'_n)$ as a *J-instance* of predicate p.

---

[1] It is actually the *least fixpoint* of T_P where 2^{B_P} is recognized as a complete lattice under the partial order of set inclusion. A least fixpoint is ≤ any other fixpoint.

Here we identify the interpretation M with a set of J-instances in the same way that we identified Herbrand interpretations with Herbrand base subsets. The mapping associated with a predicate symbol maps arguments to true precisely when the associated J-instance $\in$ M.

Clearly, M is also a model for P. In comparing M with an intended interpretation I we compare actual and intended meanings of P. The actual meaning M is sound if it is a subset of I. The actual meaning M is complete if it is a superset of I. The actual meaning is both sound and complete with respect to an intended meaning if M = I.

Finally we provide a declarative definition of a correct answer.

Let P be a definite program and G a definite goal. An *answer* for $P \cup \{G\}$ is a substitution for the variables of G.

Let P be a definite program, G a definite goal ?- $A_1,...,A_k$ and θ an answer for $P \cup \{G\}$. We say that θ is a *correct answer* for $P \cup \{G\}$ if $\forall((A_1 \wedge ... \wedge A_k)\theta)$ is a logical consequence of P.

Let P be a definite program and G a definite goal ?- $A_1,...,A_k$. Suppose θ is an answer for $P \cup \{G\}$ such that $(A_1 \wedge ... \wedge A_k)\theta$ is ground. Then θ is correct iff $(A_1 \wedge ... \wedge A_k)\theta$ is true wrt the least Herbrand model of P.

3.2.2. Bayesian Networks

In this section we define what a probability model is, and what it means. We then show that a Bayesian network, along with implicit independence assumptions, is equivalent to a probability model.

For our purposes, we define an *experiment* as a set of *random variables* and a set of distinguishable elements called *outcomes*.

A *random variable* of an experiment is a total function which maps from the set of experiment outcomes to a particular set of

values called the *range* of the random variable. Since each experiment outcome is mapped to a different range value, the random variable partitions the set of experiment outcomes into subsets of outcomes which have the same associated random variable assignment.

An experiment outcome is distinguishable from other outcomes purely by the complete set of its associated random variable assignments, one for each random variable of the experiment. For an experiment with three random variables A,B,C, an outcome with random variable assignments a1,b1,c1 is represented as { (A,a1),(B,b1),(C,c1) }.

An *event* of an experiment is a set of experiment outcomes. We write $A^{-1}(a1)$ to refer to the set of outcomes: {o| A(o) = a1 }. In this way we can represent arbitrary sets of events as set theoretic expressions. For example, we write the expression $(A^{-1}(a1) \cup B^{-1}(b1)) \cap C^{-1}(c1)$ to represent the event $\{o \mid (A(o)=a1 \vee B(o)=b1) \wedge C(o) = c1 \}$. We note that any set theoretic expression can be written in 'union of intersection' form.

An *elemental event* is an event consisting of one outcome.

A *probability model* is an experiment and a *probability function* Prob. Prob is a function which maps experiment events into a non-negative real number such that the following three *axioms of probability* hold:

I. Prob(event) ≥ 0
II. Prob(S) = 1 where S, the *certain event*, is the set of all experiment outcomes.
III. if $e1 \cap e2 = 0$ then $Prob(e1 \cup e1) = Prob(e1) + Prob(e2)$.

From these three axioms other properties of probabilities can be derived:

- $Prob(\varnothing) = 0$ where the empty set is referred to as the *impossible event.*

- Prob(e) = 1 - Prob(¬e)
- Prob(e1 ∪ e2) = Prob(e1) + Prob(e2) - Prob(e1 ∩ e1)

The *conditional probability function* is defined as:

$$\text{Prob}(e \mid m) = \frac{\text{Prob}(e \cap m)}{\text{Prob}(m)} \text{ for Prob}(m) > 0$$

We say that Prob(e|m) is the probability of event e given the condition m. If (E,Prob) is a probability model, then so is (E,Prob') where Prob' is the function s.t. Prob'(e) = Prob(e|m) for given condition m. In other words Prob(e|m) maps event e into a non-negative real number such that the three probability axioms hold. As a result, conditional probabilities with the same condition m can be combined in the same way that unconditional probabilities can. For example, we can write:

- Prob(∅|m) = o.
- Prob(e|m) = 1 - Prob(¬e|m)
- Prob(e1 ∪ e2|m) = Prob(e1|m) + Prob(e2|m) - Prob(e1 ∩ e1|m)

If m is the certain event then these expressions reduce to their earlier form, since Prob(e|certain event) = Prob(e).

We can subsequently form Prob'' from Prob' and condition m2. It can be shown that:

Prob''(e) = Prob'(e|m2) = Prob(e|m,m2).

The celebrated Bayes theorem follows from the definition of the conditional probability function:

$$\text{Prob}(e \mid m)^{1} = \frac{\text{Prob}(m \mid e)\text{Prob}(e)}{\text{Prob}(m)}$$

Event e1 is said to be *independent* of e2 if:

---

[1]Throughout the thesis, whenever we write Prob(e|m) we shall assume that Prob(m) ≠ 0 even if this is not explicitly stated.

$$Prob(e1 \cap e2) = Prob(e1)Prob(e2).$$

which is equivalent to:

$$Prob(e1 \mid e2) = Prob(e1)$$

which is equivalent to:

$$Prob(e2 \mid e1) = Prob(e2).$$

We can add a condition m to the above expressions to form definitions for conditional independence. Event e1 is said to be *conditionally independent* of e2 if:

$$Prob(e1 \cap e2 \mid m) = Prob(e1 \mid m)Prob(e2 \mid m)$$

which is equivalent to:

$$Prob(e1 \mid e2 \cap m) = Prob(e1 \mid m)$$

which is equivalent to:

$$Prob(e2 \mid e2 \cap m) = Prob(e2 \mid m)$$

Since any event can be expressed as a union of elemental events, and since elemental events are mutually exclusive, then by probability axiom III, the probability of an arbitrary event can be represented as the sum of elemental event probabilities. In other words, an experiment E together with probability values for each elemental event of E, constitutes a complete probability model. The probabilities of arbitrary events can be derived from these numbers. Likewise, since conditional probabilities are defined in terms of unconditional probabilities, arbitrary conditional probabilities can also be derived from elemental event probabilities.

If we have a probability model P for experiment E with random variables A,B,C... then we can define an associated *joint probability function* P as:

$P(X,Y,Z,...) = Prob(A^{-1}(X) \cap B^{-1}(Y) \cap C^{-1}(Z) ...)$
where the domains of X,Y,Z,... are the ranges of A,B,C,... respectively.

It is readily apparent that:

(1) $P(X,Y,Z,...) \geq 0$
(2) $\sum_{X,Y,Z,...} P(X,Y,Z,...) = 1$
(3) distinct (X,Y,Z,...) points correspond to distinct, mutually exclusive elemental events

The joint probability distribution P(X,Y,Z...) together with experiment E is a complete representation of a probability model.

Since the variable names X,Y,Z,... are arbitrary, by convention we name each variable after the random variable which assigns it a value. For example, we write P(X,Y,Z,...) as P(A,B,C,...). In order to prevent ambiguous interpretations of the symbols A,B,C... we do not use this convention for expressions involving the Prob function.

We can eliminate variables in P(X,Y,Z,...) by summing over them. For example:

$$P(X,Y) = \sum_{Z,...} P(X,Y,Z,...)$$
$$= Prob(A^{-1}(X) \cap B^{-1}(Y))$$

P(X,Y) has the following properties:

(1) $P(X,Y) \geq 0$
(2) $\sum_{X,Y} P(X,Y) = 1$
(3) distinct (X,Y) points correspond to distinct, mutually exclusive events

We can also define *conditional joint probability* functions as ratios of joint probability functions. For example:

$$P(X,Y \mid Z) = \frac{P(X,Y,Z)}{P(Z)}$$
$$= \frac{Prob(A^{-1}(X) \cap B^{-1}(Y) \cap C^{-1}(Z))}{Prob(C^{-1}(Z))}$$
$$= Prob(A^{-1}(X) \cap B^{-1}(Y) \mid C^{-1}(Z))$$

The following *chain rule* is derived by repeated application of the definition of conditional probability. This rule states that for a joint probability distribution on variables $X_1, X_2, \ldots X_n$, we have the identity:

$$P(X_1, X_2, \ldots X_n) = \prod_{i=1}^{n} P(X_i \mid Prev_i)$$

where $Prev_i$ is the set of variables X_j where $j < i$.

We now turn to semantic issues. Consider the probability model E with random variables A,B,C... . It is defined by a set of normalized *probability sentences* for elemental events:

$$Prob(A^{-1}(a_i) \cap B^{-1}(b_i) \cap C^{-1}(c_i) \cap \ldots) = r_i$$

where i denotes a particular sentence.

For convenience we use the simpler equivalent representation:

$$prob([A/a_i, B/b_i, C/c_i, \ldots]) = r_i$$

Each of these probability sentences asserts that:

(1) $[A/a_i, B/b_i, C/c_i, \ldots]$ denotes a particular experiment outcome.
(2) Its associated non-zero, elemental probability equals r_i

We can therefore treat 'prob' as a generalized predicate which maps to a probability value. A non-zero probability corresponds to TRUE. We can separate the semantic issues accordingly. Which experiment outcomes are possible? What do the probability values mean?

Considering the first question, we form the definite program DP made up of the ground atomic assertions:

prob([A/a_i,B/b_i,C/c_i,...]).

For this definite program 'A/a_i' is an infix representation of /(A,a_i). Similarly, we can consider [A/a_i,B/b_i,C/c_i,...] to be equivalent to the functor [] applied to terms (A/a_i,B/b_i,C/c_i,...) .

For this program the Herbrand universe is:

(1) 'A','B' etc.
(2) 'a_i', 'b_i' etc.
(3) 'A/a_i', 'A/b_i' etc.
(4) '[B/a_i,A/b_i,C/c_i,...]' etc,

The Herbrand pre-interpretation is just:

(1) 'A' → 'A', 'B' → 'B' etc.
(2) 'a_i'→'a_i' , 'b_i'→'b_i' etc.
(3) '/' maps 'A','a_i' to 'A/a_i' etc.
(4) '[]' maps A/a_i,B/b_i,C/c_i,... to [A/a_i,B/b_i,C/c_i,...] etc.

The least Herbrand model consists of precisely those ground atomic assertions making up DP. As required, each DP clause asserts the possible occurrence of a particular experiment outcome.

In effect DP defines an *Herbrand experiment* where the outcomes are syntactic terms such as '[A/a_i,B/b_i,C/c_i,...]'. For the set of outcomes, the random variable symbols such as 'A', do indeed act as random variables. For example, 'A' maps outcome sentence '[A/a_i,B/b_i,C/c_i,...]'to 'a_i'

We can relate DP to 'real' experiments. For example, in a circuit diagnosis application, the 'real' experiment may be to select a circuit from the set of all circuits of a particular type. An outcome of this experiment is a particular circuit. Each circuit has measurable properties such as its output signal level.

We form the semantic domain made up of:

(1) properties such as output signal level.
(2) property values such as 5.0 volts.
(3) property assignments such as output signal level = 5.0 volts.
(4) combinations of property assignments.

The semantic pre-interpretation is :

(1) mappings such as 'A' → output signal level.
(2) mappings such as 'a_i'→ 5.0 volts.
(3) '/' maps properties and property values to property assignments.
(4) '[]' maps property assignments into property assignment combinations.

The actual semantic interpretation is formed by interpreting the arguments of the atoms making up the least Herbrand model according to their semantic pre-interpretation. The program DP defines the circuits which constitute outcomes to this 'real' experiment by enumerating the measurable property values of each circuit.

We turn now to the second of our semantic questions. What do the probabilities assigned to each elemental event mean? So far these assignments have no significance other than that they are consistent with the three axioms of probability. Clearly, it is necessary to assign a 'real' interpretation to the probabilities that we assign to events. We can assign any interpretation which satisfies (makes true) the three axioms of probability. Historically, two such interpretations have been made. The relative frequency interpretation interprets probabilities according to the frequencies with which outcomes occur when an experiment is executed many times. The probability as a measure of belief interpretation assigns a subjective measure of belief in the truth or falsity of an outcome.

In this thesis we adopt the relative frequency interpretation as it is well suited to abductive problem solving. Here we consider that a good explanation is one which is quite frequently correct. We proceed to describe the relative frequency interpretation in more detail.

A *trial* is a single performance of a well defined 'real' experiment. At a trial we observe a single outcome. We say that event e *occurred* at this trial if it contains element e. The certain event occurs at every trial. The impossible event never occurs. If e1 and e2 are mutually exclusive then both events cannot occur at the same trial.

When an experiment under consideration is repeated n times, out of which event e occurs n_e times, then according to the relative frequency interpretation, its probability is defined as the limit of the relative frequency n_e/n of the occurrence of e.

$$\text{Prob(e)} = \lim_{n \to \infty} \frac{n_e}{n}$$

For large n, $\text{Prob(e)} \approx ne/n$

It is quite easily shown that this interpretation satisfies the three axioms of probability. As a result if we assign probabilities to elemental events according to this interpretation, then we can derive probabilities for arbitrary events, and assign to these numbers the same relative frequency interpretation.

We can interpret conditional probabilities according to the relative frequency interpretation. Consider an experiment which is repeated n times out of which event m occurs n_m times, and event $e \wedge m$ occurs n_{em} times. For large n:

$$\text{Prob(e}\,|\,\text{m)} = \frac{\text{Prob(e}\wedge\text{m)}}{\text{Prob(m)}} \approx \frac{n_{em}/n}{n_m/n} = \frac{n_{em}}{n_m}$$

If we discard all trials in which event m did not occur and retain the sub-sequence of n_m trials in which it occurred, then Prob(e|m)

equals the relative frequency of the occurrence of event e in that sub-sequence.

Finally, we can interpret event independence according to the relative frequency interpretation. Consider an experiment which is repeated n times out of which event m occurs n_m times, event e occurs n_e times and event $e \wedge m$ occurs n_{em} times. If e and m are independent events then for large n:

$$\text{Prob}(e) \approx n_e/n$$

$$\text{Prob}(e \mid m) \approx n_{em}/n_m$$

for independent events $\text{Prob}(e) = \text{Prob}(e \mid m)$,
therefore $n_e/n = n_{em}/n_m$

For independent events e and m, the relative frequency of the occurrence of event e in the original sequence of trials equals the relative frequency of the event e in the sub-sequence of of n_m trials in which event m occurred.

We are now in a position to describe what a Bayesian network means. Assume that we have a complete probability model for the experiment E. We show that a Bayesian network is an equivalent representation of this probability model.

As described earlier, terminal nodes are characterized by sentences of the form 'P(V=c) = x' where the x values of these sentences sum to one. These are interpreted as joint probability distributions over a single random variable. 'P(V=c) = x' is interpreted to mean $\text{Prob}(V^{-1}(c)) = x$ for experiment E.

Non-terminal nodes are characterized by sentences of the form $'P(V=c \mid B_1=b_1,\ldots,B_n=b_n) = x'$ where the x values of these sentences sum to one for each fixed set of conditioning values $b_1,\ldots,b_n$. Moreover, we have a normalized set of these sentences for each possible condition made from $b_1 \in \text{domain}(B_1),\ldots,b_n \in \text{domain}(B_n)$. $'P(V=c \mid B_1=b_1,\ldots,B_n=b_n) = x'$ is interpreted to mean

$Prob(V^{-1}(c) \mid B_1^{-1}(b_1) \cap ... \cap B_n^{-1}(b_n)) = x$ for experiment E. If $domain(B_1),...,domain(B_n)$ represent the ranges of random variables $B_1,...,B_n$ then this represents a complete conditional joint probability distribution.

We now justify our interpretation of the previously defined *domain* function. We wish to show that $domain(X_k)$ is the range of random variable X_k.

Proof. Let $X_1,X_2,...X_n$ represent a total ordering of a Bayesian network B's variables. Let this total ordering be consistent with the partial ordering represented by the DAG of B. Consider the terminal node X_1. Recall that $domain(X_1)$ consists of the set of X_1 values assigned by the probability sentences making up $P(X_1)$. Since by definition random variable X_1 partitions the set of experiment outcomes, this set represents the range of random variable X_1. We now inductively assume the domain(X) interpretation for variables which are $< X_k$ and show that as a result, the interpretation also holds for X_k. Recall that $domain(X_k)$ consists of the set of X_k values assigned by the conditional probability sentences making up $P(X_k \mid P_1,...,P_n)$ where $P_1,...,P_n$ are the parents of X_k. Since the parents of X_k are all $< X_k$ then we can form the set of possible conditions for $P_1,...,P_n$ by taking the cross product of the parent variable domains.[1] This forms a set of mutually exclusive conditions which partitions the set of experiment outcomes. Hence if we enumerate possible X_k values over these conditions we generate the range of random variable X_k.

Bayesian networks embody the following implicit independence assumption [Pearl,88]:

---

[1]Some of these parent combinations may have zero-valued probabilities and hence cannot serve as conditions in a conditional probability. In this case the Bayesian network simply does not include a conditional probability distribution for this parent condition. This does not invalidate the argument presented here.

> Each random variable x is conditionally independent of all its non-descendants, given its parents π_x. Furthermore, no proper subset of π_x satisfies this condition.

As a result of this independence condition we can multiply the probability distributions for each Bayesian network node to form a single probability distribution over all the random variables. This is true since by the chain rule:

$$P(X_1,X_2,...X_n) = \prod_{i=1}^{n} P(X_i \mid Prev_i)$$

where $Prev_i$ is the set of variables X_j where $j < i$.

If $X_1,X_2,...X_n$ represents a total ordering on the variables which is consistent with the partial ordering represented by the Bayesian network DAG, then by the implicit independence assumption we have:

$$P(X_1,X_2,...X_n) = \prod_{i}^{n} P(X_i \mid \pi_{x_i})$$

To summarize, a Bayesian network represents a complete probability model. We adopt relative frequency semantics for the numeric probabilities. We can interpret the associated experiment as the Herbrand experiment. Alternatively, we can interpret the experiment as a set of real world outcomes with measurable properties.

3.2.3. Bayesian Programs

In order to develop the semantics of Bayesian programs it is necessary to develop additional results for both DAGs and Bayesian networks.

Let D = (V,E) be a DAG and let S be a subset of V. We say that a vertex v is *barren* wrt S defined on D if:

(a) $v \in S$

(b) v has no direct successors in S

Let D = (V,E) be a DAG. We say that S is a *complete subset* of V wrt D if:

(a) $V \supseteq S$

(b) if $r \in S$ then $S \supseteq prev(r)$

We note that if A is a vertex of DAG D = (V,E), then prev(A) is a complete subset of V wrt D. We also note that if S1 and S2 are both complete subsets of V wrt D then so is $S1 \cup S2$. Combining these two observations, we note that for any set of V vertices $\{v_1,v_2,...,v_n\}$, the set $\{prev(v_1) \cup prev(v_2) \cup ... \cup prev(v_n)\}$ is a closed subset of V wrt D.

Theorem 3.1. If D = (V,E).is a DAG and C is a complete subset of V wrt D and B is the set defined as B = { b | b is barren wrt C defined on D } then $C = \bigcup_{b \in B} prev(b)$.

Proof. First we prove that $\bigcup_{b \in B} prev(b) \supseteq C$. If $c \in C$ then either:

(a) c has no direct successors in C. Therefore c is an element b of B. Therefore c is an element of prev(b). Therefore c is an element of $\bigcup_{b \in B} prev(b)$.

(b) c has direct successors in C. In a finite D, one of c's successors must be barren wrt C. Therefore c is an ancestor to an element b of B. Therefore c is an element of prev(b). Therefor v must be an element of $\bigcup_{b \in B} prev(b)$.

Next we prove that $C \supseteq \bigcup_{b \in B} prev(b)$. Since C is a complete subset of V wrt D then $C \supseteq prev(b)$ for each $b \in B$. Therefore

$C \supseteq \bigcup_{b \in B} prev(b)$.

Since $C \supseteq \bigcup_{b \in B} prev(b)$ and $\bigcup_{b \in B} prev(b) \supseteq C$ then $C = \bigcup_{b \in B} prev(b)$. This completes the proof. We say that B *spawns* the set C.

Theorem 3.2. If D = (V,E) is a DAG and v_1, v_2 are two distinct vertices of V and C is the set defined by $C = prev(v_1) \cap prev(v_2)$, then C is a complete subset of V wrt. D.

Proof. C is a subset of V since both $prev(v_1)$ and $prev(v_2)$ are subsets of V, hence so is their intersection. Let $c \in C$. By the definition of C, $c \in prev(v_1)$ and since $prev(v_1)$ is a complete subset of V wrt D, then $prev(v_1) \supseteq prev(c)$. Similarly $prev(v_2) \supseteq prev(c)$. Therefore $prev(v_1) \cap prev(v_2) \supseteq prev(c)$. Therefore by definition C is a complete.subset of V wrt. D.

Note that C is, by definition, the set of common weak ancestors of v_1 and v_2. Since C is a complete subset of V wrt D, then by theorem 3.1 we can spawn C with the set B consisting of all vertices which are barren wrt C defined on D. The elements of this set are by definition the recent weak ancestors of vertices v_1 and v_2. Thus, the following proposition:

Proposition 3.1. If D = (V,E).is a DAG and v_1, v_2 are two distinct vertices of V, and C is the set $C = \{ c \mid c \text{ is a common weak ancestor of } v_1 \text{ and } v_2 \}$ and $R = \{ r \mid r \text{ is a recent weak ancestor of } v_1 \text{ and } v_2 \}$ then $C = \bigcup_{r \in R} prev(r)$.

If A1,A2,...,An are nodes in a Bayesian network, and TA1, TA2,...,TAn are tail extensions for A1,A2,...An respectively, then

TA1,TA2,...TAn are said to be *consistent* if TA1 ∪ TA2 ∪ ...∪ TAn does not contain more that one assignment for the same random variable.

Theorem 3.3. If A1,A2,...,An are nodes in a Bayesian network, and T is a set of tail extensions T= {TA1,TA2,...,TAn} for A1,A2,....,An respectively, then T is consistent iff its elements are pairwise consistent.

Proof. If the union of any two elements of T results in multiple variable assignments, then so will the union of all elements of T. As a result we can state that T is consistent ⇒ all pairs of its elements are consistent. The reverse direction can be proved by contradiction. Under the assumption that all pairs of T elements are consistent and that T is inconsistent, then at least one random variable must be assigned more than one value. Since each element of T assigns precisely one value to each of its variables, this implies that at least two T elements must assign the same variable different values. Therefore we must have at least one pair of inconsistent T element pairs. This contradicts our assumption. which must be false. As a result we can state that all pairs of T elements are consistent ⇒ T is consistent.

Theorem 3.4. If A1 and A2 are two nodes in a Bayesian network with tail extensions TA1 and TA2 respectively then TA1 and TA2 are consistent iff they have identical sub-tail extensions for each recent weak ancestor of A1 and A2.

Proof. Let R be the set of all recent weak ancestors of A1 and A2. TA1 assigns values to each variable in prev(A1). Similarly, TA2 assigns values to each variable in prev(A2). Both TA1 and TA2 assign values to variables in the set of common weak ancestors CWA = prev(A1) ∩ prev(A2). By definition TA1 and TA2 are consistent iff x(TA1) = x(TA2) for all x ∈ CWA. With this notation x(TA1) refers to the value that TA1 assigns to random variable x.

Since by proposition 3.1, CWA = $\bigcup_{r \in R}$ prev(r) then we have that x(TA1) = x(TA2) for all x ∈ CWA iff x(TA1) = x(TA2) for all x ∈ prev(r) for all r ∈ R. This completes the proof.

Theorem 3.5. If A1,A2,...,An are nodes in a Bayesian network, and T is a set of tail extensions T= {TA1,TA2,...,TAn} for A1,A2,....,An respectively, then T is consistent iff each pair of T elements TAi, TAj have identical sub-tail extensions for each recent weak ancestor of Ai and Aj.

Proof. This follows directly from theorem 3.3 and theorem 3.4.

Theorem 3.6. If B = (N,E) is a Bayesian network for the joint probability distribution $P(X_N)$ and N' is a complete subset of N wrt B, and E' = {e | e∈ E ∧ tail(e) ∈ N' ∧ head(e) ∈ N' }, then B' = (N',E') is the Bayesian network corresponding to $P(X_{N'})$. We say that B' is a *sub-Bayesian network* of B.

Proof. We proceed in steps:

(1) Let S = {s| s is barren wrt N defined on B }. S spawns N since N is a complete subset of N wrt B.

(2) Let S' = {s'| s' is barren wrt N' defined on B' } and s ∈ S. We show that s ∈ N' ⇒ s ∈ S'. Equivalently, s ∉ S' ⇒ s ∉ N'. By definition s doesn't have a direct successor in N and since N' is a subset of N, then s also doesn't have a direct successor in N'. Therefore if s ∈ N' then s ∈ S'.

(3) We show that S' ⊇ S ⇒ N' = N. Since N' is a complete subset of N wrt B, then S' spawns N', and if S' includes S which spawns N, then S' must spawn a set which includes N. Therefore N' must include N. Since by definition N includes N', then N' = N and S' = S. We can restate this result as N ⊃ N' ⇒ ∃s(s∈ S∧s∉ S'). Combining steps 2 and 3 we have N ⊃ N' ⇒ ∃s(s ∈ S ∧ s ∉ N').

(4) We know that N ⊇ N'. If N = N',the theorem is trivially true. Let us consider the case where N ⊃ N'. From step 3 we know that we can find at least one s ∈ S that is not an element of N'. Let

this node be called x_k. In general, let the elements of N be designated as $N = \{x_1, x_2, ..., x_k, ..., x_n\}$, then:

$$P(x_1, x_2, ..., x_k, ..., x_n) = \prod_{i=1}^{n} P(x_i \mid \pi_{x_i}).$$

Since x_k is barren wrt N defined on B we can write that:

$$P(x_1, x_2, ..., x_{k-1}, x_{k+1}..., x_n) = \sum_{c \in \text{domain}(x_k)} P(x_k = c \mid \pi_k) \prod_{i \neq k} P(x_i \mid \pi_{x_i})$$

And since $\sum_{c \in \text{domain}(x_k)} P(x_k = c \mid \pi_k) = 1$, then

$$P(x_1, x_2, ..., x_{k-1}, x_{k+1}..., x_n) = \prod_{i \neq k} P(x_i \mid \pi_i)$$

(5) Step 4 constructs a new Bayesian network without node x_k. Let $N^* = N - \{x_k\}$ and $E^* = E - \{ e \mid \text{head}(e) = x_k \}$ then $B^* = (N^*, E^*)$ is a Bayesian Network representing $P(x_1, x_2, ..., x_{k-1}, x_{k+1}..., x_n)$. We can show that N' is a complete subset of N*.wrt B*. First we show that N* includes N'. Since $N \supseteq N'$ then $N^* = N - \{x_k\} \supseteq N' - \{x_k\}$ and since $x_k \notin N'$ then $N^* \supseteq N'$. If we designate the prev function for B* as prev* to distinguish it from the original prev function defined for B, then we can write prev*(n) = prev(n) for each $n \in N'$, since the only edges removed from E in forming E* were those connecting to x_k which $\notin$ N'. Since N' is a complete subset of N wrt B then if $r \in N'$ then $N' \supseteq$ prev(r). Therefore if $r \in N'$ then $N' \supseteq$ prev*(r). Therefore N' is a complete subset of N* wrt B*.

(6) steps 4 and 5 can be used repeatedly to form new B*s with one fewer node. As long as $N^* \supset N'$, we can always find a node which is barren wrt N* defined on B* and which is not included in N'. We can remove this node to form a smaller superset of N'. Since there are a finite number of N* nodes which are not included in N', this process will terminate with B* = B' = (N',E'). This sub-Bayesian network of B represents the joint probability distribution:

$$P(x_{N'}) = \prod_{x_i \in N'} P(x_i \mid \pi_i) = \sum_{c \in \text{domain}(N-N')} P(x_{N'}, x_{N-N'} = c).$$

where domain(N-N') equals the cartesian product of the domain for each node in the set N-N'. This completes the proof.

Theorem 3.7. If A is a node in a Bayesian Network B = (N,E), A's parents are B1,B2,...Bn.and CA is a tail extension of A. then P(CA) > 0 iff:

(1)CB1,CB2,CB3,...,CBn are consistent tail extensions of B1,B2,...Bn
(2)CA = {A=a} ∪ CB1 ∪ CB2 ∪ ...∪ CBn
(3)P(CB1), P(CB2), P(CB3), ..., P(CBn) > 0
(4) $P(A=a \mid \pi_A(CA)) > 0$

Note that the notation $\pi_A(CA)$ denotes the subset of CA containing only the random variable value assignments for π_A, the parents of A.

Proof.

(1) By definition CA has precisely one assignment for each element of prev(A) = {A} ∪ ancestor(A). Since ancestor(A) = $\bigcup_{Bp \in \pi_A} \text{prev}(Bp)$ then this condition is true iff CA = {A=a} ∪ CB1 ∪ CB2 ∪ ...∪ CBn where CB1,CB2,CB3,...,CBn are consistent tail extensions of B1,B2,...Bn and a is an element of domain(A).

(2) Since ancestor(A) is a complete subset of N wrt B, we can write

$$P(CA) = P(A=a \mid \pi_A(CA)) \prod_{X_i \in \text{ancestor}(A)} P(X_i(CA) \mid \pi_{X_i}(CA)).$$

Therefore P(CA) > 0 iff

$P(A=a \mid \pi_A(CA)) > 0$ and $\prod_{X_i \in \text{ancestor}(A)} P(X_i(CA) \mid \pi_{X_i}(CA)) > 0.$

(3) $\prod_{X_i \in \text{ancestor}(A)} P(X_i(CA)|\pi_{x_i}(CA)) > 0$ iff

$P(X_i(CA)|\pi_{x_i}(CA)) > 0$ for all $X_i \in$ ancestor(A).

(4) Since ancestor(A) = $\bigcup_{Bp \in \pi_A}$ prev(Bp) then we can write that

$P(X_i(CA)|\ \pi_{x_i}\ (CA)) > 0$ for all $X_i \in$ ancestor(A) is true iff

$P(X_i(CA)|\pi_{x_i}(CA)) > 0$ for all $X_i \in$ prev(Bp) for all $Bp \in \pi_A$

(5) For CA = {A=a} $\cup$ CB1 $\cup$ CB2 $\cup$ ...$\cup$ CBn, $X_i(CA) = X_i(CBp)$ for all $X_i \in$ prev(Bp) for all $Bp \in \pi_A$. Therefore $P(X_i(CA)|\pi_{x_i}(CA)) > 0$ for all $X_i \in$ ancestor(A) is true iff

$P(X_i(CBp)|\pi_{x_i}(CBp)) > 0$ for all $X_i \in$ prev(Bp) for all $Bp \in \pi_A$

(6) $P(X_i(CBp)|\pi_{x_i}(CBp)) > 0$ for all $X_i \in$ prev(Bp) for all $Bp \in \pi_A$ is true iff

$\prod_{X_i \in \text{prev}(Bp)} P(X_i(CBp)|\pi_{x_i}(CBp)) > 0$ for all $Bp \in \pi_A$.

This product = P(CBp) since Prev(Bp) is a complete subset of N wrt B. This completes the proof.

We now relate the above results to Bayesian programs. As defined earlier a tail extension of a Bayesian network node A is a set of variable assignments, one for each element of prev(A). Consider a simple three node Bayesian network where nodes B and C are parents to node A. The representation {A=a,B=b,C=c} refers to a tail extension of node A. Bayesian programs include representations of *structured tail extensions* which preserve the underlying causal relationships between variables. For example, the tail extension of A is represented as the term A/a $\leftarrow$ [B/b $\leftarrow$ [], C/c $\leftarrow$ []] . The node term A/a represents the assignment for A. The list of extensions [B/b $\leftarrow$ [], C/c $\leftarrow$ []] represents the set of tail extensions of A's parents. The structured tail extension TA' of A can be constructed from the tail extension TA of A and the original Bayesian network

B = (N,E); TA' = struct(TA,E). Similarly, a tail extension of A can be derived from a structured tail extension of A; TA = unstruct(TA'). For convenience, we often refer to TA as simply a tail extension.and disambiguate based on the context that it is used in. In expressions of the form P(TA), TA is understood as the unstructured version. In the context of node_A(TA), TA is interpreted as the structured version.

In the previous section, where the semantics of probability models was discussed, we distinguished between two aspects of the semantics. We adopt a similar approach here. Firstly, we consider the semantics of the base program for a Bayesian program. The base program is a definite program whose least Herbrand model defines the experiment of a probability model. It enumerates the experiment outcomes, and the random variables, or properties of these outcomes. Having established this, we proceed to show how Bayesian programs assign probabilities to each elemental event. In this way the equivalence of Bayesian programs and probability models is established.

Theorem 3.8. Let BP be a Bayesian program based on a Bayesian network B = (N,E). Let BP' be the base program for BP and let M_H be the least Herbrand model for BP'. If node_A is a node predicate for node A in BP', then node_A(X) $\in M_H$ iff X is a tail extension of A and P(X) > 0.

Proof. We prove by induction on a total ordering ρ of the nodes consistent with the partial ordering defined by B. This total ordering enables us to compare any two nodes. For nodes A and B, we say that A $<_\rho$ B iff A precedes B in the total ordering ρ. We show that the theorem is true for terminal nodes. Then we show that if the theorem is true for all nodes which are $<_\rho$ A for some node A then it is true for node A as well. We refer to the fixpoint function for BP' as T_P.

Let node T represent a terminal node. By definition an explicit representation of node_T consists of ground node_T clauses of the form {node_T(T/t ← []), x} iff P(T=t) = x > 0. Therefore by the definition of T_P, node_T(T/t←[]) ∈ M_H iff P(T=t) > 0. Alternatively, a generative definition of node_T may be used. Based on the way that generative representations are constructed from explicit representations, and the declarative semantics of definite programs, explicit and generative definitions of node_T are equivalent. They result in the same ground atoms for node_T in M_H. Therefore the theorem is true for terminal nodes.

Now consider the non-terminal node A. From theorem 3.7 we know that if TA is a tail extension of A and B1,B2,...,Bn are the parents of A then P(TA) > 0 iff

(1)TB1,TB2,...,TBn are consistent tail extensions of B1,B2,...Bn

(2)TA = {A=a} ∪ TB1 ∪ TB2 ∪ ...∪ TBn

(3)P(TB1), P(TB2), P(TB3), ..., P(TBn) > 0

(4) P(A=al π_A(TA)) > 0 According to the rules for creating node_A clauses, we have a single defining definite program clause for node_A. Instances of this clause have the form

node_A(A/a ← [TB1,TB2,...,TBn]):-

node_B1(TB1),node_B2(TB2),...,node_Bn(TBn),

family_A(A/a,B1/b1,...,Bn/bn)

where shared clause variables establish additional conditions, namely:

(5) TB1,TB2,...,TBn are extension terms with associated node terms B1/b1,B2/b2,...,Bn/bn.

(6) For any pair of elements, TBi,TBj from the set T={TB1,TB2,...,TBn}, TBi and TBj share common sub-tail extensions for each recent weak ancestor of Bi,Bj.

Since this is the only defining rule for node_A, then according to the function Tp, node_A(TA) is an element of H_M iff:

(6) is true

(7) TA has the form A/a ← [TB1,TB2,...,TBn]

(8) node_B1(TB1),node_B2(TB2),...,node_Bn(TBn) are all $\in M_H$

(9) (5) is true and family_A(A/a,B/b1,...,Bn/bn) is $\in M_H$.

Together with the induction assumption, these conditions can be shown to be equivalent to conditions (1),(2),(3),(4) above.

By the induction assumption, node_B1(TB1),node_B2(TB2),...,node_Bn(TBn) are all $\in M_H$ iff TB1,TB2,...,TBn are tail extensions of B1,B2,...,Bn respectively and P(TB1),P(TB2),...,P(TBn) > 0. If TB1,TB2,...,TBn are tail extensions of B1,B2,...,Bn then by theorem 3.5, (6) is true ≡ TB1,TB2,...,TBn are consistent. If TB1,TB2,...,TBn are consistent tail extensions for B1,B2,...,Bn then condition (7) becomes equivalent to condition (2).and condition (5) ensures that b1,b2,...,bn are the values assigned by TA to B1,B2,...,Bn. Finally, by definition an explicit representation of family_A consists of ground clauses of the form {family_A(A/a,B/b1,...,Bn/bn). ,x} iff P(A=a| B=b1,...,Bn=bn) =x > 0 . Therefore, by the definition of T_P , family_A(A/a,B/b1,...,Bn/bn) $\in M_H$ iff P(A=a| B=b1,...,Bn=bn) > 0. Alternatively, a generative definition of family_A may be used. Based on the way in which generative representations are constructed from explicit representations, and the declarative semantics of definite programs, explicit and generative definitions of family_A are equivalent. They result in the same ground atoms for family_A in M_H. This completes the proof.

We have shown that the least Herbrand model of BP' enumerates the *possible* tail extensions of Bayesian network nodes. Conversely, ground atoms of node predicates which are not elements of the least Herbrand model are false or not possible. This is in contrast to the declarative semantics of a definite program. A ground atom which is not an element of a definite program's least Herbrand model may or may not be false. The often used *closed world assumption* is the assumption that such atoms are false. By the semantics of a Bayesian program, such atoms *must* be false.

We now show that the numeric clause annotations of the Bayesian program BP define probabilities for Bayesian network extensions. Our approach is to extend the definition of the fixpoint function Tp.

Let P be a joint probability distribution over random variables $X_1, X_2, \ldots, X_n$. We refer to syntactic probability sentences such as 'P(X1=x_1)' or 'P(X1=x_1| X_2=x_2)', for the random variables of P as *symbolic probabilities.* Under the semantic interpretation discussed earlier, each symbolic probability can be associated with a numeric probability. For example 'P(X1=x_1)' can be associated with P(X1=x_1) which is a real number between 0 and 1 inclusive. An *assumption* is a symbolic probability whose associated numeric probability is a real number between 0 and 1 exclusive.

Let Cθ be a ground instance of a Bayesian clause C. Cθ has an associated *clause assumption* A(Cθ) which is defined as follows:

(a) if C is an annotated terminal node clause for node A with probability annotation x, then A(Cθ) = 'P(A=a)' where a is the assignment for A appearing in Cθ. 'P(A=a)' is an assumption with associated numeric value P(A=a) = x.

(b) if C is an annotated family clause for node A with parents B1,...,Bn and with clause probability annotation x, then A(Cθ) = 'P(A=a|B_1=b_1,...,B_n=b_n)' where a,b_1,...,b_n are the assignments for A,B_1,...,B_n appearing in Cθ. 'P(A=a|B_1=b_1,...,B_n=b_n)' is an assumption with associated numeric value P(A=a|B_1=b_1,...,B_n=b_n) = x.

(c) else A(Cθ) = nil

A set of assumptions is termed an *environment.* [1]

Each ground instance of a Bayesian clause Cθ, has an associated *clause environment* Env(Cθ) which is defined as follows:

(a) if A(Cθ) = nil then Env(Cθ) = ∅

[1] This terminology anticipates the use of an ATMS based architecture as described later.

(b) else Env(Cθ) = {A(Cθ)}

Let BP be a Bayesian program and let A_{BP} be the set of all clause assumptions for BP. A_{BP} is referred to as the *assumption base* for BP. Let E_{BP} be the powerset of A_{BP}. E_{BP} is referred to as the *environment base* for BP.

Let BP be a Bayesian program, BP' be the base program for BP, B_P the Herbrand base associated with BP' and E_{BP} the environment base for BP. The fixpoint function of BP' is referred to as Tp. We define the function Tp'(I)={ (A,E(A)) | A ∈ B_P and E(A) ∈ E_{BP} and the clause C = A :- A1,...,An is a ground instance of a clause in BP' and I ⊇ { (A1,E(A1)),...,(An,E(An)) } and E(A) = Env(C) ∪ E(A1) ∪ E(A2) ∪ ... ∪ E(An) }

Recall that M_H, the least Herbrand model for BP' is defined by $M_H = Tp^{\infty}(\varnothing)$. Likewise we define M_B, the *Bayesian model* for BP as $M_B = Tp'^{\infty}(\varnothing)$. In comparing Tp' and Tp we see that Tp' maps sets of ground atoms to sets of ground atoms in exactly the same way that Tp does. The difference is that Tp' also defines an environment for each ground atom included in the output set. As a result, M_B contains the same ground atoms that M_H does, but each ground atom has an associated environment.

Let BP be a Bayesian program with associated Bayesian Model M_B. Let E_{BP} be the environment base for BP, BP' be the base program for BP and M_H be the least Herbrand model for BP'. M_B is a set consisting of elements of the form (A,E(A)) where A ∈ M_H and E(A) ∈ E_{BP} . We refer to the function E as the *environment function* for BP. It assigns each A ∈ M_H an environment[1].

---

[1]The environment function actually maps *sets* of ground atoms to an environment. This supports a more general usage of the environment function, namely that it assign environments to answers of Bayesian queries. As a result E(A) is more precisely expressed as E({A}). By convention we just write E(A).

We now show that the environments of ground node atoms correspond to tail extension probabilities.

Theorem 3.9. Let BP be a Bayesian program with Bayesian model M_B. If family_A is a family predicate for node A in BP, then (family_A(A/a, $B_1/b_1,...,B_n/b_n$) ,E(family_A(A/a, $B_1/b_1,...,B_n/b_n$)) $\in$ M_B iff

(1) $a,b_1,...b_n$ are valid assignments for nodes $A,B_1,...B_n$

(2) $P(A=a|B_1=b_1,...,B_n=b_n) > 0$

(3) E(family_A(A/a, $B_1/b_1,...,B_n/b_n$)) =

$$\begin{cases} \{ P(A=a|B_1=b_1,...,B_n=b_n) \} & \text{if } P(A=a|B_1=b_1,...,B_n=b_n)<1 \\ \varnothing & \text{otherwise} \end{cases}$$

Proof. This follows directly from:
(1) the definition of T_P' and Env

(2) the criteria used to define explicit family clauses for Bayesian programs

(3) the equivalence of generative and explicit definitions for family predicates.

Theorem 3.10. Let BP be a Bayesian program. If node_A is a node predicate for node A in BP, then (node_A(TA), E(node_A(TA))) $\in$ M_B iff

(1) TA is a tail extension of A

(2) $P(TA) > 0$

(3) $E(node\_A(TA)) = \bigcup_{x \in S_{TA}} \{P(X(TA)|\pi_x(TA))\}$[1]

where $S_{TA} = \{ X \mid X \in prev(A) \wedge 0 < P(X(TA)|\pi_x(TA)) < 1 \}$.

---

[1]Note that we use symbolic probabilities like '$P(X(CA)|\pi_x(CA))$' in two ways. In set expressions like $E = \bigcup_{x \in S_{CA}} P(X(CA)|\pi_x(CA))$ it is a symbolic entity. In product

Proof. We know from theorem 3.8 that node_A(TA) $\in M_H$, the least Herbrand model for BP' where BP' is the base program of BP, iff TA is a tail extension of A and P(TA) > 0. Since M_B has the same atoms as M_H we need only prove that the environment E, assigned by Tp' to each A in M_H is equal to $\bigcup_{x \in S_{TA}} \{P(X(TA)|\pi_x(TA))\}$.

We prove by induction on a total ordering ρ of the nodes consistent with the partial ordering defined by B. From the definition of Tp and the equivalence of explicit and generative definitions for terminal node predicates, this equality is true for terminal nodes. We show that if the equality is true for all nodes which are $<_\rho$ A for some node A then it is true for A as well.

Consider the non-terminal node A with parents B1,B2,...,Bn. BP has a single defining definite program clause for node_A. As described in the previous proof, instances of this clause have the form:
node_A(A/a ← [TB1,TB2,...,TBn]):-node_B1(TB1), node_B2(TB2), ...,
node_Bn(TBn), family_A(A/a,B1/b1,...,Bn/bn)

From the results of the previous proof node_A(TA) is $\in M_H$ iff:

(1) node_B1(TB1), node_B2(TB2),...node_Bn(TBn) $\in M_H$

(2){TB1,TB2,...,TBn} is consistent

(3) TA = A/a ← [TB1,TB2,...,TBn]

(4) family_A(A/a,B/b1,...,Bn/bn) $\in M_H$ where tail extensions TB1,TB2,...,TBn have associated node terms B1/b1,B2/b2,...,Bn/bn.

By theorem 3.9 we have that
E(family_A(A/a,B/b1,...,Bn/bn)) = {P(A=a|B=b1,...,Bn=bn)}. [1]
By the inductive assumption we have that

---

expressions of the form $E = \prod_{x \in S_{CA}} P(X(CA)|\pi_x(CA))$ it refers to its associated numeric probability. The expression context can be used to disambiguate.

$$E(node\_Bi(TBi)) = \bigcup_{x \in S_{TBi}} \{P(X(TBi)|\pi_x(TBi))\} \quad [2]$$

where $S_{TBi} = \{ x \mid x \in prev(Bi) \wedge 0 < P(X(TBi)|\pi_x(TBi)) < 1 \}$.

Using the definition of Tp',

$E(node\_A(TA)) = E(node\_B1(TB1)) \cup E(node\_B2(TB2)) \cup ... \cup E(node\_Bn(TBn)) \cup E(family\_A(A/a,B/b1,...,Bn/bn))$. [3]

If we substitute [1] and [2] into [3] and take into account that {TB1,TB2,...,TBn} is consistent, then we have:

$$E(node\_A(TA)) = \bigcup_{x \in S_{TA}} \{P(X(TA)|\pi_x(TA))\}$$

where $S_{TA} = \{ x \mid x \in prev(A) \wedge 0 < P(X(TA)|\pi_x(TA)) < 1 \}$ This completes the proof.

Since for possible tail extensions TA of A,

$$P(TA) = \prod_{x \in prev(A)} P(X(TA)|\pi_x(TA)) = \prod_{x \in S_{TA}} P(X(TA)|\pi_x(TA))$$

where $S_{TA} = \{ x \mid x \in prev(A) \wedge 0 < P(X(TA)|\pi_x(TA)) < 1 \}$ then the environment of an M_H node atom can be converted into the numeric probability of the node atom's tail extension by replacing each environment assumption with its numeric probability, and forming the product.

We now provide a declarative definition of a correct answer for a query to a Bayesian program.

A *Bayesian goal* for a Bayesian program is a definite goal for which:

(1) all subgoal predicates are node predicates.

(2) subgoal arguments guarantee a consistent set of tail extensions for the complete set of subgoal tail extensions. In other words, the subgoals of a Bayesian goal are network consistent.

Let BP be a Bayesian program and G a Bayesian goal. Let BP' be the base program for BP. An *answer* for BP union {G} is (θ,ε) where θ is an answer to BP' union {G} and $\varepsilon \in E_{BP}$, the environment base for BP.

Let BP be a Bayesian program with base program BP', G be a Bayesian goal ?- $A_1,...,A_k$ and E be the environment function for BP. We say that (θ,ε) is a *correct answer* to BP $\cup$ {G} iff:

(1) θ is a correct answer to BP' $\cup$ {G}

(2) θ is a ground substitution

(3) $\varepsilon = E(G\theta) = E(A_1\theta) \cup E(A_2\theta) \cup ... \cup E(A_k\theta)$[1]

Comparing the above definition for correct answer with the earlier definition for a correct answer to a definite program, the above definition imposes additional restrictions on θ, namely that it make $(A_1,...,A_k)\theta$ ground.

Theorem 3.11. Let BP be a Bayesian program with Bayesian model M_B, and G be a Bayesian goal ?- $A_{x_1},...,A_{x_k}$. Under this notation A_{x_1} is a subgoal for node x_1. If (θ,ε) is a correct answer to BP union {G} then

$$\varepsilon = \bigcup_{x \in S_e} \{P(X(ext)|\pi_x(ext))\}$$

where $ext = \bigcup_{i=1}^{k}$ (tail extension argument of $(A_{x_i})\theta$)

[1] This extends the definition of the environment function which previously was defined only for single ground atoms.

and $S_e = \{x \mid x \in \bigcup_{i=1}^{k} prev(x_i) \wedge 0 < P(X(ext)|\pi_x(ext)) < 1 \}$

Moreover, P(ext) can be derived from ε; $P(ext) = \prod_{x \in S_e} P(X(ext)|\pi_x(ext))$

Proof. Let T_{x_i} = (the tail extension argument of $(A_{x_i})\theta$.) We know that $((A_{x_i})\theta,\varepsilon_{x_i})) \in M_B$. By theorem 3.10 $\varepsilon_{x_i} = \bigcup_{x \in S_{T_i}} \{P(X(T_{x_i})|\pi_x(T_{x_i}))\}$

where
$S_{T_i} = \{x \mid x \in prev(x_i) \wedge 0 < P(X(T_{x_i})|\pi_x(T_{x_i})) < 1 \}$

Since (θ,ε) is a correct answer to BP union {G} we have that $\varepsilon = \varepsilon_{x_1} \cup \varepsilon_{x_2} \cup ... \cup \varepsilon_{x_k}$ .Substituting for ε_{x_i} and taking into account that $\{T_{x_1}, T_{x_2}, ..., T_{x_k}\}$ is consistent leads to the desired result for E.

Since $\bigcup_{i=1}^{k} prev(x_i)$ is a complete subset of BP[1], then we can form P(ext) from ε by replacing symbolic probabilities with their numeric values and forming the product over all set elements. This completes the proof.

P(ext), as defined for the above proof, can be used to rank each correct answer relative to other correct answers. Let obs be the set of all node assignments established by ground terms in the Bayesian goal G of the preceding theorem. For example, obs might be {A=a} where A is one of the nodes for which ext has a value assignment. In

---

[1]We use a slight abuse of language here. It is understood that BP is based on the Bayesian network B=(N,E) and that $\bigcup_{i=1}^{k} prev(x_i)$ is a complete subset of N wrt B.

this case obs ensures that the 'ext' of all correct answers must assign A=a. By Baye's theorem we have that P(ext|obs) = P(ext $\wedge$ obs)/P(obs). By the chain rule P(ext $\wedge$ obs) is equal to P(obs|ext)P(ext). Depending on whether ext includes obs or not, P(obs|ext) is either one or zero respectively. Therefore, for answers which are correct, P(ext|obs) = P(ext)/P(obs). In this expression, P(obs) is simply a normalization factor and is the same for all correct answers. Therefore P(ext) ranks the correct answers to BP $\cup\{G\}$.

3.3. Procedural Semantics

In the following we define the procedural semantics of definite programs, and Bayesian programs.

3.3.1. Definite Programs

We present definitions leading to the definition of a computed answer, which is then compared with the previously defined correct answer.

Let $\theta = \{u_1/s_1,...,u_m/s_m\}$ and $\sigma = \{ v_1/t_1,...,v_n/t_n\}$ be substitutions. Then the *composition* $\theta\sigma$ of θ and σ is the substitution obtained from the set $\{u_1/s_1\sigma,...,u_m/s_m\sigma,v_1/t_1,...,v_n/t_n\}$ by deleting any binding $u_i/s_i\sigma$ for which $u_i=s_i\sigma$ and deleting any binding v_j/t_j for which $v_j\in \{u_1,...,u_m\}$.

Let E and F be expressions. we say E and F are variants if there exist substitutions θ and σ such that $E = F\theta$ and $F = E\sigma$. We also say E is a variant of F or F is a variant of E.

Let S be a finite set of simple expressions. A substitution θ is called a unifier for S if $S\theta$ is a singleton. a unifier θ for S is called a *most general unifier* (mgu) for S if, for each unifier σ of S, there exists a substitution γ such that $\sigma=\theta\gamma$.

Several unification algorithms exist [Lloyd,84] for finding the mgu of a finite set of simple expressions S. If S is unifiable such unification algorithms terminate and return a mgu for S. If S is not unifiable, then such unification algorithms terminate and report this fact.

Let G be ?- $A_1,...,A_m,...,A_k$ and C be A:- $B_1,...,B_q$. Then G' is derived from G and C using mgu θ if the following conditions hold:

(a) A_m is an atom, called the *selected atom*, in G.

(b) θ is an mgu of A_m and A.

(c) G' is the goal ?- $(A_1,...,A_{m-1},B_1,...,B_q,A_{m+1},...,A_k)\theta$

G' is called a *resolvent* of G and C.

Let P be a definite program and G a definite goal. An *SLD-derivation* of $P \cup \{G\}$ consists of a (finite or infinite) sequence $G_0=G,G_1,...$, of goals, a sequence $C_1,C_2,...$ of variants of program clauses of P and a sequence $\theta_1,\theta_2,...$ of mgu's such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}.

Each C_i is a suitable variant of the corresponding program clause so that C_i does not have any variables which already appear in the derivation up to G_{i-1}. Each program clause variant C_i is called an *input clause* of the derivation.

An *SLD-refutation* of $P \cup \{G\}$ is a finite SLD-derivation of $P \cup \{G\}$ which has the empty clause as the last goal in the derivation. If G_n equals the empty clause, we say the refutation has *length n*.

Let P be a definite program and G a definite goal. A *computed answer* θ for $P \cup \{G\}$ is the substitution obtained by restricting the composition $\theta_1...\theta_n$ to the variables of G, where $\theta_1,...,\theta_n$ is the sequence of mgu's used in an SLD-refutation of $P \cup \{G\}$.

The soundness of SLD-Resolution theorem [Lloyd,84] states that every computed answer is correct. Let P be a definite program and G

a definite goal. then every computed answer for P ∪ {G} is a correct answer for P ∪ {G}.

The completeness of SLD-resolution theorem [Lloyd,84] states that every correct answer is an instance of a computed answer. Let P be a definite program and G a definite goal. For every correct answer θ for P ∪ {G}, there exists a computed answer σ for P ∪ {G} and a substitution γ such that θ = σγ.

We briefly discuss the concept of a computation rule, which is used to select atoms in an SLD-derivation.

A *computation rule* is a function from a set of definite goals to a set of atoms such that the value of the function for a goal is an atom, called the selected atom, in that goal.

Let P be a definite program, G a definite goal and R a computation rule. An *SLD-refutation of P ∪ {G} via R* is an SLD-refutation of P ∪ {G} in which the computation rule R is used to select atoms.

Let P be a definite program, G a definite goal and R a computation rule. An *R-computed answer* for P ∪ {G} is a computed answer for P ∪ {G} which has come from an SLD-refutation of P ∪ {G} via R.

The independence of the computation rule theorem [Lloyd,84] states that for any choice of computation rule, if P ∪ {G} is unsatisfiable, we can always find a refutation using the given computation rule. Let P be a definite program and G a definite goal. Suppose there is an SLD-refutation of P ∪ {G} with computed answer σ. Then, for any computation rule R, there exists an SLD-refutation of P ∪ {G} via R with R-computed answer σ' such that Gσ' is a variant of Gσ.

3.3.2. Bayesian Programs

As with definite programs, we present definitions leading to the definition of a computed answer, which is then compared with the

previously defined correct answer. We refer to the modified resolution procedure as *SLDB resolution.*

Let BP be a Bayesian program with base program BP' and G be a Bayesian goal. A *computed answer* for BP $\cup$ {G} is (θ,ε) such that:

(a) θ is a computed answer resulting from S, an SLD-refutation of BP' $\cup$ {G} of length n.

(b) $\varepsilon = \bigcup_{i \in 1}^{n} \text{Env}(C_i\theta)$ where $C_1, C_2, \ldots, C_n$ is the sequence of input clauses in S

For comparison, we repeat the earlier definition of a correct answer to BP $\cup$ {G}.

Let BP be a Bayesian program with base program BP', G be a Bayesian goal ?- $A_1,\ldots,A_k$ and E be the environment function for BP. We say that (θ,ε) is a *correct answer* to BP $\cup$ {G} iff:

(1) θ is a correct answer to BP' $\cup$ {G}

(2) θ is a ground substitution

(3) $\varepsilon = E(G\theta) = E(A_1\theta) \cup E(A_2\theta) \cup \ldots \cup E(A_k\theta)$[1]

In the following we show first of all, that condition (a) of the definition for a computed answer to BP $\cup$ {G} is equivalent to condition (1) and (2) of the definition of a correct answer to BP $\cup$ {G}. We will then proceed to show that condition (b) of the definition for a computed answer to BP $\cup$ {G} is equivalent to condition (3) of the definition of a correct answer to BP $\cup$ {G}. In so doing, we establish the completeness and soundness of SLDB resolution.

We note that by definition the base programs of Bayesian programs are ground definite programs.

---

[1] This extends the definition of the environment function which previously was defined only for single atoms.

Theorem 3.12. If P is a ground definite program, G is a definite goal and θ is a computed answer to $P \cup \{G\}$, then θ is a ground substitution.

Proof. We prove by induction on the length of the computed answer's SLD-refutation of $P \cup \{G\}$.

First consider the case where the refutation length is 1. G is of the form ?- A_1 and P has a unit clause of the form A. We have that G_1 = the empty clause under mgu θ_1. Therefore $A_1\theta_1 = A\theta_1$. Since P ia a ground definite program, A is ground. Therefore $A_1\theta_1 = A$ and θ_1 is a ground substitution.

We now show that if the theorem is true for SLD refutations of length n-1 then it is true for SLD-refutations of length n. Let $G = ?- A_1,...,A_m,...,A_k$ be a goal for which there is an SLD-refutation of length n. Let $C = A:- B_1,...,B_q$ be the first input clause of this SLD-refutation, and let A_m be the selected atom of G. The SLD-refutation derives G_1 from G and C using mgu θ_1. $G_1 = ?-(A_1,...,A_{m-1},B_1,...,B_q,A_{m+1},...,A_k)\ \theta_1$. Since G has an SLD-refutation of length n then G_1 has an SLD-refutation of length n-1. By the inductive assumption $(A_1,...,A_{m-1},B_1,...,B_q,A_{m+1},...,A_k)$ $\theta_1...\theta_n$ is ground. Therefore $(B_1,...,B_q)\theta_1...\theta_n$ is ground. Therefore since the head of C does not include any variables which are not in the body of C, then $(A_1,...,A_{m-1}A_m,A_{m+1},...,A_k)\theta_1...\theta_n = (G)\theta_1...\theta_n$ is ground. Thus we have that the computed answer $\theta_1...\theta_n$ to $P \cup \{G\}$ is a ground substitution. This completes the proof.

Theorem 3.13. Let P be a ground definite program and G a definite goal. Then:

(a)Each computed answer for $P \cup \{G\}$ is a correct answer for $P \cup \{G\}$

(b)Each correct answer for $P \cup \{G\}$ is a computed answer for $P \cup \{G\}$

Proof. (a) follows directly from the soundness of SLD-Resolution theorem. (b) follows from the completeness of SLD-Resolution theorem and from theorem 3.12 above. By the completeness of SLD-resolution theorem, for every correct answer θ for $P \cup \{G\}$, there exists a computed answer σ for $P \cup \{G\}$ and a substitution γ such that $\theta = \sigma\gamma$. By theorem 3.12, σ is ground. Therefore $\theta = \sigma$ and γ is the identity substitution.

Theorem 3.14. If P is a ground definite program and G is a definite goal and θ is a correct answer to $P \cup \{G\}$, then θ is a ground substitution.

Proof. This follows directly from theorem 3.12 and theorem 3.13 above.

We can now incorporate this result into our original definition for a correct answer to a Bayesian goal.

Theorem 3.15. Let BP be a Bayesian program with base program BP', G be a Bayesian goal ?- $A_1,...,A_k$ and E be the environment function for BP. Then (θ,ε) is a *correct answer* to $BP \cup \{G\}$ iff:

(1) θ is a correct answer to $BP' \cup \{G\}$

(2) $\varepsilon = E(G\theta) = E(A_1\theta) \cup E(A_2\theta) \cup ... \cup E(A_k\theta)$

Proof. This follows directly from the definition of a correct answer to $BP \cup \{G\}$ and from the fact that BP' is a ground definite program and from theorem 3.14.

Theorem 3.16. Let BP' be a Bayesian program and G a Bayesian goal. If (θ,ε) is a computed answer of $BP \cup \{G\}$ then $\varepsilon = E(G\theta)$ where E is the environment function for BP.

Proof. We prove this by induction on the length n of the SLD-refutation associated with a computed answer. We consider $G' = ?- A_1,...,A_n$ where G' is not necessarily a Bayesian goal as it may contain subgoals for predicates other than node predicates. We consider goals of this more general form since they occur in the

sequence of goals associated with an SLD-refutation for a Bayesian goal. Since G' is a more general goal type than a Bayesian goal, if the theorem holds for G' then it must also hold for Bayesian goals.

Suppose first that n=1. This means that G' is a goal of the form ?- A_1 and BP has a ground unit clause of the form A. and $A_1\theta_1 = A$. Therefore $\varepsilon = Env(A)$. By the definition of Tp, the fixpoint function for BP, $E(A) = Env(A)$. Therefore $\varepsilon = E(A) = E(G'\theta_1)$.

Now suppose that the result holds for computed answers derived from refutations of length n-1. Let $\theta_1,...,\theta_n$ be the mgu sequence of the SLD-refutation S for BP $\cup$ {G'}, $C = A{:-}\ B_1,...,B_q$ be the first input clause and A_m the selected atom of G'. Therefore the first derived goal of S is:
G'_1 = ?- $(A_1,...,A_{m-1},B_1,...,B_q,A_{m+1},...,A_n)\theta_1$. Since G'_1 has a refutation length of n-1, then by the inductive hypothesis $\varepsilon_1 = E(G'_1\theta')$ where $\theta' = \theta_2...\theta_n$. Therefore $\varepsilon = Env(C\theta) \cup E(G'_1\theta')$ where $\theta = \theta_1\theta'$.
Substituting for G'_1 we have that

$\varepsilon =$

$Env(C\theta)\cup E(A_1\theta)\cup...\cup E(A_{m-1}\theta)\cup E(B_1\theta)\cup...\cup E(B_q\theta)\cup E(A_{m+1}\theta)...\cup E(A_n\theta)$

It remains to show that $Env(C\theta)\cup E(B_1\theta)\cup...\cup E(B_q\theta) = E(A_m\theta)$. This is true by the definition of Tp and the fact that $A_m\theta{:-}\ B_1\theta,...,B_q\theta$ is a ground instance of C. This completes the proof.

We can now assert that SLDB resolution is complete and sound.

Theorem 3.17 Let BP be a Bayesian program and G a Bayesian goal. Then:

(a)Each computed answer for BP $\cup$ {G} is a correct answer for BP $\cup$ {G}

(b)Each correct answer for BP $\cup$ {G} is a computed answer for P $\cup$ {G}

Proof. Let BP' be the base program for BP. We consider (a) first: By the definition of a computed answer and theorem 3.13 if (θ,ε) is a computed answer to BP $\cup$ {G} then θ is a computed answer to BP' $\cup$ {G} which is also a correct answer to BP' $\cup$ {G}. By theorem 3.16 we also have that $\varepsilon = E(G\theta)$. Therefore (θ,ε) is a correct answer to BP $\cup$ {G}.

Now consider (b) above. If (θ,ε) is a correct answer for BP $\cup$ {G} then θ is a correct answer for BP' $\cup$ {G} which is also a computed answer to BP' $\cup$ {G}. Therefore there must exist an SLD-refutation for BP' $\cup$ {G} which results in θ. Therefore there must exist a computed ε' for this SLD-refutation such that (θ,ε') is a computed answer to BP $\cup$ {G}. Therefore by theorem 3.16, $\varepsilon' = E(G\theta)$ which equals ε by definition. Therefore (θ,ε) is a computed answer to BP $\cup$ {G}.

SLDB resolution is also independent of the computation rule.

Theorem 3.18. Let BP be a Bayesian program with base program BP' and G be a Bayesian goal. Suppose there is an SLD-refutation of BP' $\cup$ {G} resulting in computed answer (θ,ε) to BP $\cup$ {G}. Then, for any computation rule R, there exists an SLD-refutation of BP' $\cup$ {G} via R resulting in the same computed answer (θ,ε) to BP $\cup$ {G}.

Proof. Let BP' be the base program for BP. Since (θ,ε) is a computed answer for BP $\cup$ {G} then θ is a computed answer for BP' $\cup$ {G}. By the independence of computation rule theorem, for any computation rule R, there exists an SLD-refutation of BP' $\cup$ {G} via R with R-computed answer θ' such that $G\theta'$ is a variant of $G\theta$. There must also exist a computed ε' for this SLD-refutation such that (θ',ε') is a computed answer to BP $\cup$ {G}. Since $G\theta$ and $G\theta'$ are ground then $\theta = \theta'$. Therefore $\varepsilon' = E(G\theta') = E(G\theta) = \varepsilon$ and $(\theta',\varepsilon') = (\theta,\varepsilon)$.

3.4. Examples

The advantage of Bayesian programs over Bayesian networks is that whereas Bayesian networks are propositional, Bayesian

programs contain logical variables, thereby enabling one clause to take the place of many ground propositions. Consider, for example, a Bayesian network family of nodes X,Y and Z where Z is deterministically 'caused' by X and Y according to the equation Z = X + Y. This is represented as:

```
node_Z(_Z ← [_X ← _XA, _Y ←_YA] ) :- node_X(_X ← _XA),
                                     node_Y(_Y ← _YA).
                                     family_Z(_Z,_X,_Y).


family_Z(Z/_Z, X/_X, Y/_Y) :- plus(_Z,_X,_Y).
```

In this example, it is assumed that the 'plus' predicate is defined with the 'correct' plus semantics. Every instance of the family_Z clause has the same associated conditional probability, P(_Z|_X,_Y) = 1. As a result, ground 'plus' instances are represented generatively rather than explicitly enumerated.

Consider the case where X,Y,Z are boolean variables and *usually* Z = X ∧Y when S = ok. This is represented as:

```
node_Z(_Z ←[_S ← _SA,_X ← _XA,_Y ← _YA] ) :-
                                     node_X(_X ← XA),
                                     node_Y(_Y ← _YA),
                                     node_S(_S← _SA),
                                     family_Z(_Z,_X,_Y,_S).

{family_Z(Z/_Z,X/_X,Y/_Y,S/ok) :- and(_Z,_X,_Y ), .95}
{family_Z(Z/_Z,X/_X,Y/_Y,S/ok) :- nand(_Z,_X,_Y),.05}
{family_Z(Z/_Z,X/_X,Y/_Y,S/nok) :- bool(_Z),bool(_X),bool(_Y),
.5}
```

In this example, it is assumed that the predicates 'and', 'nand' and 'bool' are defined with the appropriate semantics. We can think of the Z node as a component in a model based diagnosis system. When the state S is ok, the component exhibits normal 'AND' behavior *most* of the time but not *all* of the time. The .05 probability summarizes

those outcomes for which the output of the 'and' component is faulty even though the 'and' component is healthy. This can occur, for example, if there is a short in the system that this component is a part of. In other words we allow for unmodeled causes of Z's value. In this way a complete diagnostic component model is constructed, something which is not possible in conventional model based diagnosis.

In effect, conventional model based diagnosis is implicitly based on the assumption that the component structure is fixed or given. This dramatically reduces the number of possibilities that need to be modeled. However, as a result we are not able to correctly diagnose failure modes which violate this assumption. In a Bayesian program we are able to entertain the possibility of this assumption not holding, without dramatically increasing modeling complexity. Rather than explicitly identifying additional 'causes' of the Z output, we simply model the ok-state component behavior as non-deterministic.

The final clause is an example of a nil constraint on component behavior. When the component state is 'not ok' then for any given _X,_Y pair, the _Z value is just as likely TRUE as FALSE. In this case, when the component is broken the component behavior is random.

3.5. Comparison with Probabilistic Horn Abduction

In [Poole,91] Poole describes a formalism, referred to as probabilistic Horn abduction, for representing Bayesian networks in a Horn clause logic. His formalism is similar to that described for Bayesian programs, but different.

A major difference is that Poole does not explicitly represent Bayesian network extensions. As a result, Poole must introduce non-Horn clauses to ensure that no assumption (annotated clause) is counted more than once. With Bayesian programs we accomplish this through the explicit representation of node tail extensions and the

use of network consistent subgoals in both Bayesian goals and node predicate definitions. As a result non-Horn clauses are not needed.

The semantics of Bayesian programs are also developed differently. Poole develops his formalism as a generalization of a simplified form of Theorist [Poole,87], a reasoning system for defaults and diagnosis. As a result his semantics is based on the notion of adding assumptions, which are equivocal and have associated probabilities, to unequivocally true clauses in order to construct a theory explaining a set of observations. Each theory thus constructed has its own least Herbrand model. Under probabilistic Horn abduction each such minimal model can be assigned a probability. We thus have a set of ranked possible worlds or minimal models to choose from. In contrast, with Bayesian programs we have a single least Herbrand model. Each element of the model is tagged with a probability representing the frequency with the atomic proposition occurs in the semantic domain.

A final difference is that Poole's assumptions are by definition atoms. With Bayesian programs we can provide non-atomic definitions for annotated family and terminal node predicates. It is this feature that allows us to exploit the expressivity of predicate calculus over propositional calculus, in order to succinctly represent assumptions which share the same associated probability.

Chapter 4
An Architecture for Abduction

In this chapter we present a problem solving architecture for Bayesian programs which is based on De Kleer's focused consumer architecture [de Kleer,86c], [de Kleer,86d], [Forbus,88a] Following a description of the basic architecture, we show how it may be adapted to a problem solving architecture for Bayesian programs.

4.1. Consumer Architecture

De Kleer describes several variations of the Consumer Architecture all of which are based on an ATMS. Indeed, De Kleer also distinguishes between several ATMS variants. The main distinguishing feature of these architectures is the degree to which problem solving control is focused. In the following we do not distinguish between the different architectures. Rather, we describe a single architecture which is suited to our purposes. We adopt the labeling algorithm described in [de Kleer,88].We also explicitly extend De Kleer's focusing mechanism to include label propagation. Although De Kleer suggests this approach in [Forbus,88a] it is not explicitly described.

Figure 4.1 illustrates the three components which make up the Consumer architecture. The *ATMS* maintains the current state of the problem solving, and supports queries on this information. The *Controller* focuses problem solving by establishing intermediate problem solving goals. The *scheduler* maintains an agenda of things to do, given the current problem solving state and the intermediate goals established by the controller. It executes items from this agenda until its agenda is complete, or an inconsistency is detected. The Controller analyzes detected inconsistencies and revises the problem solving focus.

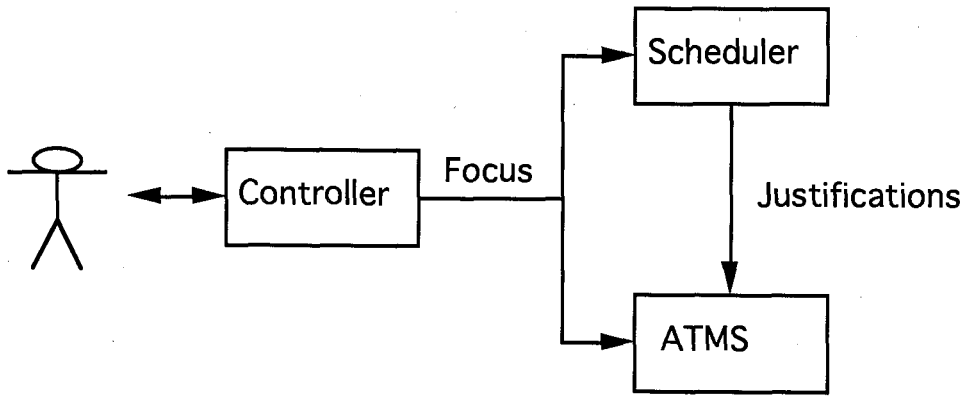


Figure 4.1. Consumer Architecture.

The ATMS maintains the problem solving state in the form of *assumptions*, *datums* and *justifications*. Assumptions represent choice points in a search space. Datums are propositions resulting from the execution of inferencing steps. Justifications relate inferred datums to the antecedents from which they were inferred. The scheduler's agenda items are called *consumers*. The controller focuses problem solving by establishing one or more problem solving *focus environment*.s. A focus environment defines a conjunctive set of active problem solving choices or assumptions. The scheduler schedules for execution those *consumers* which are activated by an established focus environment. Consumers, when executed, record their outcome by establishing new ATMS assumptions, datums and justifications. The ATMS supports queries on the results of problem solving for any one of the established focus environments. The ATMS supports datum sharing between focus environments by maintaining a *label* for each datum, associating it with the assumptions which logically entail it.

We proceed to discuss each Consumer architecture level in detail beginning with the ATMS.

4.1.1. ATMS

An *ATMS node* is a tuple with three parts. Using De Kleer's notation we represent an ATMS node as γ_{datum} : < datum, label, justification set >. γ_x designates the node with datum x; however, the same designation is often used to refer to both the node and its datum.

A *datum* is a proposition asserted by the consumer level during problem solving which the ATMS treats as atomic. Although from the problem solving point of view each datum may be a complex syntactic structure, the ATMS treats each datum as an atomic proposition. It has no access to problem solving semantics.

A *justification* is a reasoning step asserted by the consumer level during problem solving. It is a propositional Horn clause of the form $a_1 \wedge a_2 \wedge ... \wedge a_n \rightarrow c$ where $a_1, a_2, ..., a_n$ and c are ATMS nodes. The nodes $a_1, a_2, ..., a_n$ are referred to as the *antecedent* nodes. The node c is referred to as the *consequent*.

A *justification set* for γ_c is a set of justifications each with the same consequent c. Since by definition a justification set is associated with a node c, and since each justification set member has the same consequent c, each justification in a justification set is represented simply by its set of antecedent nodes. As a result, the justification set can be interpreted as a disjunctive normal form expression which implies the node datum. For example, for the node $<x,label,\{(a_1,a_2),(a_3,a_4)\}>$ we have $(a_1 \wedge a_2) \vee (a_3 \wedge a_4) \rightarrow x$.

In a moment an assumption will be defined as a distinguished kind of ATMS node. For now it suffices to say that an assumption is a node which can be presumed true unless there is evidence to the contrary.

An *environment* is a set of assumptions.

A *label* is a set of environments. The main role of the ATMS is the derivation of node labels. The environments of a node label imply the node datum. More precisely, let J be the current set of ATMS justifications. Then for each environment E of datum x's label we have that J $\Rightarrow$ [1]E $\rightarrow$ x . A label can be interpreted as a disjunctive normal form expression which implies the node datum. For example, for the node <x,{{A_1,A_2},{A_3,A_4}}, justification set> we have J $\Rightarrow$ $(A_1 \wedge A_2) \vee (A_3 \wedge A_4) \rightarrow$ x.

There are four node types corresponding to premises, assumptions, assumed nodes and derived nodes. A *premise* node has a justification with no antecedents. For example < p, {{}},{()}> represents the premise p which is always true.

An *assumption* is a node whose label contains an environment mentioning itself.

An assumption whose label contains a singleton environment mentioning itself is called a *primitive assumption*. The node <A,{{A}},{(A)}> represents the primitive assumption A. A primitive assumption can be presumed true unless there is evidence to the contrary.

A non-primitive assumption is referred to a *dependent assumption*. The node <B,{{A,B}},{(B,...)}> is a dependent assumption. B can be presumed true provided that A is also presumed true, and that there is no evidence to the contrary. In effect B is an assumption whose existence depends on having already made assumption A.

An *assumed node* is neither a premise nor an assumption and has a justification mentioning an assumption. The assumed datum a which holds under assumption A is represented as <a, {{A}},{(A)}> .

All other nodes are *derived nodes*.

---

[1]The symbol $\Rightarrow$ is used here to mean 'logically follows'.

The distinguished node $\gamma_\perp$: < $\perp$, label, justification set > represents falsity. The environments of the label are called *nogoods*. Each nogood E represents an inconsistent conjunction of assumptions since we have J $\Rightarrow$ E $\rightarrow$ false which is equivalent to J $\Rightarrow \neg$E. The label of $\gamma_\perp$ is termed the *nogood set*. Consumers which detect inconsistent results indicate this to the ATMS by specifying justifications with $\gamma_\perp$ as the consequent.

Assumptions are often established as a set of disjunctive choices. For example, a subgoal may be unified with one of several clauses. If we associate each clause choice with a separate assumption we have a disjunctive set of assumptions. Precisely one of the assumptions must be true in any solution.

In order to model disjunctive assumption sets, the ATMS allows negated assumptions to appear directly in justification antecedents. The negation of assumption A is a *non*-assumption node and is referred to as $\neg$A. this enables the disjunction A $\vee$ B $\vee$ C to be expressed as the justification $\neg$A,$\neg$B,$\neg$C $\rightarrow \perp$. We can also express disjunctions for dependent assumptions. If the disjunction B 1 $\vee$ B 2 $\vee$ B3 depends on assumption A1 we assert: A 1,$\neg$ B 1,$\neg$ B 2,$\neg$ B3 $\rightarrow \perp$, which is logically equivalent to A1 $\rightarrow$ B1 $\vee$ B2 $\vee$ B3.

A *oneof disjunction* is a disjunctive set of assumptions for which precisely one assumption can be true. The ATMS represents oneof disjunctions as a disjunction in combination with additional nogood justifications, one for each pair of assumptions declared in the choose assertion. For example, oneof(C_1,C_2,C_3) is represented as $\neg C_1,\neg C_2,\neg C_3 \rightarrow \perp$ and $C_1,C_2 \rightarrow \perp$, $C_1,C_3 \rightarrow \perp$, $C_2,C_3 \rightarrow \perp$. The justifications used to represent a oneof disjunction for a given set of assumptions C, are referred to as the *oneof justifications* for set C.

A *focus environment* is an environment established by the focusing level to focus both ATMS label derivation and consumer level consumer execution.

An *active environment* is an environment which is a subset of a focus environment.

An *active nogood* is a nogood which is a subset of a focus environment.

A maximal consistent set of assumptions is called an *interpretation*. An *extension* is a set of datums which are true under an interpretation.

An *active interpretation* is an interpretation which is also a focus environment. Later we will see that in our use of the ATMS, whenever a new focus environment is first presented for consumer execution, it is also an interpretation. When problem solving is complete, each focus environment which remains is an active interpretation whose extension corresponds to a distinct solution.

We are now in a position to completely characterize node labels. Let J be the set of all ATMS justifications and let $\{E_1,...,E_n\}$ be a label for node n. The ATMS ensures that this label has the following properties[1]:

(1) [Soundness.] $J \Rightarrow E_i \rightarrow n$ for each E_i

(2) [Consistency.] Each active E_i is not nogood. Moreover, the nogood set consists of all active nogoods which follow from J.

(3) [Completeness.] each active interpretation in which n holds is a superset of some E_i

(4) [Minimality.] No E_i is a proper subset of any other.

As a result of the above properties, the ATMS can determine the extension of any active interpretation. It is only necessary to examine each node to determine whether it has an environment which is a subset of the interpretation.

The ATMS supports the following operations:

(1) definition of a new premise

[1]Historically, these properties have been defined for an unfocused ATMS. Here we provide definitions for a focused system.

(2) definition of a new primitive assumption
(3) definition of a new justification
(4) establishment of a new focus environment
(5) removal of a focus environment
(6) general ATMS state queries
(7) consumer installment
(8) consumer removal
(9) establishment of a class consumer

The significance of consumer installment and removal are described in the next section. Of the above operations, most result in straight forward ATMS database changes or queries. The exceptions are operations three and four which trigger *label propagation.*

Consider the case where a new justification is established. If the justification has a consequent which is not yet known to the ATMS, a new node is created for it and the consequent is copied into the datum slot. Referring to the node designated by the consequent, whether it be newly created or an already existing node, as N, the antecedents of the justification are copied into N's justification slot. The addition of the new justification triggers label propagation which maintains correct node labels by propagating incremental label changes.

At the heart of the algorithm is an operation called *reduce.* Reduce takes an input set S of ATMS node labels. Each label in S is associated with a particular ATMS node. Reduce returns a label L. Reduce performs the following:

[REDUCE(S) → L]

R1. If any label in S is the empty set, then return label L = { }.
R2. Check each label for inactive environments. Mark each inactive environment as *blocked* for the ATMS node associated with the environment's label.
R3. Viewing the labels as propositional expressions in disjunctive normal form, compute a new label L by

converting the expression $\bigwedge_i label^*_i$ to disjunctive normal form where $label^*_i$ is the i^{th} label minus its blocked environments.

R4. Remove inconsistent environments, that is, environments which are supersets of nogoods, from L.

R5. Remove subsumed environments, that is, environments which are supersets of other environments in L, from L.

R6. Return L.

Let J be a newly added justification for node N. The ATMS executes the following steps:

(I) [Derive incremental change to N's label.]

I1. Form the set of labels associated with the J antecedent nodes

I2. REDUCE this set of labels to the incremental label L

(A) [Add L to N's label.]

A1. Delete from L every environment which is a superset of an environment in N's label.

A2. Delete from N's label every environment which is a superset of an environment in L.

A3. If L is the empty set then we are done, otherwise add remaining L environments to N's label.

A4. If N is $\gamma_\perp$, each environment E of L is a nogood. Invoke nogood processing for E (step (N) described below). We are done.

A5. If N is not $\gamma_\perp$, then step (P) is executed for each node N' which is a consequent of a justification J' for which N is an antecedent.

(P) [Propagate L through J' to N' from N.]

P1. Let R be the set of J' antecedents minus {N}. Considering L to be associated with N, form the set of labels consisting of L together with the labels associated with R nodes.
P2. Reduce this set of labels to the incremental label L'
P3. Invoke step (A) to add L' to the N' label.

It remains to describe nogood processing. (step A4 above). Let E be a newly discovered nogood. Nogood processing consists of the following:

[NOGOOD]
N1. Sweep E and any superset from every node label.
N2. For every assumption A $\in$ E for which $\neg$A appears in some justification invoke step (2) above to add E - {A} to $\neg$A's label.

An example serves to illustrate the necessity of the second nogood processing step. Suppose we have A $\vee$ B. This is represented as $\neg$A,$\neg$B $\rightarrow \perp$. Suppose we also have the two nogoods {A,C} and {B,C}. Intuitively, it is clear that {C} must also be a nogood since it cannot occur with either A or B and we know that at least one of A or B is true. Nogood processing produces <$\neg$A,{{C}}>, <$\neg$B,{{C}}>, and <$\neg$C,{{A},{B}}> which, when propagated through the justification $\neg$A,$\neg$B $\rightarrow \perp$, results in the required discovery of the new nogood {C}. Nogood processing ensures that *all* active nogoods are derived thereby guaranteeing label consistency.

ATMS implementations typically perform label propagation very efficiently. The results of a simple label propagation is shown in figure 4.2. A is a datum with label {{a1}}; B is a datum with label {{a2,a3}}, and datum C has label {{a3},{a4}}. These labels are propagated through justifications J1 and J2 to produce the label for datum D.

When a new focus environment is established, the ATMS unblocks those blocked environments which are made active by the new focus environment. This is accomplished by unblocking active

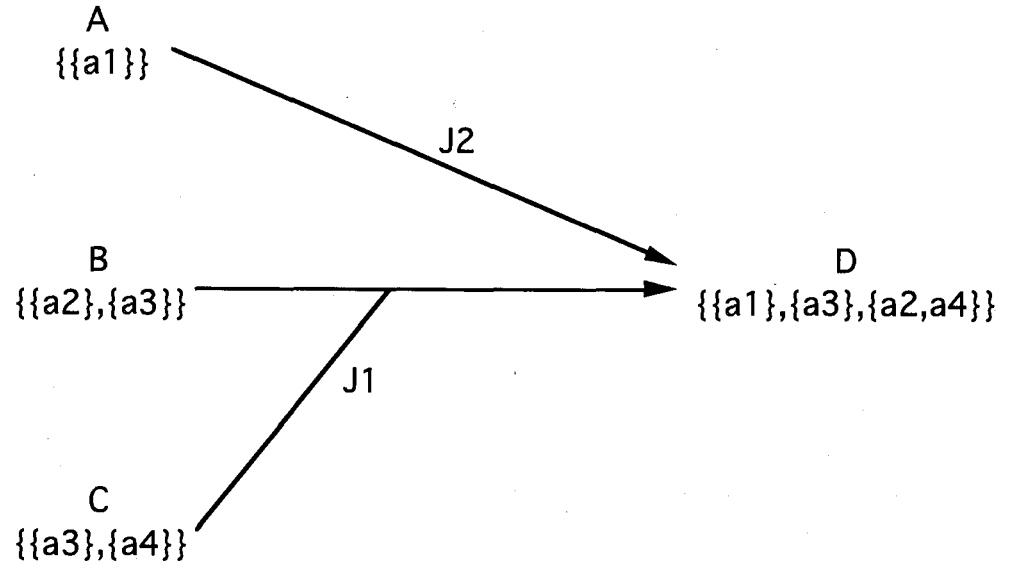


Figure 4.2. Simple label propagation in an ATMS.

environments and executing label propagation. The label propagation described above is only slightly modified to unblock additional environments as label propagation proceeds. In the following italics are used to identify the label propagation differences.

The ATMS executes the following steps.:

(S) Select an ATMS node N which has blocked active environments. Set L = {}

(AU) [Add L to N's label and unblock active environments.]

AU1. Delete from L every environment which is a superset of an environment in N's label.

AU2. Delete from N's label every environment which is a superset of an environment in L.

AU3. Add remaining L environments to N's label.

AU4. If N is $\gamma_{\perp}$, each environment E of L is a nogood. Invoke nogood processing for E (step (N)). We are done.

AU5. Unblock any blocked active environments in N's label and add to L. If L is the empty set then we are done.

U6. The step (PU) is executed for each node N' which is a consequent of a justification J' for which N is an antecedent.

(PU) [Propagate L through J' to N' from N and unblock active environments.]

PU1. Let R be the set of J' antecedents minus {N}. Considering L to be associated with N, form the set of labels consisting of L together with the labels associated with R nodes.

PU2. Reduce this set of labels to the incremental label L'

PU3. Invoke step (AU) to add L' to the N' label *and unblock active environments.*

(RR) Repeat from (S) until there are no more nodes with blocked active environments.

4.1.2. Consumer Level

The consumer level consists of consumers and a consumer scheduler.

A consumer is very similar to an ATMS node with a single justification except that in place of a datum a consumer has a procedure or rule which can be applied to the antecedent nodes in order to derive new problem solving results. In effect, a consumer is an instance of a procedure whose argument bindings serve as connections to antecedent ATMS nodes.

The process of establishing a consumer and its antecedent connections is referred to as *consumer installation.* The ATMS treats the antecedent node, consumer relationship as a conventional justification with the consumer playing the role of the consequent. We refer to a consumer and its antecedents as a *consumer justification.* The ATMS maintains labels for consumers by propagating labels through consumer justifications in the conventional way. An *active consumer* is one whose label contains at least one active environment.

Unlike ATMS nodes, consumers have only a single justification. Consumers cannot be used as antecedents in other justifications.

Each consumer, when executed, establishes its results by communicating new justifications to the ATMS. Consumers may establish new justifications for any type of datum including ⊥. Each new justification has precisely the same set of antecedents as the consumer itself. An exception is the case where a consumer installs a new assumption. In this case the new justification includes the new assumption node itself, as an additional antecedent. Consumers only execute once. They remove themselves immediately after they are executed. There is no need for a consumer to execute more than once as their results will have already been captured by the ATMS. Consumers may also install new consumers.

It is also possible to define generic consumers whose antecedents are classes of nodes rather than specific nodes. A *class* is an ATMS construct used to represent a set of ATMS nodes of a particular type. For example we may represent all nodes with datums of the form X|binding as the X-variable class. With class consumers a class is specified for each consumer antecedent. A *class consumer* is a procedure which can be applied to any combination of antecedent nodes provided that each antecedent node is of the designated class. Whereas a consumer is a procedure instance, a class consumer is a procedure. After a class consumer is registered with the ATMS, the ATMS ensures that a class consumer instance is attached to each existing or new combination of member nodes of the specified antecedent classes. Whenever a new member of a class is added to the ATMS database, the ATMS automatically checks whether new class consumer instances need to be installed. Once installed, there is no distinguishing between class consumer instances and conventional consumers. Class consumer instances are, in fact, consumers.

The consumer scheduler is activated whenever a new focus environment is established. The scheduler responds by executing active consumers. Consumer execution continues until either all active consumers have been successfully executed, or until a

consumer's execution uncovers an inconsistency (establishes a justification for $\perp$).

We now examine the scheduler's strategy regarding which active consumer to execute next. If $F_1,..,F_n$ are the current focus environments and $P(F_i)$ is the powerset of F_i then $S = P(F_1) \cup P(F_2) \cup ...,\cup P(F_n)$ is the set of all active environments. Each $E \in S$ occurs in zero or more consumer labels. We say that E *activates* the consumers whose labels it occurs in. The scheduler selects consumers for execution indirectly. It selects an element of S and then executes the consumers which are activated by the selected environment. The scheduler selects elements of S in order of increasing environment size. Starting with the empty set, ever larger environments are considered, ending with the focus environments themselves. If we have the single focus environment, {A1,A2,A3}, the scheduler will schedule the following sequence of environments: {},{A1},{A2},{A3},{A1,A2},{A1.A3},{A2,A3},{A1,A2,A3}.

The above scheduling strategy is based on finding the most general (minimal) node labels as early as possible. In doing so we avoid superfluous label updating, and greater reuse of already derived results whenever a new focus environment is established. In the case of $\gamma_\perp$, more general labels result in more effective search space pruning.

4.1.3. Focusing Level

The focusing level consists of a controller whose basic role is to provide top level control over the problem solving. The set of assumptions encountered during problem solving form a search space of possible solutions. The controller moves the problem solving from point to point in this search space by establishing different focus environments. The controller must ensure that each search space point which is not ruled out by a nogood is considered a solution candidate, but that no point is visited more than once. We

can think of the controller as having an adaptive strategy for ranking search space points and for ensuring that both the ATMS and consumer levels are focused on the most highly ranked possibility.

Before describing the controller's algorithm we introduce some definitions.

The *current focus environment* is the most recently established focus environment.

Recall that an interpretation is a maximal consistent set of assumptions. In computing solutions to problems, we extend focus environments until they are interpretations and there are no remaining active consumers. A focus environment satisfying these conditions is called a *solution environment.* Extensions to solution environments correspond to *solutions.*

The aim of the controller is to extend the current focus environment into a solution environment. The controller must subject each new focus environment to consumer execution in order to determine whether or not the current focus environment must be modified again, or whether it is already a solution environment. When a solution environment is found, and if additional solutions are desired, the controller begins work on another focus environment, leaving the previous one intact. Over time a number of solution environments are generated.

The controller algorithm is described in terms of the following operations.

(1) SETUP. The initial problem state and consumers are established with the ATMS.

(2) FIND_NEXT_fe. A new current focus environment is established.

(3) FIND_NEXT_SOLUTION. A new solution environment is established.

(4) FIND_SOLUTIONS. The desired number of solution environments are established.

The current focus environment is denoted by *current*. We write *current = start* to refer to the startup condition where no focus environments have been established yet. We write *current = end* to refer to the terminating condition where no further solution environments are to be found.

The controller executes the following:

(C) C1. SETUP
C2. *current* = start.
C3. FIND_SOLUTIONS

[FIND_SOLUTIONS]
S1. FIND_NEXT_SOLUTION
S2. if *current = end* return
S3. if desired number of solutions already found then return
S4. repeat from step S1.

[FIND_NEXT_SOLUTION]
N1. FIND_NEXT_fe
N2. if *current = end* then return
N3. invoke scheduler
N5. if *current* is a solution environment then return.
N6. repeat from step N1.

For the moment we leave undefined the operations FIND_NEXT_fe and SETUP, as well as the criteria for deciding what constitutes the desired number of solutions. These items depend on the particular problem that is being solved. Consider them as slots to be filled in converting the consumer architecture schema to an actual problem solving architecture.

4.2. Definite Programs

The consumer architecture described above is a framework for a design. If we are to apply it to a particular problem it is necessary to fill in the generic 'slots' of the framework with specific design

decisions. At the ATMS level the types of datums and classes must be specified. At the consumer level the types of consumers and consumer classes must be defined. At the focusing level we need to define the strategy for finding the next focusing environment. In this section we apply the consumer architecture to the problem of computing answers to definite program goals.

Variables which appear in one or more goals of an SLD-refutation are referred to as *active variables*. For example, consider the query ?-p(X,g(Y)), and the clause p(g(W),z) :- a(W),b(Z). If we resolve our query with this clause we obtain the new goal a(W),b(g(Y)) and binding X|g(W). In this example variables X,Y and W are active variables. Variables X and Y appear in the first goal. Variable W appears in the second. Variable Z is inactive and does not appear in any goal. The binding X|g(W) is termed an *active variable binding*. It is only necessary to keep track of bindings for active variables. We do not need to store the binding for variable Z.

We make use of three kinds of datums, subgoal datums, variable datums and assumptions. A *subgoal datum* corresponds to a subgoal which must be proven true in order to complete a refutation. Subgoal datums are only defined for those refutation subgoals which have multiple clause choices. The set of *clause choices* for a subgoal with predicate p, and a definite program P is the subset of P which defines p. We refer to subgoals with multiple clause choices as *non-deterministic subgoals*. Subgoals with a single clause choice are referred to as *deterministic subgoals*. In the above example p(X,g(Y)), a(W) and b(g(Y)) are possible subgoal datums, depending on whether or not they are non-deterministic.

Variable datums are used to record variable bindings for active variables. In the above example, X|g(W) is established as a variable datum. The ATMS recognizes the set of all variable datum nodes for a particular active variable as a distinct *variable class*.

Assumptions are used to represent non-deterministic subgoal clause choices. Each clause choice is established as a distinct assumption.

We often refer to nodes and justifications by the type of their datum or consequent. For example, we refer to *subgoal nodes*, *variable nodes* and *assumption nodes*. We also refer to *subgoal justifications*, *variable justifications* and *assumption justifications*. A justification for $\perp$ is referred to as a *nogood justification*.

At the consumer level we make use of resolution consumers and unification class consumers.

Each *resolution consumer* is attached to a non-deterministic subgoal S and an assumption corresponding to one of the clause choices of S. When a resolution consumer is executed it performs the following sequence of steps. Let S be the consumer's antecedent subgoal, and C be the program clause associated with its antecedent assumption.

[Make a non-deterministic choice C for subgoal S]

(1) Interpreting G as the goal :- S, derive G' from G and C using mgu θ.

(2) If unification fails establish a nogood justification. Remove consumer and return

[Resolve deterministic subgoals of G'. Establish variable bindings in final θ]

(1) Select a deterministic subgoal D from G'.

(2) Derive new G' from G' and clause associated with D using mgu θ'.

(3) If unification fails establish a nogood justification. Remove consumer and return

(4) Derive new θ from θ composition θ'.

(5) Repeat from step (1) until resulting G' contains only non-deterministic subgoals.

(6) Establish a variable justification for each active variable binding in final θ.

[Establish new active variables for final G']

(1) Each variable appearing in final G' which has not yet been established as an active variable is a new active variable. Establish each such variable as an active variable by establishing a unification class consumer for the variable.

[Establish new non-deterministic subgoals for final G']

(1) Remove a non-deterministic subgoal N from G'.
(2) Establish a subgoal justification for N.
(3) Establish an assumption justification for each of N's clause choices.
(4) Establish oneof justifications for the complete set of these choice assumptions for N.
(5) Attach a new resolution consumer to each pairing of N with one of its clause choice assumptions.
(6) Repeat from step (1) until there are no more members of G'.

[We are done]

(1) Remove consumer
(2) Return

Recall that all justifications installed by a consumer have the same antecedents as the consumer itself except that assumption justifications also have themselves as an antecedent. Resolution consumers deviate from this practise somewhat in the case of assumption justifications. Resolution consumers install subgoal justifications, and then attach assumptions to subgoals. Each assumption justification has exactly two antecedents, itself and the subgoal for which it is a clause choice to. As a result, each subgoal node is the antecedent of its own clause choices. We can think of these assumption nodes as belonging to their antecedent subgoal node. We refer to these assumption nodes as *subgoal assumptions*. Logically, we can think of the resolution consumer as attaching a choice consumer to each new subgoal which immediately installs the

subgoal's assumptions. The situation is illustrated in figure 4.3. Justifications for variable nodes and subgoal nodes are established with the resolution consumer's antecedents in the conventional way.

A *unification class consumer* is defined for each active variable. A unification class consumer for active variable X has two antecedents. The class of variable X is the designated class of both antecedents. By establishing a unification class consumer for each active variable, we automatically cause a unification class consumer instance, referred to as a *unification consumer*, to be attached to each pair of distinct X variable nodes. When a unification consumer is executed, it performs the following:

[Unify antecedent variable bindings]
(1) Unify the first antecedent variable binding with the second.
(2) If unification fails establish a nogood justification. Remove consumer and return
(3) Otherwise, establish variable justifications to record the results of the unification.
(4) Remove consumer (instance) and return

As an example consider the definite program goal, ?- A(X,f(Y)),B(X). Let both subgoals be non-deterministic subgoals, each with two defining clauses. Let the first clause choices for subgoals A and B be:

A(g(V),W) :- C(V),D(V,W).
B(f(z)) :- E(Z).

Figure 4.3 illustrates some of the justifications installed by consumers during problem solving. In the figure, subgoal, variable, and assumption datums are represented by rectangles, circles and diamonds respectively. In this case problem solving has uncovered the nogood {A1,B1}. For clarity, figure 4.3 does not show the oneof justifications installed for each subgoal's clause choices. For example, the following justifications are installed for C1,C2.

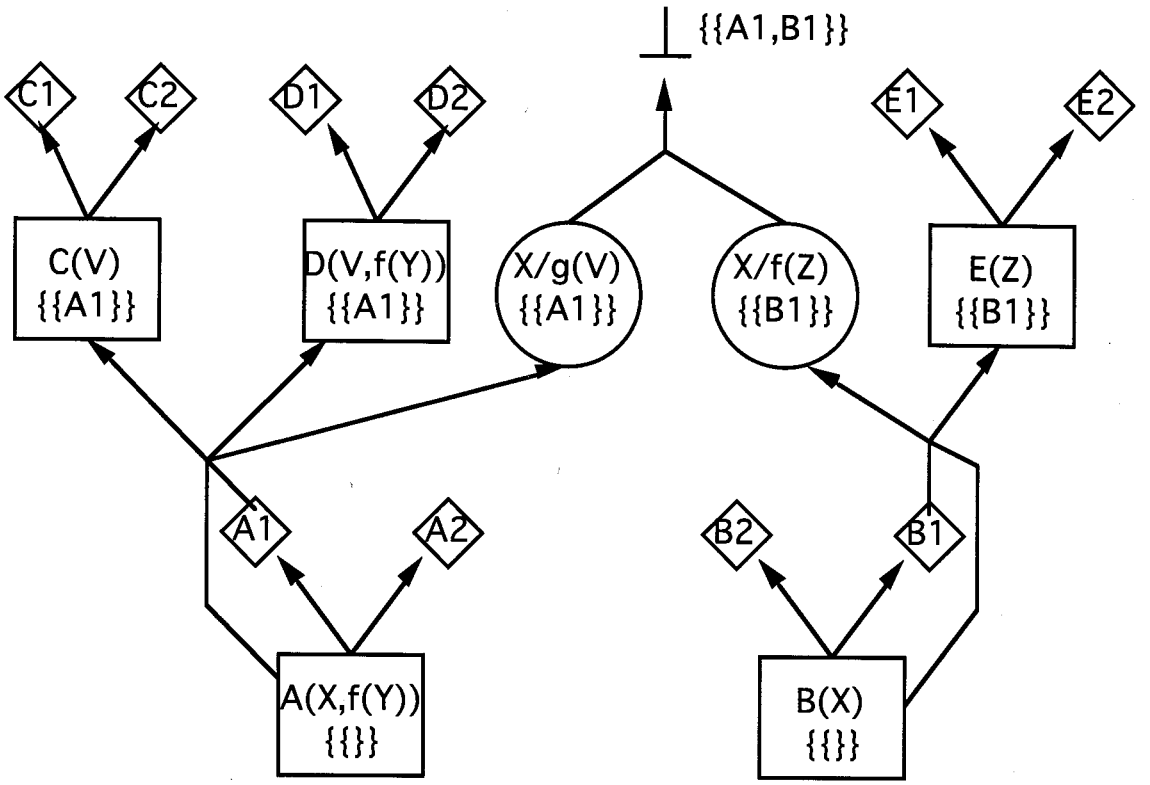


Figure 4.3 Justification lattice for a definite program refutation

$C(V),\neg C1,\neg C2 \rightarrow \perp.$
$C1,C2 \rightarrow \perp.$

These justifications establish C1,C2 as a conditional oneof disjunction. As long as C(V) holds, then precisely oneof its clause choices must be true. Since the label of C(V) is {{A1}}, the occurrence of the C1,C2 oneof disjunction depends on having first made assumption A1.

At the focusing level we need to provide procedures for SETUP and FIND_NEXT_fe, and we need to describe criteria for deciding when enough solutions have been found.

The initial problem state is set up by establishing a set of premises which describe the goal to which we require an answer. In effect our basic premise is that the goal does have a computed

answer. Discovering that this premise is not true amounts to finding that the empty set is a nogood. If this ever occurs, problem solving will halt as required. Setting up the initial problem state involves many of the same steps as resolution consumer execution. In the following, previously described resolution consumer steps are surrounded by curly brackets.

[SETUP]

[Set up all variables appearing in goal as active variables.]
(1) Establish a unification class consumer for each variable appearing in the goal.

[Set up to execute resolution consumer steps]
(1) G' = goal
(2) θ = identity substitution
(3) treat consumer antecedents as the empty set. As a result, variable and subgoal justifications established by the following resolution consumer steps create premises. Assumption justifications create primitive assumptions.

{ Resolve deterministic subgoals of G'. Establish variable bindings in final θ.}

{ Establish new active variables for final G'. }

{ Establish new non-deterministic subgoals for final G'}

We turn now to describing FIND_NEXT_fe. In moving through the search space, the basic strategy of chronological backtracking is adopted. Whenever an inconsistency is encountered, we simply change the most recent choice and try again. Whenever we find that no choices from a oneof set lead to a solution, we back up to the most recent oneof set with untried choices.

This approach has been generalized and made more effective by RMS-based backtracking algorithms [Bruynooghe,84], [Cox,84], [Drakos,88], [Havens,91], [You,89]. In these algorithms, whenever a

nogood is encountered the most recent choice which also contributed to the nogood is changed. The situation where all choices of a oneof set are exhausted results in a new nogood containing those earlier choices which lead to this situation. We adopt the same approach and adapt it to the ATMS-based consumer architecture.

Each subgoal node is associated with the chronological time of its creation by the ATMS. This forms a total ordering of all subgoal nodes. Since each subgoal has an associated oneof assumption set, then we can consider chronological time as providing a total ordering of subgoal assumption sets. This total ordering is, of course, consistent with the partial ordering defined by logical subgoal dependencies. Since focus environments do not contain more than one assumption from the same subgoal assumption set, then all assumptions in any focus environment are totally ordered by the times associated with their assumption sets.

When a single nogood makes the current focus environment inconsistent, we replace the most recent assumption in the nogood with an untried assumption from its assumption set. The case where all assumptions from the assumption set have been tried, and all introduce nogoods, leads to a new nogood containing earlier assumptions which lead to this situation. In this case we change assumptions by considering the nogoods one at a time in a particular order. We consider the nogood consisting of assumptions which are all earlier than the most recent assumption of any other nogood first. This is analogous to backing up to the most recent assumption set for which there remain untried assumptions.

Under this backtracking strategy, the ranking of solution environments is arbitrary and does not draw upon problem domain semantics. The controller does however ensure completeness.

We now examine the mechanism by which new nogoods arise when all assumptions from an assumption set are exhausted. Consider what happens when nogood A_1,B_1,C_1 is uncovered. Here we assume that A_1,B_1,C_1 correspond to assumptions for subgoals A,B,C

respectively. We also assume that the ordering of these subgoals is $A < B < C$. The ATMS' NOGOOD procedure records $\{B_1,C_1\}$ as an environment in $\neg A_1$'s label. Similarly, $\{A_1,C_1\}$ is recorded for $\neg B_1$; $\{A_1,B_1\}$ is recorded for $\neg C_1$. The controller then changes the most recent assumption of A_1,B_1,C_1. Suppose it tries the combination A_1,B_1,C_2 next. Of the three new environments established for nodes $\neg A_1,\neg B_1,\neg C_1$, only the $\{A_1,B_1\}$ environment for $\neg C_1$ remains active. We refer to this environment as E_1 as it serves to justify the retraction of C1. Note that E_1 is made up of the other assumptions in the initial nogood, all of which are earlier than C. In exhausting each of the C choices, an active environment E_i is created for each C_i. The ATMS automatically propagates these environments through the oneof justification $\neg C1,\neg C2,... \rightarrow \perp$. The result is a new nogood made up of the union of the E_i's. This new nogood consists entirely of assumptions which are earlier than C.

Before describing the algorithm for FIND_NEXT_fe in detail it is necessary to introduce some definitions.

A *left out subgoal* is a subgoal node which:
(1) has a label containing an environment which is a subset of the current focus environment.
(2) does not have any of its subgoal assumptions included in the current focus environment.

A *solution assumption* is introduced after a new solution environment is found. It is the assumption that there exists another solution which is distinct from any solution found thus far. In most prolog implementations additional solutions are found by treating the current solution as a nogood. Solution assumptions enable problem solving to continue without invalidating the solutions found thus far. There is at most one solution assumption per focus environment. By definition it is considered the oldest assumption in the focus environment.

Strictly speaking, we do not consider solution assumptions to be a part of solution environments, however, when the distinction is not important we sometimes refer to focus environments formed from a solution environment $\cup$ {S} where S is a solution assumption, as a solution environment.

The algorithm for FIND_NEXT_fe is described in terms of the following operations.

(1) REPLACE(fe) . The current focus environment is replaced with the new focus environment fe, which becomes the new current focus environment. This operation causes label propagation which may result in new active nogoods.

(2) NEW(fe). A new focus environment, fe, is established. It becomes the new current focus environment. This operation causes label propagation which may result in new active nogoods.

(3) EXTEND. This operation is invoked when the current focus environment is consistent but not maximal. The current focus environment is replaced with one which is bigger by one assumption.

(4) BACKTRACK. This operation executes a single backtrack step. The current focus environment is replaced.

(5) NEXT. This operation is invoked when the current focus environment is a solution environment. A new solution assumption is established. The new solution assumption is used to invalidate the current focus environment.

We follow the conventions established earlier. The current focus environment is denoted by *current*. We write *current* = *start* to refer to the startup condition where no focus environments have been established yet. We write *current* = *end* to refer to the terminating condition where no further solution environments are to be found. We detect this by testing for a nogood consisting of either $\emptyset$ or {S} where S is a solution assumption.

[NEW(fe)]

(1) establish fe as a new focus environment.

[REPLACE(fe)]

(1) remove *current* as a focus environment.

(2) establish fe as a new focus environment.

[EXTEND]

(1) fe = *current*

(2) select a left out subgoal.

(3) add one of selected subgoal's assumptions to fe while ensuring that fe remains consistent. Note that we can always accomplish this as long as *current* is consistent to begin with.

(4) REPLACE(fe)

[BACKTRACK]

(1) fetch the set S of nogoods which make *current* inconsistent

(2) select from S the nogood N with the earliest most recent assumption.

(3) fe = *current*.

(4) let A be the most recent assumption in N. Replace A in fe with an assumption from A's oneof set which is consistent with the earlier assumptions in fe. Note that we can always accomplish this as long as N is the nogood with the earliest most recent assumption.

(5) remove any assumption from fe which depends on A

(6) REPLACE(fe).

[NEXT]

(1) Form the set $A = \{A_0, A_1, \ldots, A_n\}$ of solution environments discovered so far. In this case we strip off solution assumptions from each A_i. *Current* corresponds to the final solution environment A_n.

(2) Form the set $S = \{S_1, \ldots, S_n\}$ of solution assumptions defined so far. Note that A_0 does not have an associated solution assumption.

(3) Define the new solution assumption S_{n+1}

(4) install S_{n+1}, $A_i \rightarrow \perp$ for each $0 \leq i \leq n$

(5) install S_{n+1}, $S_i \rightarrow \perp$ for each $1 \leq i \leq n$

(6) NEW($\{S_{n+1}\} \cup A_n$).

```
[ FIND_NEXT_fe ]
   CASE
   { current = start }
        NEW(∅)
   { current = end }
        return
   { current = a solution environment }
        NEXT
   ENDCASE
   UNTIL current is consistent and no more left out subgoals
        CASE
        { current is consistent }
             EXTEND
        { no more solutions }
             REPLACE(end)
             Return
        { ¬end and current is not consistent }
             BACKTRACK
        ENDCASE
   ENDUNTIL
   Return
```

It remains to define the controller's criteria for deciding when enough solutions have been obtained. Since, in this case, solution environment ranking is arbitrary, all solution environments are equally good from a domain semantics point of view. As a result, the most useful mechanism is probably an interactive one. The interactive user can simply indicate when enough solutions have been obtained.

Having introduced an interactive user, it is useful to exploit the multiple solution architecture by allowing the interactive user to

incrementally add to the goal that we are computing solutions to. For instance, the user may wish to add new conjunctive subgoals, or to provide bindings to goal variables. As the case of new conjunctive subgoals requires slightly more elaborate control we illustrate the principle by simply allowing the user to specify goal variable bindings. The following is a updated version of the top level control exercised by the controller. We also repeat the earlier FIND_SOLUTIONS procedure for easy reference.

[TOP]
C1. SETUP
C2. *current* = start.
C3. FIND_SOLUTIONS
C4. If no solutions found then finished
C5. If no further goal refinements then finished
C6. REFINE
C7. Repeat from C3.

[FIND_SOLUTIONS]
S1. FIND_NEXT_SOLUTION
S2. if *current* = *end* return
S3. if desired number of solutions already found then return
S4. repeat from step S1.

Step S3 is an interactive step. FIND_SOLUTIONS will generate as many solutions as requested by the interactive user at which point control returns to TOP. Step C5 enables the user to incrementally refine the original goal. The REFINE operation of step C6 communicates the changed goal to the ATMS. As a result some previously computed solution environments may no longer correspond to goal solutions. These solution environments are dropped from the list of focus environments maintained by the controller. Finally, normal problem solving resumes with FIND_SOLUTIONS. As usual the current focus environment will be modified in an attempt to generate the next solution environment.

The REFINE procedure is as follows.

[REFINE]

(1) Establish new goal bindings as premises. This may result in label propagation. It will also result in the creation of new unification consumers.

(2) Invoke scheduler to execute the new consumers.

(3) Remove any focus environments that have become inconsistent as a result.

In principle it is not difficult to provide for the addition of new conjunctive subgoals as well as bindings to goal variables. The controller invokes the SETUP procedure on the new subgoals. If this results in new non-deterministic subgoals then the controller marks all previously discovered solution environments which remain consistent as *partial solution environments.* The very first partial solution environment becomes the new current focus environment. Now when the controller returns to finding solutions it attempts to extend the first partial solution environment into a new solution environment. Whenever one partial solution environment becomes inconsistent the controller skips ahead to the next partial solution environment. Eventually the controller is left with only complete solution environments once again.

This goal revision mechanism enables the interactive user to compare multiple solutions, and to incrementally apply filtering conditions to reduce them in number. This is a process which occurs naturally in abductive reasoning problems.

4.3. Comparison with Intelligent Backtracking

In this section we briefly compare the backtracking approach described above, which we refer to as ATMS-based backtracking, with various intelligent backtracking schemes [Bruynooghe,84], [Cox,84], [Drakos,88], [Havens,91], [You,89] for Prolog. For the most part we make this comparison at an abstract design level rather than at an implementation level.

Perhaps the most significant difference is that whereas intelligent backtracking schemes are based on a single problem solving context, ATMS-based backtracking supports multiple contexts. As discussed above multiple context problem solving supports the generation and comparison of multiple solutions. Moreover, a goal can be interactively refined and the changes applied to all previously discovered solutions. While single context systems provide some mechanisms for generating multiple solutions, they do not maintain dependency information for any but the latest partial solution. Nor do they support mechanisms for maintaining an ongoing relationship between several solutions and a changing goal. The consumer architecture described above is explicitly based on supporting and maintaining this relationship.

Another difference between the two approaches is that whereas intelligent backtracking schemes discard datums and nogoods which are of limited use, ATMS-based backtracking, as described above, never discards any information that is registered with the ATMS. The rationale for discarding information is based on a practical tradeoff between search space reduction and datum reuse on the one hand, and the space and storage overheads which perfect search space reduction and datum reuse require. Many of these tradeoffs are tied to the efficient stack-based memory allocation scheme used in most Prolog implementations.

We point out that this is not a fundamental difference between the two approaches. It is not difficult to modify ATMS-based backtracking to discard essentially the same information as the intelligent backtracking schemes. For example, one simple mechanism is for the ATMS to purge all inactive environments from labels. Any node whose label becomes the empty set as a result, is discarded. As nodes disappear so do any consumers which are attached to them. Resolution consumers are retained as long as they are attached to an active subgoal, and are reexecuted each time they are activated. The environment purging mechanism deletes nogoods from the nogood set while preserving a record of those nogoods that

are needed in order to prevent the same choice being tried more than once. For instance, if choice B1 in combination with an earlier choice A1 is found to be nogood then our conventional ATMS-based backtracking system creates the justification A1 $\rightarrow \neg$B1. This results in the active environment {A1} for node $\neg$B1 even after we have modified the B choice to B2. This active label for $\neg$B1 acts to record the fact that B1 cannot be retried as long as A1 remains in the focus environment. This mechanism is analogous to that found in intelligent backtracking schemes. Like intelligent backtracking, this modified approach ensures no search space point is visited more than once while skipping over portions of the search space. On the other hand some nogoods which are discarded may end up being discovered more than once.

It would seem that the above modified ATMS scheme makes the same tradeoffs as intelligent backtracking while retaining the advantages of multiple context problem solving. This is not quite the case, however, since intelligent backtracking implementations often tie their tradeoffs explicitly to their stack-based architecture.The stack-based architecture is not suited to multiple context problem solving, nor is it suited to probabilistic best-first type search of the kind discussed in the next section. We argue that abductive problem solving is especially suitable to probabilistic reasoning and multiple context problem solving. Under these circumstances, the basic practical tradeoffs of RMS-based search techniques for Prolog need to be reexamined.

A more fundamental difference between the two approaches concerns the way in which ATMS-based backtracking schedules consumers. This is best illustrated by an example. Consider the goal -? p1(X),p2(X),p3(X). Let these three subgoals be non-deterministic subgoals. We designate the first clause choice for subgoal p1(X) as A1, the first clause choice for p2(X) as B1, and the first clause choice for p3(X) as C1. The clauses associated with these choices are:

A1 p1(f(Y))
B1 p2(f(a))
C1 p3(f(b))

Given a focus environment of A1,B1,C1 the consumer scheduler schedules active consumer execution according to the environment sequence {A1},{B1},{C1},{A1,B1},{A1,C1},{B1,C1},{A1,B1,C1}. The resulting justifications are shown in figure 4.4 (a). The nogood {B1,C1} is discovered after six reasoning steps. In contrast, intelligent backtracking schemes execute incremental reasoning steps. They begin by deriving new results under assumption A1. Next assumption B1 is added to assumption A1 and new results obtained under these combined assumptions. Finally assumption C1 is added to A1 and B1 to form the environment sequence {A1},{A1,B1},{A1,B1,C1}. Each new assumption triggers reasoning steps which make use of the results obtained under previous assumptions. The result is shown in figure 4.4 (b). The nogood {A1,B1,C1} is discovered after three reasoning steps.

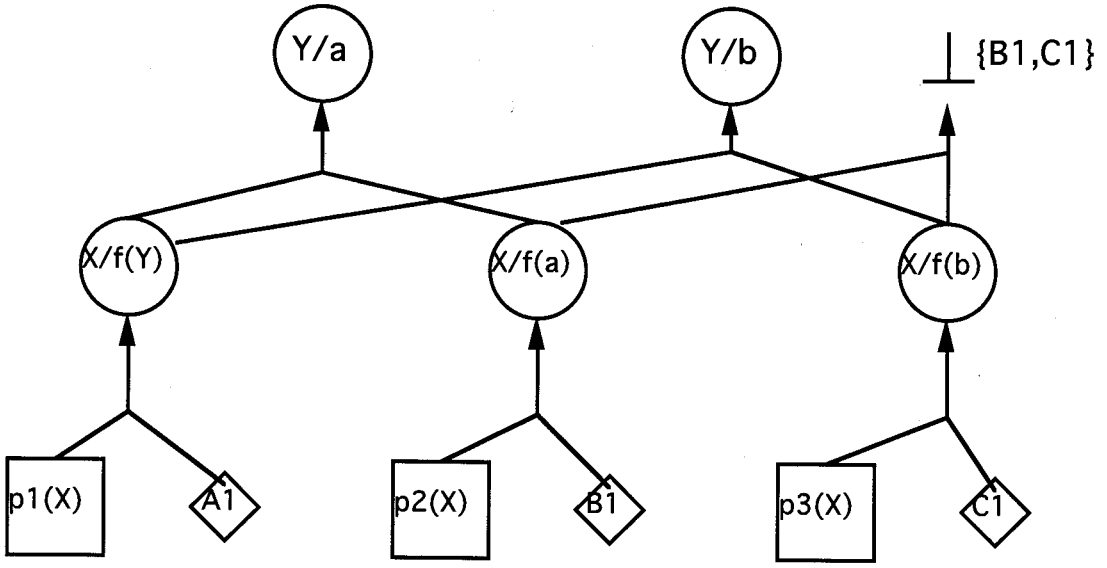
In comparing the results of the two approaches we see that ATMS-based backtracking finds the most general form of the nogood, but executes more reasoning steps to find it. Here we have a basic tradeoff. More general nogoods lead directly to more efficient search space reduction. Hence we save on future reasoning steps but pay in current reasoning steps.

We can characterize the difference between the labels derived by the two approaches. The labels computed by intelligent backtracking reflect the order in which subgoal choices are made during problem solving. If the choices are made in a different order, the label environments may be in a more general form. In the example, if we choose B1 first, followed by C1, we discover the more general nogood {B1,C1}. In contrast, ATMS backtracking computes labels which are independent of the order in which subgoal choices are made. This follows from the fact that every possible way of incrementally building the combination A1,B1,C1 is represented in the scheduler's environment sequence. The environment sequence includes

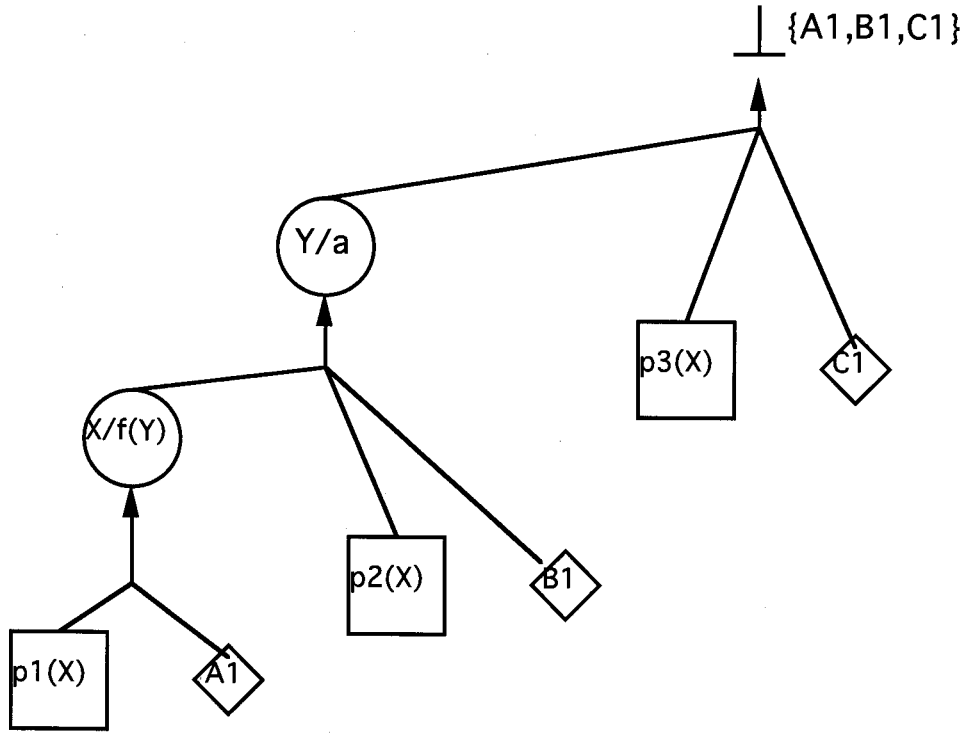
{A1},{A1,B1},{A1,B1,C1} as a sub-sequence. It also includes sub-sequence {C1},{B1,C1},{A1,B1,C1}. The result is labels which are in their logically most general form and do not reflect arbitrary subgoal choice orderings.

The tradeoff represented by order independent labels requires further study. At this point we simply point out the tradeoff and reflect on its relationship to a stack-based architecture. For example, a characteristic of intelligent backtracking's incremental approach is that variables become incrementally more specific as problem solving proceeds. Said differently, variable bindings form a total ordering of increasing specificity. In the example first we have X|f(Y) then Y|a. From the point of view of X it is bound to f(Y) first, then f(a). Compare this with ATMS-based backtracking where we have partially ordered bindings for X: X|f(Y), X|f(a) and X|f(b). The incremental nature of intelligent backtracking is well suited to the stack-based architecture which incrementally extends and rolls back the problem solving state. Again, for abductive problem solving the tradeoff should be examined in a different light.

The incremental single context nature of intelligent backtracking results in labels which consist of single environments only. With ATMS-based backtracking even if we maintain only a single context, we may have labels consisting of several disjunctive environments. This occurs for example in the case where choice A1 leads to the same X binding as choice B1 taken by itself. Intelligent backtracking systems actually maintain a datum environment rather than a datum label.



(a) Order-independent Reasoning



(b) Order-dependent Reasoning

Figure 4.4. ATMS-based backtracking derives order-independent labels.

Finally, we point out that the order-independent approach of ATMS-based backtracking makes it easier to reuse datums when focus environments are changed. In the example, consider the case where we wish to change the initial assumption A1. In intelligent backtracking this is not easy as the resolution step for p2(X), resulting in the justification for Yla, is based on the results of A1. For ATMS-based backtracking, the results of the resolution step for p2(X) are available independent of assumption A1. Hence this earlier result is readily reused. Again, clause reuse is not a consideration for conventional intelligent backtracking systems, which rely on the stack mechanism to roll back problem solving to the point where the choice being changed was made, in the process discarding results that could, in principle, be reused.

4.4. Bayesian Programs

In this section we examine how the consumer architecture may be applied to the problem of computing answers to Bayesian program goals. Our approach is to extend the definite program design of the previous section.

Consider the Bayesian network shown in figure 4.5. The Bayesian program BP for this network contains the following node predicate definitions for non-terminal nodes.

```
node_D(_D←[_B←[_A],_C←[_A]]):-
node_B(_B←[_A]),node_C(_C←[_A]),family_D(_D,_B._C).

node_B(_B←[_A←_X1]):- node_A(_A←_X1),family_B(_B,_A).

node_C(_C←[_A←_X1]):- node_A(_A←_X1),family_C(_C,_A).
```

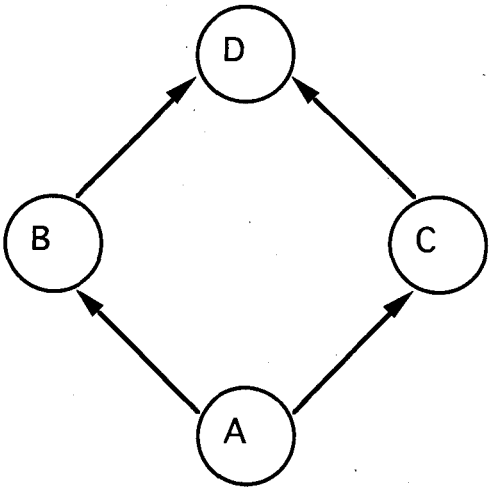


Figure 4.5 A Simple Bayesian Network

We refer to the family predicates of a Bayesian program, together with node predicates for terminal nodes as *characteristic predicates*. Subgoals for these predicates are referred to as *characteristic subgoals*. In the above example the characteristic predicates are family_B, family_C, family_D, and node_A. We refer to predicates with annotated clauses in their definitions as *annotated predicates*. Subgoals for these predicates are referred to as *annotated subgoals*. Annotated predicates are always characteristic predicates.

Consider the Bayesian goal G, ?- node_D(_X). In the preceding section for definite programs, we defined a SETUP procedure which executes as many deterministic SLD derivation steps as possible, leaving only non-deterministic subgoals. If we execute the SETUP procedure for G we end up with the following set of non-deterministic subgoals, and binding for _X.

node_A(_A),family_B(_B,_A),family_C(_C,_A),
family_D(_D,_B,_C)

_X | _D←[_B←[_A],_C←[_A]]

We are left with a characteristic subgoal for each node in the original Bayesian Network. It is not difficult to establish that this is true in general. For the Bayesian goal G, based on subgoals for nodes $N_1,N_2,...,N_n$, the SETUP step results in a single characteristic subgoal for each node in the set $prev(N_1) \cup prev(N_2) \cup ...\cup prev(N_n)$. Typically, a subset of these characteristic subgoals are annotated subgoals.

Note that the same characteristic subgoal may be derived by SETUP more than once. In the above example node_A(_A) will be derived twice, once as node_B is resolved, and again as node_C is resolved. Our use of shared variables to represent most recent ancestors in Bayesian goals ensures that each occurrence of the node_A subgoal has identical arguments. Since we are only interested in the final *set* of non-deterministic subgoals, the redundant subgoal is removed from further consideration. This is implicit in the operation of the ATMS which checks whether the consequent of each new justification is a new datum or not. In this case, since SETUP establishes premises, or justifications without antecedents, the ATMS simply ignores all but the first attempt to establish a premise. This simple mechanism acts to ensure that the cost (defined below) of an assumption is not counted more than once.

Consider the case where a proper subset of the characteristic goals established by SETUP consists of annotated subgoals. As we have shown previously, each clause choice for an annotated subgoal contributes a probability factor to the probability associated with the final computed answer. The probability of the final computed answer consists of the product of these clause probabilities. If our objective is to find the most likely answer first, then we wish to find the answer which maximizes the final probability. This suggests an approach based on a best-first-search. We associate a *cost* with each annotated clause selection, cost = $\log (1/P_c) = -\log P_c$ where P_c is the probability associated with clause c. This definition has the following desirable properties. Cost equals zero for $P_c=1$. Cost increases monotonically to infinity as P_c is changed monotonically towards

$P_c = 0$. We have $Cost_1 + Cost_2 = -(\log P_1 + \log P_2)$ which equals $-\log P_1 P_2$. Thus we can add individual clause costs together to obtain the combined cost of having made a particular set of clause choices. We can now equivalently redefine our objective in terms of costs rather than probabilities. Our objective is to find the answer which minimizes the final cost.

Note that not all characteristic subgoals are necessarily annotated subgoals. The clause choices of non-annotated characteristic subgoals have an effective cost of zero. The appearance of both zero cost subgoals and non-zero cost subgoals suggests a searching strategy which combines both backtracking and best-first-search techniques.

As described earlier, we may provide intensional definitions for characteristic predicates. For example, We may have:

```
{family_D(D/_d,B/_b,C/c1) :- p1(_d,_b), .8 }
{family_D(D/_d,B/_b,C/c2) :- p2(_d,_b), .2 }
```

During problem solving, the first choice may be taken. This choice comes at the cost of -log(.8) and leaves us with the unannotated subgoal p1(_d,_b). Note that the p1 subgoal's choices correspond to assumptions which depend on the earlier clause choice for the family_D subgoal. Here we see that generative definitions for characteristic predicates result in zero-cost subgoals, whose choices depend on earlier annotated subgoal choices. Again, the combination of zero cost subgoals and non-zero cost subgoals suggests that we combine best-first-search with backtracking. Note, however, that the situation is simplified by the fact that whereas zero cost choices may depend on non-zero cost choices, the reverse is never true.

We are now in a position to extend the previous design for definite programs. The SETUP procedure remains unchanged. We extend the FIND_NEXT_fe procedure to provide a best-first-search over annotated subgoal choices. First, some definitions are presented.

An *annotated assumption* is an assumption belonging to an annotated subgoal. Note that an annotated assumption may have zero cost. This corresponds to the case where a particular assumption of an annotated subgoal has zero cost. Other choices for the subgoal will have non-zero cost.

An *unannotated assumption* is an assumption belonging to an unannotated subgoal.

We consider the current focus environment to be made up of three components:

(1) Optionally, a solution assumption.
(2) A set, designated as C_1, of annotated assumptions.
(3) A set, designated as C_2, of unannotated assumptions.

We adopt the chronological ordering of the backtracking algorithm described earlier. By definition, for any focus environment, we have that:

(1) The solution assumption, if it exists, is the oldest assumption.
(2) All the annotated assumptions are older than the unannotated assumptions.

We say that C_1 *is complete* if there are no left out annotated subgoals.

We say that C_2 *is complete* if there are no left out unannotated subgoals.

We say that *current is complete* if both C_1 and C_2 are complete.

FIND_NEXT_fe establishes a complete current focus environment to serve as the basis for further problem solving. Initially, a best-first search (procedure BFS) establishes a complete C_1. FIND_NEXT_fe then extends the current focus environment to include a complete C_2. When inconsistencies are encountered, FIND_NEXT_fe backtracks over the C_2 assumption space. Whenever forced to back up into the annotated assumption space, BFS is invoked to reestablish a new

complete C_1. The terminating condition occurs whenever the solution assumption itself comes into question.

```
[ FIND_NEXT_fe ]
   CASE
   { current = start }
        BFS
   { current = end }
        return
   { current = a solution environment }
        NEXT
   ENDCASE
   UNTIL current is consistent and complete
        CASE
        { current is consistent }
             EXTEND
        { no more solutions }
             REPLACE(end)
             Return
        { ¬end and C1 is consistent and C2 is not consistent }
             BACKTRACK
        { ¬end and C1 is not consistent }
             set fe2 = C2
             BFS
             If current = end  the return
             remove from fe2 any assumptions which depend on
             assumptions no longer included in current.
             REPLACE_C2(fe2)
        ENDCASE
   ENDUNTIL
   Return
```

Note that REPLACE_C2 is similar to REPLACE except that it only replaces C_2 assumptions of the current focus environment.

The procedure BFS does a best-first search over the annotated assumption space. The complete set of annotated subgoals are established by SETUP. The time of subgoal creation is once again used to provide a total ordering of these subgoals. We can therefore talk of the subgoals which precede and follow a particular subgoal.

The best-first search maintains a tree data structure representing the current state of the best-first search over the annotated assumption space. This tree is referred to as SS (Search Space). Except for the root node, each node in SS represents an annotated assumption. The root node represents the situation where no annotated assumptions have been made. If node N represents an assumption for an annotated subgoal SG, then its children correspond to the clause choices of the annotated subgoal which follows SG. The children of the root node correspond to the clause choices of the very first subgoal. For each node, the path back to the root node represents a distinct combination of assumptions. As we move from root to leaf node we encounter assumptions from each annotated subgoal in order of subgoal creation. Leaf nodes correspond to assumptions for the very last annotated subgoal. Figure 4.6 shows an example of a simple SS tree. In this example there are two annotated subgoals, A and B. There are three clause choices for subgoal A and two for subgoal B.

A path from the SS root node to a leaf is called a *complete branch.* Each complete branch corresponds to a complete C_1. A complete branch which, when established as C_1, does not activate any nogoods, is termed a *consistent complete branch.*

A path from the SS root node to a non-leaf is called a *partial branch.* A partial branch which when established as C_1, does not activate any nogoods, is termed a *consistent partial branch.*

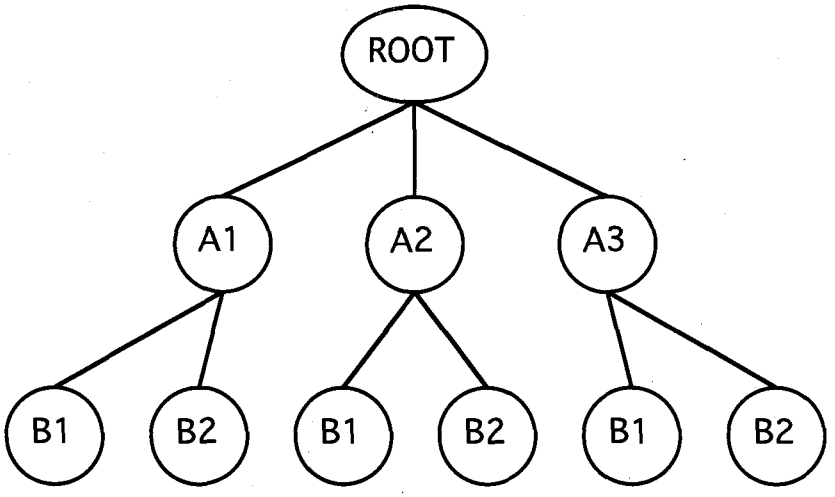


Figure 4.6. A simple SS tree

Starting at the root node the best-first procedure incrementally builds a path to a leaf node. At each step it extends its best partial branch by one assumption. If successful, this process results in a least-cost consistent complete branch. To ease memory requirements, SS branches are expanded only as they are deemed worth exploring. C_1 is kept in step with the best-first search. As assumptions are added or subtracted, C_1 is updated. Whenever an assumption is added to C_1, the ATMS is queried for active nogoods. Each SS branch which activates a nogood is deleted from further consideration. In this way SS is expanded as the search takes into consideration new search space points, and pruned as search space points are discovered to be nogood.

Each node in SS is a tuple of the form:

node(C,E,A,SG,Cs) where:

C subgoal clause choice

E estimated total cost of best solution which includes this SS branch

A actual cost of choices made so far (root node to here)

SG next subgoal to make a clause choice for

Cs list of children nodes. Each child represents a choice for SG.

We reference fields of a node by writing node.field. For example, node C is the subgoal clause choice associated with node.

For the root node C = *none* and A=0 by definition.

Following conventional best-first search algorithms, when a node is created we derive E = A + H where H is a heuristic estimate of the least cost path from this node to an SS leaf. In other words, H is based on estimating the cost of making additional assumptions for each left out annotated subgoal. As long as the H estimate is less than or equal to the correct H value, we are guaranteed of finding the overall least cost combination first. A search algorithm is said to be *admissible* if it always produces an optimal solution provided that a solution exists at all. A heuristic used to estimate H is said to be an *admissible heuristic* if it is guaranteed to generate estimates which are less than or equal to correct H values.

Each node is created when the partial branch ending with its parent node is first actively considered as part of a best complete branch. Each node is created with an empty children list. If, subsequently, this node comes under active consideration, its node representation is expanded to include its children. From this point on, the node's E is updated to reflect the smallest E value of any of its children.

The BFS procedure simply sets up BEST_fe which does the real work.

```
[ BFS ]
   (1) If current = start  then
         - NEW(∅)
         - estimate H
         - SG = first (earliest) annotated subgoal
         - ROOT = node(SG,none,H,0,nil)
```

(2) BEST_fe(ROOT,∞) → result
(3) If result = *never* then REPLACE(*end*)

Whenever BEST_fe(node,bound) is called, a partial branch ending in node's parent will have already been established for C_1. BEST_fe searches for a path from *node* to a leaf such that the combined complete branch is consistent and has a total cost of less than or equal to *bound*. BEST_fe returns one of {*no,never,yes* }. A *yes* indicates success. A *never* indicates that there are no complete branches which include the partial branch ending at this node. A *no* indicates that while there remain untried paths to a leaf, all have an estimated cost greater than *bound*.

[BEST_fe(node,bound)]
(1) set fe = C_1
(2) remove any assumptions ≥ node.C from fe
(3) add node.C to fe
(4) REPLACE_C1(fe)
(5) If C_1 is inconsistent then delete all children (and their children etc.) and return *never*.
(6) If C_1 is complete then return *yes*.
(7) If node.Cs = *nil* then EXPAND(node).
(8) find node child, N1 with smallest E.
(9) node.E = N1.E
(10) If N1.E > bound then return *no*.
(11) new_bound = bound
(12) If there is one, get the N1 sibling, N2 with smallest E. If N2.E < bound then new_bound = N2.E
(13) BEST_fe(N1,new_bound) → result.
(14) If result = *yes* then set node.E = N1.E and return *yes*.
(15) If result = *never* then delete N1 and determine if there exists an active nogood whose most recent assumption is earlier then node.SG. If so then delete all node's children.
(16) If node has no remaining children then return *never*.
(17) repeat from step (8).

[Expand(node)]
(1) SG = node.SG
(2) Let R be the set of left out annotated subgoals minus {SG}
(3) Estimate H from R
(4) Form set S consisting of SG's clause choices.
(5) Select ch, an element of S
(6) A = node.A + cost of ch
(7) E = A + H
(8) add new node(SG,ch,E,A,nil) to node.Cs list.
(9) repeat from (5) until S is empty

It remains to specify an admissible heuristic for estimating H. One possibility is to always consider H = 0. This corresponds to the trivially admissible assumption that we will encounter no additional costs in extending a partial branch into a complete branch. While this heuristic is indeed admissible, it has no heuristic power and does not provide any guidance for the search. Ideally, we would like to use a heuristic which is as close as possible to the real H, while still being admissible. At this point, this is a topic for further research.

Shimony and Charniak [Shimony,90] report on a best-first search algorithm for Bayesian network assignments. Their approach is based on mapping Bayesian networks into equivalent networks consisting of only nodes with boolean (ie: only 0 or 1) conditional probabilities. They then find a maximum a-posteriori assignment of values for the Bayesian network by using a best-first search on the new structure. The approach described here can be compared with that of Shimony and Charniak where the propositions stored in the ATMS play a similar role to the boolean structure of Shimony and Charniak's intermediate network. They report reasonable results for the trivial admissible heuristic H=0 and remark on their expectation of improved results through the use of a better admissible heuristic. Along these lines it may be possible to adapt the recent work of Henrion [Henrion,90], [Henrion,91] to obtain bounds on the relative probability of partial diagnoses.

We can adopt the interactive mechanism described earlier for detecting when a sufficient number of solutions have been discovered. Alternatively, the probabilities associated with computed answers can be used. For example, we may continue problem solving until the probability associated with the next best answer is below some minimal fraction of the first best answer. There are many possible heuristics which could be used. De Kleer [de Kleer,89a] adopts this approach in his ATMS-based diagnosis system. Conventional second order predicates for Prolog such as *bagof* can be generalized to allow the programmer to define his own thresholding heuristics.

The REFINE mechanism described earlier can also be used to incrementally refine Bayesian goals. This is particularly appropriate in interactive diagnosis systems which often recommend measurements or tests to reduce the number of likely explanations. New measurements are registered with our system as new variable binding premises. This triggers unification consumers which automatically eliminate diagnoses which are inconsistent with the new information. This, in turn, may affect our heuristic thresholding and result in a search for additional explanations. Here we see that goal refinement and heuristic second order predicates work together to provide flexible, high level support for interactive, abductive problem solving.

The design described in this section computes Bayesian answers in best-first order. Moreover, it supports the ongoing interactive comparison of more than one answer, and incremental query refinement. The main cost of these capabilities is that of best-first search as compared with backtracking. The well known disadvantage of a best-first search over backtracking is that it requires additional storage to maintain the search state (SS in our case). We keep this overhead to a minimum by resorting to backtracking for unannotated subgoals. A related disadvantage is the loss of the efficient stack-based memory management mechanism associated with Prolog backtracking implementations.

In the previous section we mentioned that most intelligent backtracking schemes discard information and recompute rather than suffer the memory and time overheads associated with storing all previous results and fetching them when appropriate. We briefly described an equivalent mechanism for ATMS-based backtracking involving purging inactive environments from node labels. We point out that in the above design we can adopt this same purging mechanism. The SS data structure ensures that no search space point is visited more than once. We point out, however, that the tradeoff is not clear. For instance the more nogoods we maintain, the more effectively SS is pruned, thereby limiting the principal disadvantage of best-first search.

Another consideration is that some environments, even though inactive, may have high probabilities (low costs) associated with them. For example, in a diagnosis application, single fault diagnoses are often much more likely than multiple fault diagnoses. However, the controller may try several candidates for the single fault before arriving at one which is consistent with observations. We will want to keep datums representing components which are healthy even when they are currently inactive, as they will quite likely be under active consideration again. In contrast, datums representing unhealthy components can be discarded as soon as they become inactive. In principle, node labels can be converted to probabilities, and heuristics established to control which results are kept and which are discarded. More research into these issues is required.

4.5. Introducing Constraints

As a result of their importance in model based diagnosis, we briefly mention the possibility of integrating constraints into Bayesian programs. Constraint suspension is a technique often used in model based diagnosis [Davis,88]. The basic idea is to model the normal behavior of each component in a network of components as a constraint on its interface properties. The result is a constraint

network. Observed property values are registered by providing bindings for property variables. Constraint propagation propagates the effect of these bindings on other variables. In a system where all input values are specified, constraint propagation derives the associated output values for a healthy system. If however, we observe output properties which are different from these expected values, then we need to relax or suspend one or more of the constraints in order to restore consistency. This corresponds to hypothesizing that a component is unhealthy. Often systems model unhealthy as well as healthy component behavior. In this case a constraint representing healthy behavior is replaced with one representing unhealthy behavior.

Recently, Constraint Logic Programming (CLP) languages [Jaffar,87a], [Jaffar,87b], [Colmerauer,90] have emerged which integrate constraint propagation techniques into logic programming. Echidna [Havens,90] and CHIP [Van Hentenryck,89] are CLP languages based on built-in constraint predicates and the notion of *domain variables*. Domain variables are logical variables with an associated restricted domain of possible values. For example, the variable X may be assigned the domain consisting of an integer interval from one to ten. During constraint propagation, domain variables can be bound to either a value contained in its domain, or a new domain which is a subset of its current domain. Echidna and CHIP both contain built-in constraint predicates for arithmetic operations such as multiplication, addition, boolean comparisons and others. Resolution is not used for constraint subgoals encountered during an SLD-refutation. Rather, a rule of inference based on *arc-consistency* [Mackworth,77], [Mackworth,85], [Mohr,86], [Sidebottom,91] is invoked each time a constraint variable is bound to a new, more specific domain. Under arc-consistency, each constraint variable is assigned a new value by eliminating values in its current domain which are not possible given the constraint and the current domain values of other variables in the constraint. Repeated invocation of this arc-consistency rule of inference propagates new variable bindings through constraints to other

variables and triggers backtracking in the event that any variable domain becomes the empty set.

Constraint propagation can be added to the consumer design defined for definite programs by defining *constraint class consumers* for constraint subgoals. These class consumers ensure that *constraint consumers* are attached to constraints and their constraint variables. A constraint consumer for a constraint subgoal C is attached to C and each set of variable datums representing all but one of C's variables. Upon execution, constraint consumers apply arc-consistency to derive a new binding for the left out constraint variable. By defining constraint consumers as class consumers, we ensure that arc-consistency processing is triggered after each new constraint variable binding.

This approach can be compared with Echidna which combines RMS-based dependency backtracking with constraint propagation. Unlike Echidna, however, the consumer based design derives minimal, subgoal-order independent labels.

Additional formalism is required to justify the use of constraints as a family predicate in a Bayesian program. Informally, we point out that constraint propagation executes only reasoning steps which follow from other choices or assumptions. As a result, constraints have a zero cost associated with them. They perform deterministic reasoning steps only. However, computed CLP answers can contain domain variables with non-ground answers. In this case we must interpret the computed probability mass as being distributed in some fashion over the possible ground values for the variable. In a similar situation, de Kleer [de Kleer,87], [de Kleer,89a] makes the assumption that the probability mass is evenly distributed. For example, if we have a computed answer for which domain variable X has either an integer value of 0 or 1 with computed probability P, then we can postulate the existence of a ground answer with X value of 1 with probability P/2. We must be tentative, however, since while arc-consistency guarantees local consistency, it does not guarantee the existence of globally consistent solutions. To find

global solutions CLP programs choose consistent ground values for remaining domain variables until all domain variables are both ground and arc-consistent. This process is both sound and complete. In a diagnosis system, this last stage can be interactive. The interactive user makes incremental measurements until all domain variables are ground and arc-consistent. Each interactive measurement is treated as an assumption and assigned a probability which correctly distributes the probability mass.

Chapter 5
Conclusion

This thesis addresses the need for a generalized approach to diagnostic or abductive problem solving. We have introduced a way of representing Bayesian networks as logic programs with extra-logic, probabilistic semantics. These Bayesian programs retain the dual procedural and declarative semantics of conventional definite programs. The probabilistic semantics of Bayesian programs provides relative rankings to abductive explanations. Explanations which are more likely are assigned higher rankings.

The advantage of Bayesian programs over Bayesian networks is based on the extra expressivity of predicate logic over propositional logic. Bayesian networks are propositional. They describe *particulars*. A Bayesian network is fully specified by explicitly enumerating the conditional probabilities of each network node. In contrast, Bayesian programs support intensional representations of equal-valued conditional probabilities.

This is of little value for problem domains which are not well understood, or for which we do not have a theory based on first principles. For such domains, our understanding is itself propositional. As a result, universally quantified variables are of little use. For domains with a well understood theory, the situation is different. Typically, there is a set of assumptions which sanction the theory. As long as these assumptions hold we have deterministic behavior, which can be represented intensionally. A Bayesian program can succinctly assert that the conditional probability of a healthy adder's output having a value, given any combination of adder input values, equals one, as long as the inputs add to form the output. This statement is true for all input, output value combinations. In other words, we can represent input, output values

as universally quantified variables rather than having to explicitly enumerate the complete set of input, output value combinations.

In the above adder example, the sanctioning assumption is that the adder is *healthy*. If the adder is not healthy, then the adder theory is no longer valid. Ultimately, as sanctioning assumptions are called into question, we must resort to statistical modeling. Bayesian programs represent an improvement over other model-based representations in that they support both statistical and theoretical modeling equally well. Bayesian programs provide a unified framework for modeling both idealized device theory, and the statistical nature of our incomplete understanding of the real world. Diagnostic completeness is a case in point. Model-based systems are implicitly sanctioned by the assumption that the system topology is as specified by the model. Model based systems enumerate possible diagnoses which are consistent with this assumption, however, they fail to consider the possibility of the assumption not holding. In this situation, a Bayesian program can explicitly resort to a completely general, probabilistic model of the system.

We have presented an architecture for computing answers to Bayesian program queries. This architecture is specifically designed to meet the needs of abductive, or diagnostic problem solving. The architecture combines best-first search with dependency backtracking in order to efficiently compute answers in order of decreasing likelihood. The architecture efficiently maintains multiple solutions and supports interactive, incremental query refinement. This enables an interactive user to compare more than one highly likely explanation or diagnosis, and to incrementally apply additional observations in order to reduce them in number. This is a process which occurs naturally in abductive reasoning applications.

There are remain several outstanding research issues as well as new directions to pursue. Firstly, there remain implementation level issues to resolve. Should the ATMS database ever discard any information, once it is established? Alternatively, should it act as a cache, retaining only datums which are either under active

consideration, or are likely to be needed again? The more nogoods, the more the search space is pruned. This is particularly important in our case as the best-first search maintains an explicit representation of the search space. Hence, by not pruning, we pay in memory storage as well as in execution time. One strategy is to convert datum labels to probabilities, and to discard information with low probabilities. This translation can be accomplished along the lines of D'Ambrosio's work [D'Ambrosio,90b].

Another issue is to decide upon an appropriate admissible heuristic for the best-first search. The work of Henrion [Henrion,90], [Henrion,91] is relevant to this issue.

Unlike intelligent backtracking schemes for Prolog, the architecture presented here generates general, order-independent labels for both datums and nogoods. This results in more efficient search space pruning, and in increased datum reuse. However, these benefits come at the expense of additional reasoning steps during problem solving. Additional research is needed to examine the nature of this tradeoff. Under what circumstances does this represent a net benefit?

One research direction is to extend Bayesian programs to include constraint processing as briefly described earlier. Hamscher [Hamscher,91] incorporates ATMS extensions to improve efficiency for ATMS-based constraint propagation.

De Kleer's SHERLOCK [de Kleer,89a] supports a myopic decision theory policy, based on an entropy calculation, for deciding which measurement to recommend taking next. Similar support could be built into the design presented here, resulting in a Prolog-like system with built in decision support. The basic architectural capabilities are in place, namely, the ability to maintain multiple solutions, each with an assigned probability. The programmer could reference the distinct solutions through the use of conventional second order predicates for Prolog-like *bag_of,* and *set_of.*

It should also be possible to offer generalized *bag_of,set_of* predicates which enable the programmer to specify probabilistic search cutoff criteria. In this way the programmer can exercise control over how many solutions are computed during problem solving.

The architecture presented here resembles a Blackboard architecture [Nii,89]. The ATMS is the blackboard, the consumers are the knowledge sources, and the controller is the monitor. This is not surprising as both blackboard systems and the design presented here are based on adaptive, opportunistic problem solving. Recent research in cooperative distributed problem solving (CDPS) [Durfee,89] makes substantial use of blackboard architectures. These systems are comprised of agents, each with their own blackboard, exchanging generalized problem solving results. Each agent independently decides what aspect of his part of the problem to focus on next. Much of CDPS research has to do with local decision making algorithms governing which partial results to transmit to a neighbor, and which local problem to focus on next. The intent is that local, distributed decision making should eventually lead to a single consistent global solution.

The local nature of message-based belief propagation algorithms [Pearl,88] for Bayesian networks, together with the blackboard-like nature of the architecture presented here suggests the possibility of a CDPS approach to abductive problem solving. Under this approach, a Bayesian program is partitioned among several problem-solving agents, each with its own consumer architecture for computing answers to Bayesian queries. The agents exchange belief messages summarizing the belief status of shared variables. Incoming messages trigger an agent's inferencing. Each agent takes into account the beliefs of its neighbors in forming a local Bayesian goal whose answer is most likely needed as part of an overall explanation. Once formed, this goal serves as a focus for local problem solving. Local inferencing results in updated belief status

messages to neighboring agents. The belief propagation protocol ensures convergence to a global solution.

Recent work [Bridgeland,90], [Mason,89] in distributed RMS systems is relevant to this approach.

Finally, additional research, in the form of real world applications, is required in order to explore the representational adequacy of Bayesian programs.

References

[Bacchus,90] Bacchus,F., Lp, a logic for representing and reasoning with statistical knowledge. Computational Intelligence, Vol. 6, Number 4, November 1990, pp. 209-231.

[Bridgeland,90] Bridgeland,D.M. and Huhns,M.N., Distributed Truth Maintenance,in Proc. 8th National Conference on Artificial Intelligence, Vol. 1,1990, pp. 72-77.

[Bruynooghe,84] Bruynooghe, M. and Pereira, L.M., Deduction Revision by Intelligent Backtracking. In Implementations of Prolog, pp.194-215, Ellis Horwood Limited, 1984.

[Buchanan,84] Buchanan, Shortliffe (eds), Rule-Based Expert Systems : The MYCIN Experiments of The Stanford Heuristics Programming Project, Addison-Wesley, 84.

[Charniak,87] Charniak,E. and McDermott, D., Introduction to Artificial Intelligence, Addison-Wesley Publishing Company, 1987.

[Charniak,90] Charniak,E. and Shimony, S.E., Probabilistic semantics for cost-based abduction. Technical Report Cs-90-02, Computer Science Department, Brown University, Febrauary 1990.

[Cheeseman,88] Cheeseman,P., An inquiry into computer understanding. Computational Intelligence, 4(1): 58-66, 1988 .

[Colmerauer,90] Colmerauer, A., An Introduction to Prolog III, Communications of the ACM 33 (7), pp. 69-90.

[Cox,84] Cox,P.T., Finding backtrack points for intelligent backtracking. In Implementations of Prolog, pp.216-233, Ellis Horwood Limited, 1984.

[D'Ambrosio,87] D'Ambrosio, B., Truth Maintenance with Numeric Certainty Estimates. In Proceedings Third Conference on AI Applications, pp. 244-249, Kissimmee, Florida, Computer Society of the IEEE, February, 1987.

[D'Ambrosio,90a] D'Ambrosio,B., Process, Structure, and Modularity in Reasoning with Uncertainty, Uncertainty in Artificial Intelligence 4, 1990, pp. 15-25.

[D'Ambrosio,90b] D'Ambrosio,B., Incremental Construction and Evaluation of Defeasible Probabilistic Models. International Journal of Approximate Reasoning, vol. 4, no. 4, p. 233-60, July, 1990.

[D'Ambrosio,90c] Shachter, R.D., D'Ambrosio,B. and DelFavero,B., Symbolic probabilistic inference in belief networks. In Proceedings 8th National Conference on AI, pages 126-131. AAAI, August, 1990.

[D'Ambrosio,91] D'Ambrosio,B. and Edwards,J., A Partitioned ATMS, Proceedings, Seventh IEEE Conference on Artificial Intelligence Applications, p. 330-6, Feb., 1991.

[Davis,84] Davis, R., Diagnostic Reasoning Based on Structure and Behaviour, Artificial Intelligence 24 (1984) 347-410.

[Davis,88] Davis, R., and Hamscher, W., Model-based Reasoning: Troubleshooting, in Exploring Artificial Intelligence, edited by H.E. Shrobe and the American Association for Artificial Intelligence, (Morgan Kaufman, 1988), 297-346.

[Dechter,91] Dechter, R. and Pearl, J., Directed Constraint Networks: A Relational Framework for Causal Modeling, proc AAAI Spring Symposium on Constraint Reasoning, Stanford, Ca, pp 110-127, March,1991.

[de Kleer,86a] de Kleer, J., An Assumption-Based Truth Maintenance System, Artificial Intelligence 28 (1986), 127-162.

[de Kleer,86b] de Kleer, J., Extending the ATMS, Artificial Intelligence 28 (1986), 163-196.

[de Kleer,86c] de Kleer, J., Problem Solving with the ATMS, Artificial Intelligence 28 (1986), 197-223.

[de Kleer,86d] de Kleer, J., and Williams, B.C., Back to Backtracking: Controlling the ATMS. In Proc. 5th National Conf. on Artificial Intelligence, pages 910-917, Philadelphia, PA,August 1986.

[de Kleer,86e] de Kleer, J. and Brown, J.S., Theories of Causal Ordering, Artificial Intelligence 29, 1986, pp. 33-61.

[de Kleer,87] de Kleer, J. and Williams, B.C, Diagnosing Multiple Faults, Artificial Intelligence 32(1) (1987), 97-130.

[de Kleer,88] de Kleer, J., A General Labeling Algorithm for Assumption-Based Truth Maintenance. In Proc. 7th

National Conference on Artificial Intelligence, pages 188-192, St. Paul, Minnesota, August 1988.

[de Kleer,89a] de Kleer, J. and Williams, B.C, Diagnosis with Behavioral Modes, in: Proceedings IJCAI-89, Detroit, MI (1989), 1324-1330.

[de Kleer,89b] de Kleer, J., A Comparison of ATMS and CSP Techniques, Proceedings IJCAI-89, Detroit, MI (1989), 290-296.

[de Kleer,90a] de Kleer, J., A.K. Mackworth, R. Reiter, Characterizing Diagnoses, In Proceedings of AAAI-90, 1990, p. 324-330.

[de Kleer,90b] de Kleer, J., Using Crude Probability Estimates to Guide Diagnosis, Research Note, Artificial Intelligence, 45(3), 1990.

[Doyle,79] Doyle, J., A Truth Maintenance System, Artificial Intelligence 12 (1979), 231-272.

[Drakos,88] Drakos, N., Reason Maintenance in Horn Clause Logic Programs, in Smith & Kelleher (eds.) Reason Maintenance Systems and their Applications, John Wiley and Sons, Toronto.

[Durfee,89] Durfee,E.H.,Lesser,V.R. and Corkill,D.D., Cooperative Distributed Problem Solving, in Barr,A.,Cohen,P.R. and Feigenbaum,E.A. (eds.),The Handbook of Artificial intelligence, Vol IV,Addison-Wesley Publishing Company,1989, pp. 85-147.

[Forbus,88a] Forbus, K.D., and de Kleer, J., Focusing the ATMS. In Proc. 7th National Conf. on Artificial Intelligence, pages 192-198, Minneapolis, MN, 1988.

[Forbus,88b] Forbus, K.D., Qualitative Physics: Past, Present, and Future, in Exploring Artificial Intelligence, edited by H.E. Shrobe and the American Association for Artificial Intelligence, (Morgan Kaufman, 1988), 239-296.

[Genesereth,84] Genesereth, M.R., The Use of Design Descriptions in Automated Diagnosis, Artificial Intelligence 24 (1984), 411-436.

[Goebel,90] Goebel, R., A quick review of hypothetical reasoning based on abduction, Proc. of the AAAI Spring

Symposium on Automated Abduction, Stanford U., March 27-29,1990, pp. 145-149.

[Hamscher,90a] Hamscher, W., XDE: Diagnosing Devices with Hierarchic Structure and Known Component Failure Modes. IEEE Conference on AI Applications, 1990.

[Hamscher,90b] Hamscher, W., Modelling Digital Circuits for Troubleshooting: An Overview, IEEE Conference on AI Application, March 90

[Hamscher,91] Hamscher, W., Reason Maintenance and Inference Control for Const Propagation over Intervals. In Proc AAAI Spring Symposium on Constraint Reasoning, Stanford, Ca, pp 93-97, March,1991.

[Havens,90] Havens, W.S., S. Sidebottom, G. Sidebottom, J. Jones, M. Cuperman, R. Davison, Echidna Constraint Reasoning System: Next Generation Expert System Technology, Simon Fraser University Technical Report, CSS-IS TR 90-09.

[Havens,91] Havens, W.S., Dataflow Dependency Backtracking in a new CLP Language, proc AAAI Spring Symposium on Constraint Reasoning, Stanford, Ca, pp 110-127, March,1991.

[Henrion,88] Henrion, M., Propagation of uncertainty by probabilistic logic sampling in Bayes' networks. In Uncertainty in Artificial Intelligence, Vol. 2, J. Lemmer & L.N. Kanal (Eds.),1988, North-Holland, amsterdam, pp149-164.

[Henrion,90] Henrion, M., Towards efficient probabilistic diagnosis in multiply connected belief networks. In Influence Diagrams, Belief Nets, and Decision Analysis, R.M. Oliver & J.Q. Smith (eds.),1990, Wiley, London.

[Henrion,91] Henrion, M., Search-based Methods to Bound Diagnostic Probabilities in Very Large Belief Nets. In Proc. of the Seventh Conference on Uncertainty in Artificial Intelligence, UCLA, July, 1991, pp142-150.

[Iwasaki,86a] Iwasaki,Y. and Simon,H., Causality in Device Behaviour, Artificial Intelligence 29, 1986, pp. 3-32.

[Iwasaki,86b] Iwasaki,Y. and Simon,H., Theories of Causal Ordering: Reply to de Kleer and Brown, Artificial Intelligence 29, 1986, pp. 63-72.

[Jaffar,87a] Jaffar, J. and J.L. Lassez, Constraint Logic Programming, In Proc. Fourteenth ACM POPL Conf., Munich, 1987.

[Jaffar,87b] Jaffar, J. and S. Michaylov, Methodology and Implementation of a CLP System, In Proc. Fourth International Conference in Logic Programming, Melbourne, 1987.

[Jensen,89a] Jensen,F.V.,Olesen,K.G. and Andersen,S.K., An Algebra of Bayesian Belief Universes for Knowledge-Based Systems, Networks, Vol. 20 (1990) pp. 637-659.

[Jensen,89b] Jensen,F.V.,Lauritzen,S.L. and Olesen,K.G. Bayesian Updating in Recursive Graphical Models by Local Computations, Technical Report R-89-15. ,Institute of Electronic Systems, Aalborg University, Aalborg, Denmark, June 1989.

[Lloyd,84] Lloyd,J.W., Foundations of logic Programming. Spinger Verlag, New York.1984.

[Mackworth,77] Mackworth, A.K., Consistency in Network Relations, Artificial Intelligence, 8, pp. 99-118, 1977.

[Mackworth,85] Mackworth, A.K., E.C. Freuder, The Complexity of Some Polynomial Network Consistency Algorithm for Constraint Satisfaction Problems, Artificial Intelligence, 25, pp.65-74, 1985.

[Mason,89] Mason, C.L. and Johnson,R.R., Datms: A framework for Distributed Assumption Based Reasoning. In Les Gasser and Michael N. Huhns, editors, Distributed Artificial Intelligence, Vol. II, pages 293-317. Pitman Publishing, London, 1989.

[McCarthy,80] McCarthy,J., Circumscription - a form of non-monotonic reasoning. Artificial Intelligence, 13: 27-39, 1980.

[McDermott,80] McDermott,D. and Doyle,J., Non-monotonic logic I, Artificial Intelligence, 25, pp.41-72, 1980.

[McDermott,87] McDermott,D., A critique of pure reason. Computational Intelligence, 3: 151-160, 1987.

[Mohr,86] Mohr, R. and Henderson, T.C., Arc and Path Consistency Revisited, Reseach Note, Artificial Intelligence, 28, pp 225-233, 1986.

[Nii,89] Nii,H.P., Blackboard Systems, in Barr,A.,Cohen,P.R. and Feigenbaum,E.A. (eds.),The Handbook of Artificial intelligence, Vol IV,Addison-Wesley Publishing Company,1989, pp. 3-82.

[Papoulis,65] Papoulis,A., Probability, Random Variables, and Stochastic Processes, McGraw-Hill,Inc., 1965.

[Pearl,88] Pearl, J., Probabilistic Reasoning in Intelligent Systems: Networks of Plausible inference, Morgan Kaufmann Publishers, Inc. San Mateo, California, 1988.

[Poole,87] Poole, D.L., Goebel,R.G., and Aleliunas,R., Theorist: A logical reasoning system for defaults and diagnosis, in: N. Cercone and G. McCalla (Eds.), The Knowledge Frontier: Essays in the Representation of Knowledge (Springer, New York, 1987) pp. 331-352.

[Poole,88] Poole, D., A Logical Framework for Default Reasoning, Artificial Intelligence, 36 (1988), pp. 27-47.

[Poole,91] Poole, D., Representing Bayesian Networks within Probabilistic Horn Abduction. In Proc. of the Seventh Conference on Uncertainty in Artificial Intelligence, UCLA, July, 1991, pp271-278.

[Reiter,80] Reiter, R., A Logic for Default Reasoning,Artificial, Intelligence 13 (1980), 81-132.

[Reiter,87a] Reiter, R., A Theory of diagnosis from the first principles, Artificial Intelligence 32 (1987), 57-95.

[Reiter,87b] Reiter, R., and de Kleer, J., Foundations of Assumption-Based Truth Maintenance Systems: Preliminary Report, Proceedings of the National Conference on Artificial Intelligence, Seattle, WA (july, 1987), 183-188.

[Selman,90] Selman,B. and Levesque,H.J., Abductive and Default Reasoning: A Computational Core. In Proc. of the 8th National Conference on Artificial Intelligence, Vol. 1, 1990, pp. 343-348.

[Shachter,86] Shachter, R.D.,Evaluating influence diagrams, Operations Research Vol. 34, No. 6, November-December 1986, pp 871-882.

[Shachter,88] Shachter, R.D.,Probabilistic Inference and Influence Diagrams, Operations Research Vol. 36, No. 4, July-August, 1988, pp 589-604.

[Shimony,90] Shimony,S.E. and Charniak,E., A new algorithm for finding MAP assignments to belief networks. In Proc. of Sixth Conference on Uncertainty in AI, Cambridge, Ma.,1990, pp 98-103.

[Sidebottom,91] Sidebottom, G., W.S. Havens, Hierarchical Arc Consistency Applied to Numeric Constraints Processing in Logic Programming, Simon Fraser University Technical Report, in preparation.

[Struss,89] Struss, P. and Dressler, O., "Physical negation"- Integrating Fault Fodels into the General Diagnosis Engine, in: Proceedings IJCAI-89, Detroit, MI (1989), 1318-1323.

[Van Hentenryck,89] Van Hentenryck, P., Constraint Satisfaction in Logic Programming, MIT Press, Cambridge, Mass. 1989.

[Warren,77] Warren, D.H.D., Implementing Prolog: Compiling Predicate Logic Programs, D.A.I. Research Report nos. 39 & 40, Univ. of Edinburgh, Scotland. 1977.

[Watanabe,87] Watanabe,S.,Inductive ambiguity and the limits of artificial intelligence. Computational Intelligence, 3: 304-309,1987.

[You,89] You, J.H. and Wong, B., Intelligent Backtracking in Prolog Made Practical, tech. rep. TR89-4, Dept. of Comp. Science, U. of Alberta, Edmonton, Alberta, Canada, January 1989.