CHAPTER 7

SYMBOLIC REASONING UNDER UNCERTAINTY

There are many methods for predicting the future. For example, you can read horoscopes, tea leaves, tarot cards, or crystal balls. Collectively, these methods are known as 'nutty methods.' Or you can put well-researched facts into sophisticated computer models, more commonly referred to as "a complete waste of time."

—Scott Adams
(1957-) Author Known for his comic strip Dilbert

So far, we have described techniques for reasoning with a complete, consistent, and unchanging model of the world. Unfortunately, in many problem domains it is not possible to create such models. In this chapter and the next, we explore techniques for solving problems with incomplete and uncertain models.

7.1 INTRODUCTION TO NONMONOTONIC REASONING

In their book, *The Web of Belief,* Quine and Ullian [1978] provide an excellent discussion of techniques that can be used to reason effectively even when a complete, consistent, and constant model of the world is not available. One of their examples, which we call the ABC Murder story, clearly illustrates many of the main issues that such techniques must deal with. Quoting Quine and Ullian [1978]:

Let Abbott, Babbitt, and Cabot be suspects in a murder case. Abbott has an alibi, in the register of a respectable hotel in Albany. Babbitt also has an alibi, for his brother-in-law testified that Babbitt was visiting him in Brooklyn at the time. Cabot pleads alibi too, claiming to have been watching a ski meet in the Catskills, but we have only his word for that. So we believe

1. That Abbott did not commit the crime
2. That Babbitt did not commit the crime
3. That Abbott or Babbitt or Cabot did.

But presently Cabot documents his alibi—he had the good luck to have been caught by television in the sidelines at the ski meet. A new belief is thus thrust upon us:

4. That Cabot did not.

Our beliefs (1) through (4) are inconsistent, so we must choose one for rejection. Which has the weakest evidence? The basis for (1) in the hotel register is good, since it is a fine old hotel. The basis for (2) is weaker, since Babbitt's brother-in-law might be lying. The basis for (3) is perhaps twofold: that there is no sign of burglary and that only Abbott, Babbitt, and Cabot seem to have stood to gain from the murder apart from burglary. This exclusion of burglary seems conclusive, but the other consideration does not; there could be some fourth beneficiary. For (4), finally, the basis is conclusive: the evidence from television. Thus (2) and (3) are the weak points. To resolve the inconsistency of (1) through (4) we should reject (2) or (3), thus either incriminating Babbitt or widening our net for some new suspect.

See also how the revision progresses downward. If we reject (2), we also revise our previous underlying belief, however tentative, that the brother-in-law was telling the truth and Babbitt was in Brooklyn. If instead we reject (3), we also revise our previous underlying belief that none but Abbott, Babbitt, and Cabot stood to gain from the murder apart from burglary.

Finally, a certain arbitrariness should be noted in the organization of this analysis. The inconsistent beliefs (1) through (4) were singled out, and then various further beliefs were accorded a subordinate status as underlying evidence: a belief about a hotel register, a belief about the prestige of the hotel, a belief about the television, a perhaps unwarranted belief about the veracity of the brother-in-law, and so on. We could instead have listed this full dozen of beliefs on an equal footing, appreciated that they were in contradiction, and proceeded to restore consistency by weeding them out in various ways. But the organization lightened our task. It focused our attention on four prominent beliefs among which to drop one, and then it ranged the other beliefs under these four as mere aids to choosing which of the four to drop.

The strategy illustrated would seem in general to be a good one: divide and conquer. When a set of beliefs has accumulated to the point of contradiction, find the smallest selection of them you can that still involves contradiction; for instance, (1) through (4). For we can be sure that we are going to have to drop some of the beliefs in that subset, whatever else we do. In reviewing and comparing the evidence for the beliefs in the subset, then, we will find ourselves led down in a rather systematic way to other beliefs of the set. Eventually we find ourselves dropping some of them too.

In probing the evidence, where do we stop? In probing the evidence for (1) through (4) we dredged up various underlying beliefs, but we could have probed further, seeking evidence in turn for them. In practice, the probing stops when we are satisfied how best to restore consistency: which ones to discard among the beliefs we have canvassed.

This story illustrates some of the problems posed by uncertain, fuzzy, and often changing knowledge. A variety of logical frameworks and computational methods have been proposed for handling such problems. In this chapter and the next, we discuss two approaches:

- Nonmonotonic reasoning, in which the axioms and/or the rules of inference are extended to make it possible to reason with incomplete information. These systems preserve, however, the property that, at any given moment, a statement is either believed to be true, believed to be false, or not believed to be either.
- Statistical reasoning, in which the representation is extended to allow some kind of numeric measure of certainty (rather than simply TRUE or FALSE) to be associated with each statement.

Other approaches to these issues have also been proposed and used in systems. For example, it is sometimes the case that there is not a single knowledge base that captures the beliefs of all the agents involved in solving a problem. This would happen in our murder scenario if we were to attempt to model the reasoning of Abbott, Babbitt, and Cabot, as well as that of the police investigator. To be able to do this reasoning, we would require a technique for maintaining several parallel *belief spaces*, each of which would correspond to the beliefs of one agent. Such techniques are complicated by the fact that the belief spaces of the various agents, although not identical, are sufficiently similar that it is unacceptably inefficient to represent them as completely separate knowledge bases. In Section 15.4.2 we return briefly to this issue. Meanwhile, in the rest of this chapter, we describe techniques for nonmonotonic reasoning.

Conventiotnal reasoning systems, such first-order predicate logic, are designed to work with information that has three important properties:

- It is complete with respect to the domain of interest. In other words, all the facts that are necessary to solve a problem are present in the system or can be derived from those that are by the conventional rules of first-order logic.
- It is consistent.
- The only way it can change is that new facts can be added as they become available. If these new facts are consistent with all the other facts that have already been asserted, then nothing will ever be retracted from the set of facts that are known to be true. This property is called *monotonicity.*

Unfortunately, if any of these properties is not satisfied, conventional logic-based reasoning systems become inadequate. Nonmonotonic reasoning systems, on the other hand, are designed to be able to solve problems in which all of these properties may be missing.

In order to do this, we must address several key issues, including the following:

1. *How can the knowledge base be extended to allow inferences to be made on the basis of lack of knowledge as well as on the presence of it?* For example, we would like to be able to say things like, "If you have no reason to suspect that a particular person committed a crime, then assume he didn't," or "If you have no reason to believe that someone is not getting along with her relatives, then assume that the relatives will try to protect her." Specifically, we need to make clear the distinction between:
   - It is known that $\neg P$.
   - It is not known whether P.

   First-order predicate logic allows reasoning to be based on the first of these. We need an extended system that allows reasoning to be based on the second as well. In our new system, we call any inference that depends on the lack of some piece of knowledge a *nonmonotonic inference.*[1]

   Allowing such reasoning has a significant impact on a knowledge base. Nonmonotonic reasoning systems derive their name from the fact that because of inferences that depend on lack of knowledge, knowledge bases may not grow monotonically as new assertions are made. Adding a new assertion may invalidate an inference that depended on the absence of that assertion. First-order predicate logic systems, on the other hand, are monotonic in this respect. As new axioms are asserted, new wff's may become provable, but no old proofs ever become invalid.

   In other words, if some set of axioms T entails the truth of some statement w, then T combined with another set of axioms N also entails w. Because nonmonotonic reasoning does not share this property, it is also called *defeasible:* a nonmonotonic inference may be defeated (rendered invalid) by the addition of new information that violates assumptions that were made during the original reasoning process. It turns out, as we show below, that making this one change has a dramatic impact on the structure of the logical system itself. In particular, most of our ideas of what it means to find a proof will have to be reevaluated.
2. *How can the knowledge base be updated properly when a new fact is added to the system (or when an old one is removed)?* In particular, in nonmonotonic systems, since the addition of a fact can cause

[1] Recall that in Section 2.4, we also made a monotonic/nonmonotonic distinction. There the issue was classes of production systems. Although we are applying the distinction to different entities here, it is essentially the same distinction in both cases, since it distinguishes between systems that never shrink as a result of an action (monotonic ones) and ones that can (nonmonotonic ones).

previously discovered proofs to be become invalid, how can those proofs, and all the conclusions that depend on them be found? The usual solution to this problem is to keep track of proofs, which are often called *justifications.* This makes it possible to find all the justifications that depended on the absence of the new fact, and those proofs can be marked as invalid. Interestingly, such a recording mechanism also makes it possible to support conventional, monotonic reasoning in the case where axioms must occasionally be retracted to reflect changes in the world that is being modeled. For example, it may be the case that Abbott is in town this week and so is available to testify, but if we wait until next week, he may be out of town. As a result, when we discuss techniques for maintaining valid sets of justifications, we talk both about nonmonotonic reasoning and about monotonic reasoning in a changing world.

3. *How can knowledge be used to help resolve conflicts when there are several in consistent nonmonotonic inferences that could be drawn?* It turns out that when inferences can be based on the lack of knowledge as well as on its presence, contradictions are much more likely to occur than they were in conventional logical systems in which the only possible contradictions were those that depended on facts that were explicitly asserted to be true. In particular, in nonmonotonic systems, there are often portions of the knowledge base that are locally consistent but mutually (globally) inconsistent. As we show below, many techniques for reasoning nonmonotonically are able to define the alternatives that could be believed, but most of them provide no way to choose among the options when not all of them can be believed at once.

To do this, we require additional methods for resolving such conflicts in ways that are most appropriate for the particular problem that is being solved. For example, as soon as we conclude that Abbott, Babbitt, and Cabot all claim that they didn't commit a crime, yet we conclude that one of them must have since there's no one else who is believed to have had a motive, we have a contradiction, which we want to resolve in some particular way based on other knowledge that we have. In this case, for example, we choose to resolve the conflict by finding the person with the weakest alibi and believing that he committed the crime (which involves believing other things, such as that the chosen suspect lied).

The rest of this chapter is divided into five parts. In the first, we present several logical formalisms that provide mechanisms for performing nonmonotonic reasoning. In the last four, we discuss approaches to the implementation of such reasoning in problem-solving programs. For more detailed descriptions of many of these systems, see the papers in Ginsberg [1987].

7.2 LOGICS FOR NONMONOTONIC REASONING

Because monotonicity is fundamental to the definition of first-order predicate logic, we are forced to find some alternative to support nonmonotonic reasoning. In this section, we look at several formal approaches to doing this. We examine several because no single formalism with all the desired properties has yet emerged (although there are some attempts, e.g., Shoham [1987] and Konolige [1987], to present a unifying framework for these several theories). In particular, we would like to find a formalism that does all of the following things:

- Defines the set of possible worlds that could exist given the facts that we do have. More precisely, we will define an *interpretation* of a set of wff's to be a domain (a set of objects) *D*, together with a function that assigns: to each predicate, a relation (of corresponding arity); to each n-ary function, an operator that maps from *D”* into *D;* and to each constant, an element of *D*. A *model* of a set of wff's is an interpretation that satisfies them. Now we can be more precise about this requirement. We require a mechanism for defining the set of models of any set of wff's we are given.
- Provides a way to say that we prefer to believe in some models rather than others.

- Provides the basis for a practical implementation of this kind of reasoning.
- Corresponds to our intuitions about how this kind of reasoning works. In other words, we do not want vagaries of syntax to have a significant impact on the conclusions that can be drawn within our system.

As we examine each of the theories below, we need to evaluate how well they perform each of these tasks. For a more detailed discussion of these theories and some comparisons among them, see Reiter [1987a], Etherington [1988], and Genesereth and Nilsson[1987].

Before we go into specific theories in detail, let's consider Fig. 7.1, which shows one way of visualizing how nonmonotonic reasoning works in all of them. The box labeled *A* corresponds to an original set of wff's. The large circle contains all the models of *A*. When we add some nonmonotonic reasoning capabilities to *A*, we get a new set of wff's, which we've labeled *B*.[2] *B* (usually) contains more information than *A* does. As a result, fewer models satisfy 5 than *A*. The set of models corresponding to *B* is shown at the lower right of the large circle. Now suppose we add some new wff's (representing new information) to *A*. We represent *A* with these additions as the box *C*. A difficulty may arise, however, if the set of models corresponding to *C* is as shown in the smaller, interior circle, since it is disjoint with the models for *B*. In order to find a new set of models that satisfy *C*, we need to accept models that had previously been rejected. To do that, we need to eliminate the wff's that were responsible for those models being thrown away. This is the essence of nonmonotonic reasoning.

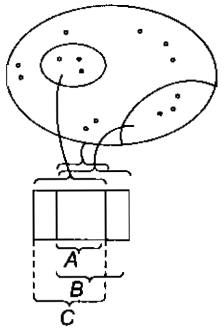


Fig. 7.1 *Models, Wff's, and Non-monotonic Reasoning*

7.2.1 Default Reasoning

We want to use nonmonotonic reasoning to perform what is commonly called *default reasoning*. We want to draw conclusions based on what is most likely to be true. In this section, we discuss two approaches to doing this.

- Nonmonotonic Logic[3]
- Default Logic

We then describe two common kinds of nonmonotonic reasoning that can be defined in those logics:

- Abduction
- Inheritance

Nonmonotonic Logic

One system that provides a basis for default reasoning is *Nonmonotonic Logic* (NML) [McDermott and Doyle, 1980], in which the language of first-order predicate logic is augmented with a modal operator M, which can be read as "is consistent." For example, the formula

$$\forall x, y : Related(x, y) \wedge \mathrm{M}\ GetAlong(x, y) \rightarrow \neg WillDefend(x, y)$$

should be read as, "For all *x* and *y*, if *x* and *y* are related and if the fact that *x* gets along with *y* is consistent with everything else that is believed, then conclude that *x* will defend *y*."

[2] As we will see below, some techniques add inference rules, which then generate wff's, while others add wff's directly. We'll ignore that difference for the moment.

[3] Try not to get confused about names here. We are using the terms "nonmonotonic reasoning" and "default reasoning" generically to describe a kind of reasoning. The terms "Nonmonotonic Logic" and "Default Logic" are, on the other hand, being used to refer to specific formal theories.

Once we augment our theory to allow statements of this form, one important issue must be resolved if we want our theory to be even semidecidable. (Recall that even in a standard first-order theory, the question of theoremhood is undecidable, so semide-cidability is the best we can hope for.) We must define what "is consistent" means. Because consistency in this system, as in first-order predicate logic, is undecidable, we need some approximation. The one that is usually used is the PROLOG notion of negation as failure, or some variant of it. In other words, to show that P is consistent, we attempt to prove ¬P. If we fail, then we assume ¬ to be false and we call P consistent. Unfortunately, this definition does not completely solve our problem. Negation as failure works in pure PROLOG because, if we restrict the rest of our language to Horn clauses, we have a decidable theory. So failure to prove something means that it is not entailed by our theory. If, on the other hand, we start with full first-order predicate logic as our base language, we have no such guarantee. So, as a practical matter, it may be necessary to define consistency on some heuristic basis, such as failure to prove inconsistency within some fixed level of effort.

A second problem that arises in this approach (and others, as we explain below) is what to do when multiple nonmonotonic statements, taken alone, suggest ways of augmenting our knowledge that if taken together would be inconsistent. For example, consider the following set of assertions:

$$\forall x : Republican(x) \wedge \mathrm{M}\ \neg Pacifist(x) \rightarrow \neg Pacifist(x)$$
$$\forall x : Quaker(x) \wedge \mathrm{M}\ Pacifist(x) \rightarrow Pacifist(x)$$
$$Republican(Dick)$$
$$Quakev\{Dick)$$

The definition of NML that we have given supports two distinct ways of augmenting this knowledge base. In one, we first apply the first assertion, which allows us to conclude ¬*Pacifist*(*Dick*). Having done that, the second assertion cannot apply, since it is not consistent to assume *Pacifist*(*Dick*). The other thing we could do, however, is apply the second assertion first. This results in the conclusion *Pacifist*(*Dick*), which prevents the first one from applying. So what conclusion does the theory actually support?

The answer is that NML defines the set of theorems that can be derived from a set of wff's A to be the intersection of the sets of theorems that result from the various ways in which the wff's of A might be combined. So, in our example, no conclusion about Dick's pacifism can be derived. This theory thus takes a very conservative approach to theoremhood.

It is worth pointing out here that although assertions such as the ones we used to reason about Dick's pacifism look like rules, they are, in this theory, just ordinary wff's which can be manipulated by the standard rules for combining logical expressions. So, for example, given

$$A \wedge \mathrm{M}\ B \rightarrow B$$
$$\neg A \wedge \mathrm{M}\ B \rightarrow B$$

we can derive the expression

$$\mathrm{M}\ B \rightarrow B$$

In the original formulation of NML, the semantics of the modal operator M, which is self-referential, were unclear. A more recent system, *Autoepistemic Logic* [Moore, 1985] is very similar, but solves some of these problems.

[4] Reiter's original notation had ":M" in place of ":", but since it conveys no additional information, the M is usually omitted.

Default Logic

An alternative logic for performing default-based reasoning is Reiter's *Default Logic* (DL) [Reiter, 1980], in which a new class of inference rules is introduced. In this approach, we allow inference rules of the form[4]

$$\frac{A : B}{C}$$

Such a rule should be read as, "If A is provable and it is consistent to assume B then conclude C." As you can see, this is very similar in intent to the nonmonotonic expressions that we used in NML. There are some important differences between the two theories, however. The first is that in DL the new inference rules are used as a basis for computing a set of plausible *extensions* to the knowledge base. Each extension corresponds to one maximal consistent augmentation of the knowledge base.[5] The logic then admits as a theorem any expression that is valid in any extension. If a decision among the extensions is necessary to support problem solving, some other mechanism must be provided. So, for example, if we return to the case of Dick the Republican, we can compute two extensions, one corresponding to his being a pacifist and one corresponding to his not being a pacifist. The theory of DL does not say anything about how to choose between the two. But see Reiter and Criscuolo [1981], Touretzky [1986], and Rich [1983] for discussions of this issue.

A second important difference between these two theories is that, in DL, the nonmonotonic expressions are rules of inference rather than expressions in the language. Thus they cannot be manipulated by the other rules of inference. This leads to some unexpected results. For example, given the two rules

$$\frac{A : B}{C} \qquad \frac{\neg A : B}{B}$$

and no assertion about A, no conclusion about B will be drawn, since neither inference rule applies.

Abduction

Standard logic performs deduction. Given two axioms:

$$\forall x : A(x) \rightarrow B(x)$$
$$A(C)$$

we can conclude $B(C)$ using deduction. But what about applying the implication in reverse? For example, suppose the axiom we have is.

$$\forall x : Measles(x) \rightarrow Spots(x)$$

The axiom says that having measles implies having spots. But suppose we notice spots. We might like to conclude measles. Such a conclusion is not licensed by the rules of standard logic and it may be wrong, but it may be the best guess we can make about what is going on. Deriving conclusions in this way is thus another form of default reasoning. We call this specific form *abductive reasoning*. More precisely, the process of abductive reasoning can be described as, "Given two wff's $(A \rightarrow B)$ and (B), for any expressions A and B, if it is consistent to assume A, do so."

In many domains, abductive reasoning is particularly useful if some measure of certainty is attached to the resulting expressions. These certainty measures quantify the risk that the abductive reasoning process is

[5] What we mean by the expression "maximal consistent augmentation" is that no additional default rules can be applied without violating consistency. But its is important to note that only expressions generated by the application of the stated inference rules to the original knowledge are allowed in an extension. Gratuitous additions are not permitted.

wrong, which it will be whenever there were other antecedents besides *A* that could have produced *B*. We discuss ways of doing this in Chapter 8.

Abductive reasoning is not a kind of logic in the sense that DL and NML are. In fact, it can be described in either of them. But it is a very useful kind of nonmonotonic reasoning, and so we mentioned it explicitly here.

Inheritance

One very common use of nonmonotonic reasoning is as a basis for inheriting attribute values from a prototype description of a class to the individual entities that belong to the class. We considered one example of this kind of reasoning in Chapter 4, when we discussed the baseball knowledge base. Recall that we presented there an algorithm for implementing inheritance. We can describe informally what that algorithm does by saying, "An object inherits attribute values from all the classes of which it is a member unless doing so leads to a contradiction, in which case a value from a more restricted class has precedence over a value from a broader class." Can the logical ideas we have just been discussing provide a basis for describing this idea more formally? The answer is yes. To see how, let's return to the baseball example (as shown in Figure 4.5) and try to write its inheritable knowledge as rules in DL.

We can write a rule to account for the inheritance of a default value for the height of a baseball player as:

$$\frac{\mathit{Baseball\text{-}Player}(x) : \mathit{height}(x, \text{6-1})}{\mathit{height}(x, \text{6-1})}$$

Now suppose we assert *Pitcher(Three-Finger-Brown)*. Since this enables us to conclude that *Three-Finger-Brown* is a baseball player, our rule allows us to conclude that his height is 6-1. If, on the other hand, we had asserted a conflicting value for Three Finger' had an axiom like

$$\forall x, y, z : \mathit{height}(x, y) \wedge \mathit{height}(x, z) \rightarrow y = z,$$

which prohibits someone from having more than one height, then we would not be able to apply the default rule. Thus an explicitly stated value will block the inheritance of a default value, which is exactly what we want. (We'll ignore here the order in which the assertions and the rules occur. As a logical framework, default logic does not care. We'll just assume that somehow it settles out to a consistent state in which no defaults that conflict with explicit assertions have been asserted. In Section 7.5.1 we look at issues that arise in creating an implementation that assures that.)

But now, let's encode the default rule for the height of adult males in general. If we pattern it after the one for baseball players, we get

$$\frac{\mathit{Adult\text{-}Male}(x) : \mathit{height}(x, \text{5-10})}{\mathit{height}(x, \text{5-10})}$$

Unfortunately, this rule does not work as we would like. In particular, if we again assert *Pitcher(Three-Finger-Brown),* then the resulting theory contains two extensions: one in which our first rule fires and Brown's height is 6-1 and one in which this new rule applies and Brown's height is 5-10. Neither of these extensions is preferred. In order to state that we prefer to get a value from the more specific category, baseball player, we could rewrite the default rule for adult males in general as:

$$\frac{\mathit{Adult\text{-}Male}(x) : \neg \mathit{Baseball\text{-}Player}(x) \wedge \mathit{height}(x, \text{5-10})}{\mathit{height}(x, \text{5-10})}$$

This effectively blocks the application of the default knowledge about adult males in the case that more specific information from the class of baseball players is available.

Unfortunately, this approach can become unwieldy as the set of exceptions to the general rule increases. For example, we could end up with a rule like:

$$\frac{Adult\text{-}Male(x) : \neg Baseball\text{-}Player(x) \wedge \neg Midget(x) \wedge - Jockey(x) \wedge height(x, 5\text{-}10)}{height(x,\ 5\text{-}10)}$$

What we have done here is to clutter our knowledge about the general class of adult males with a list of all the known exceptions with respect to height. A clearer approach is to say something like, "Adult males typically have a height of 5-10 unless they are abnormal in some way." We can then associate with other classes the information that they are abnormal in one or another way. So we could write, for example:

$$\forall x\text{: } Adult\text{-}Male(x) \wedge \neg AB(x, aspect1) \rightarrow height(x, 5\text{-}10)$$
$$\forall x : Baseball\text{-}Player(x) \rightarrow AB(x, aspect\ 1)$$
$$\forall x : Midget(x) \rightarrow AB(x, aspect\ 1)$$
$$\forall x : Jockey(x) \rightarrow AB(x, aspect\ 1)$$

Then, if we add the single default rule:

$$\frac{: \neg\, AB(x,\ y)}{\neg\, AB(x,\ y)}$$

we get the desired result.

7.2.2 Minimalist Reasoning

So far, we have talked about general methods that provide ways of describing things that are generally true. In this section we describe methods for saying a very specific and highly useful class of things that are generally true. These methods are based on some variant of the idea of a *minimal model*. Recall from the beginning of this section that a model of a set of formulas is an interpretation that satisfies them. Although there are several distinct definitions of what constitutes a minimal model, for our purposes, we will define a model to be minimal if there are no other models in which fewer things are true. (As you can probably imagine, there are technical difficulties in making this precise, many of which involve the treatment of sentences with negation.) The idea behind using minimal models as a basis for nonmonotonic reasoning about the world is the following: "There are many fewer true statements than false ones. If something is true and relevant it makes sense to assume that it has been entered into our knowledge base. Therefore, assume that the only true statements are those that necessarily must be true in order to maintain the consistency of the knowledge base." We have already mentioned (in Section 6.2) one kind of reasoning based on this idea, the PROLOG concept of negation as failure, which provides an implementation of the idea for Horn clause-based systems. In the rest of this section we look at some logical issues that arise when we remove the Horn clause limitation.

The Closed World Assumption

A simple kind of minimalist reasoning is suggested by the *Closed World Assumption* or CWA [Reiter, 1978]. The CWA says that the only objects that satisfy any predicate *P* are those that must. The CWA is particularly powerful as a basis for reasoning with databases, which are assumed to be complete with respect to the properties they describe. For example, a personnel database can safely be assumed to list all of the company's employees. If someone asks whether Smith works for the company, we should reply "no" unless he is explicitly listed as an employee. Similarly, an airline database can be assumed to contain a complete list of all the routes flown by that airline. So if I ask if there is a direct flight from Oshkosh to El Paso, the answer should be "no" if none can be found in the database. The CWA is also useful as a way to deal with *AB* predicates, of the sort we introduced in Section 7.2.1, since we want to take as abnormal only those things that are asserted to be so.

Although the CWA is both simple and powerful, it can fail to produce an appropriate answer for either of two reasons. The first is that its assumptions are not always true in the world; some parts of the world are not realistically "closable." We saw this problem in the murder story example. There were facts that were relevant to the investigation that had not yet been uncovered and so were not present in the knowledge base. The CWA will yield appropriate results exactly to the extent that the assumption that all the relevant positive facts are present in the knowledge base is true.

The second kind of problem that plagues the CWA arises from the fact that it is a purely syntactic reasoning process. Thus, as you would expect, its results depend on the form of the assertions that are provided. Let's look at two specific examples of this problem.

Consider a knowledge base that consists of just a single statement:

$$A(Joe) \vee B(Joe)$$

The CWA allows us to conclude both ? $A(Joe)$ and ?$B(Joe)$, since neither A nor 6 must necessarily be true of Joe. Unfortunately, the resulting extended knowledge base

$$A(Joe) \vee B(Joe)$$
$$\neg A(Joe)$$
$$\neg B(Joe)$$

is inconsistent.

The problem is that we have assigned a special status to positive instances of predicates, as opposed to negative ones. Specifically, the CWA forces completion of a knowledge base by adding the negative assertion *P whenever it is consistent to do so. But the assignment of a real world property to some predicate P and its complement to the negation of P may be arbitrary. For example, suppose we define a predicate *Single* and create the following knowledge base:

$$Single(John)$$
$$Single(Mary)$$

Then, if we ask about Jane, the CWA will yield the answer $\neg Single\{Jane)$. But now suppose we had chosen instead to use the predicate *Married* rather than *Single*. Then the corresponding knowledge base would be

$$\neg Married(Johri)$$
$$\neg Married(Mary)$$

If we now ask about Jane, the CWA will yield the result $\neg Married(Jane)$.

Circumscription

Although the CWA captures part of the idea that anything that must not necessarily be true should be assumed to be false, it does not capture all "of it. It has two essential limitations:

- It operates on individual'predicates without considering the interactions among predicates that are defined in the knowledge hase. We saw an example of this above when we considered the statement $A(Joe) \vee B(Joe)$.
- It assumes that all predicates have all of their instances listed. Although in many database applications this is true, in many knowledge-based systems it is not. Some predicates can reasonably be assumed to

be completely defined (i.e., the part of the world they describe is closed), but others cannot (i.e., the part of the world they describe is open). For example, the predicate *has-a-green-shirt* should probably be considered open since in most situations it would not be safe to assume that one has been told all the details of everyone else's wardrobe.

Several theories *of circumscription* (e.g., McCarthy [1980], McCarthy [1986], and Lifschitz [1985]) have been proposed to deal with these problems. In all of these theories, new axioms are added to the existing knowledge base. The effect of these axioms is to force a minimal interpretation on "a selected portion of the knowledge base. In particular, each specific axiom describes a way that the set of values for which a particular axiom of the original theory is true is to be delimited (i.e., circumscribed).

As an example, suppose we have the simple assertion

$$\forall x : Adult(x) \wedge \neg AB(x, aspect1) \rightarrow Literate(x)$$

We would like to circumscribe *AB*, since we would like it to apply only to those individuals to which it applies. In essence, what we want to do is to say something about what the predicate *AB* must be (since at this point we have no idea what it is; all we know is its name). To know what it is, we need to know for what values it is true. Even though we may know a few values for which it is true (if any individuals have been asserted to be abnormal in this way), there are many different predicates that would be consistent with what we know so far. Imagine this universe of possible binary predicates. We might ask, which of these predicates could be *AB?* We want to say that *AB* can only be one of the predicates that is true only for those objects that we know it must be true for. We can do this by adding a (second order) axiom that says that *AB* is the smallest predicate that is consistent with our existing knowledge base.

In this simple example, circumscription yields the same result as does the CWA since there are no other assertions in the knowledge base with which a minimization of *AB* must be consistent. In both cases, the only models that are admitted are ones in which there are no individuals who are abnormal in *aspect 1*. In other words, *AB* must be the predicate FALSE.

But, now let's return to the example knowledge base

$$A(Joe) \vee B(Joe)$$

If we circumscribe only *A*, then this assertion describes exactly those models in which A is true of no one and *B* is true of at least *Joe*. Similarly, if we circumscribe only *B*, then we will accept exactly those models in which *B* is true of no one and *A* is true of at least *Joe*. If we circumscribe *A* and *B* together, then we will admit only those models in which *A* is true of only *Joe* and *B* is true of no one or those in which B is true of only *Joe* and *A* is true of no one. Thus, unlike the CWA, circumscription allows us to describe the logical relationship between *A* and *B*.

7.3 IMPLEMENTATION ISSUES

Although the logical frameworks that we have just discussed take us part of the way toward a basis for implementing nonmonotonic reasoning in problem-solving programs, they are not enough. As we have seen, they all have some weaknesses as logical systems. In addition, they fail to deal with four important problems that arise in real systems.

The first is how to derive exactly those nonmonotonic conclusions that are relevant to solving the problem at hand while not wasting time on those that, while they may be licensed by the logic, are not necessary and are not worth spending time on.

The second problem is how to update our knowledge incrementally as problem-solving progresses. The definitions of the logical systems tell us how to decide on the truth status of a proposition with respect to a given truth status of the rest of the knowledge base. Since the procedure for-doing this is a global one (relying on some form of consistency or minimality), any change to the knowledge base may have far-reaching consequences. It would be computationally intractable to handle this problem by starting over with just the facts that are explicitly stated and reapplying the various nonmonotonic reasoning steps that were used before, this time deriving possibly different results.

The third problem is that in nonmonotonic reasoning systems, it often happens that more than one interpretation of the known facts is licensed by the available inference rules. In Reiter's terminology, a given nonmonotonic system may (and often does) have several extensions at the moment, even though many of them will eventually be eliminated as new knowledge becomes available. Thus some kind of search process is necessary. How should it be managed?

The final problem is that, in general, these theories are not computationally effective. None of them is decidable. Some are semidecidable, but only in their propositional forms. And none is efficient.

In the rest of this chapter, we discuss several computational solutions to these problems. In all of these systems, the reasoning process is separated into two parts: a problem solver that uses whatever mechanism it happens to have to draw conclusions as necessary and a truth maintenance system whose job is just to do the bookkeeping required to provide a solution to our second problem. The various logical issues we have been discussing, as well as the heuristic ones we have raised here are issues in the design of the problem solver. We discuss these issues in Section 7.4. Then in the following sections, we describe techniques for tracking nonmonotonic inferences so that changes to the knowledge base are handled properly. Techniques for doing this can be divided into two classes, determined by their approach to the search control problem:

- Depth-first, in which we follow a single, most likely path until come new piece of information comes in that forces us to give up this path and find another.
- Breadth-first, in which we consider all the possibilities as equally likely. We consider them as a group, eliminating some of them as newfacts become available. Eventually, it may happen that only one (or a small number) turn out to be consistent with everything we come to know.

It is important to keep in mind throughout the rest of this discussion that there is no exact correspondence between any of the logics that we have described and any of the implementations that we will present. Unfortunately, the details of how the two can be brought together are still unknown.

7.4 AUGMENTING A PROBLEM-SOLVER

So far, we have described a variety of logical formalisms, all of which describe the theorems that can be derived from a set of axioms. We have said nothing about how we might write a program that solves problems using those axioms. In this section, we do that.

As we have already discussed several times, problem-solving can be done using either forward or backward reasoning. Problem-solving using uncertain knowledge is no exception. As a result, there are two basic approaches to this kind of problem-solving (as well as a variety of hybrids):

- Reason forward from what is known. Treat nonmonotonically derivable conclusions the same way monotonically derivable ones are handled. Nonmonotonic reasoning systems that support this kind of reasoning allow standard forward-chaining rules to be augmented with *unless* clauses, which introduce a basis for reasoning by default. Control (including deciding which default interpretation to choose) is handled in the same way that all other control decisions in the system are made (whatever that may be, for example, via rule ordering or the use of metarules).

- Reason backward to determine whether some expression *P* is true (or perhaps to find a set of bindings for its variables that make it true). Nonmonotonic reasoning systems that support this kind of reasoning may do either or both of the following two things'.
  - Allow default (unless) clauses in backward rules. Resolve conflicts among defaults using the same, control strategy that is used for other kinds of reasoning (usually rule ordering).
  - Support a kind of debate in which an attempt is made to construct arguments both in favor of *P* and opposed to it. Then some additional knowledge is applied to the arguments to determine which side has the stronger case.

Let's look at backward reasoning first. We will begin witb the simple case of backward reasoning in which we attempt to prove (and possibly to find bindings for) an expression *P*. Suppose that we bave a knowledge base that consists of the backward rules shown in Fig. 7.2.

```
Suspect(x) ← Beneficiary(x)
                 UNLESS Alibi(x)
Alibi(x) ← SomewhereElse(x)
SomewhereElse(x) ← RegisteredHotel(x, y) and FarAway(y)
                        UNLESS ForgedRegister(y)
Alibi(x) ← Defends(x, y)
               UNLESS Lies(y)
SomewhereElse(x) ← PictureOf(x, y) and FarAway(y)
Contradiction() ← TRUE
                      UNLESS ∃x: Suspect(x)
Beneficiary (Abbott)
Beneficiary(Babbitt)
Beneficiary(Cabot)
```

Fig. 7.2 *Backward Rules Using UNLESS*

Assume that the problem solver that is using this knowledge base uses the usual PROLOG-style control structure in which rules are matched top to bottom, left to right. Then if we ask the question? *Suspect(x)*, the program will first try Abbott, who is a fine suspect given what we know now, so it will return Abbott as its answer. If we had also included the facts

RegisteredHotel(Abbott, Albany)
FarAway(Albany)

then, the program would have failed to conclude that Abbott was a suspect and it would instead have located Babbitt.

As an alternative to this approach, consider the idea of a debate. In debating systems, an attempt is made to find multiple answers. In the ABC Murder story case, for example, all three possible suspects would be considered. Then some attempt to choose among the arguments would be made. In this case, for example, we might want to have a choice rule that says that it is more likely that people will lie to defend themselves than to defend others. We might have a second rule that says that we prefer to believe hotel registers rather than people. Using these two rules, a problem solver would conclude that the most likely suspect is Cabot.

Backward rules work exactly as we have described if all of the required facts are present when the rules are invoked. But what if we begin with the situation shown in Fig. 7.2 and conclude that Abbott is our suspect. Later, we are told that he was registered at a hotel in Albany. Backward rules will never notice that anything has changed. To make our system data-driven, we need to use forward rules. Figure 7.3 shows how the same knowledge could be represented as forward rules. Of course, what we probably want is a system that can exploit both. In such a system, we could use a backward rule whose goal is to find a suspect, coupled with forward rules that fire as new facts that are relevant to finding a suspect appear.

```
If: Beneficiary(x),
     UNLESS Alibi(x),
then Suspect(x)
If: SomewhereElse(x),
then Alibi(x)
If: RegisteredHotel(x, y), and
     FarAway(y),
     UNLESS ForgedRegister(y),
then SomewhereElse(x)
If Defends(x,y),
     UNLESS Lies(y),
then Alibi(x)
If PictureOf(x, y), and
     FarAway(y),
then SomewhereElse(x)
If TRUE,
     UNLESS ∃x: Suspect(x)
then Contradiction()
Beneficiary(Abbott)
Beneficiary(Babbitt)
Beneficiary(Cabot)
```

Fig. 7.3 *Forward Rules Using UNLESS*

7.5 IMPLEMENTATION: DEPTH-FIRST SEARCH

7.5.1 Dependency-Directed Backtracking

If we take a depth-first approach to nonmonotonic reasoning, then the following scenario is likely to occur often: We need to know a fact, F, which cannot be derived monotonically from what we already know, but which can be derived by making some assumption A which seems plausible. So we make assumption A, derive F, and then derive some additional facts G and H from F. We later derive some other facts M and N, but they are completely independent of A and F. A little while later, a new fact comes in that invalidates A. We need to rescind our proof of F, and also our proofs of G and H since they depended on F. But what about M and N? They didn't depend on F, so there is no logical need to invalidate them. But if we use a conventional backtracking scheme, we have to back up past conclusions in the order in which we derived them. So we have to backup past M and N, thus undoing them, in order to get back to F, G, H and A. To get around this problem, we need a slightly different notion of backtracking, one that is based on logical dependencies rather than the chronological order in which decisions were made. We call this new method *dependency-directed backtracking* [Stallman and Sussman, 1977], in contrast to *chronological backtracking*, which we have been using up until now.

Before we go into detail on how dependency-directed backtracking works, it is worth pointing out that although one of the big motivations for it is in handling nonmonotonic reasoning, it turns out to be useful for conventional search programs as well. This is not too surprising when you consider that what any depth-first search program does is to "make a guess" at something, thus creating a branch in the search space. If that branch eventually dies out, then we know that at least one guess that led to it must be wrong. It could be any guess along the branch. In chronological backtracking we have to assume it was the most recent guess and back up there to try an alternative. Sometimes, though, we have additional information that tells us which guess caused the problem. We'd like to retract only that guess and the work that explicitly depended on it, leaving everything else that has happened in the meantime intact. This is exactly what dependency-directed backtracking does.

As an example, suppose we want to build a program that generates a solution to a fairly simple problem, such as-finding a time at which three busy people can all attend a meeting. One way to solve such a problem is first to make an assumption that the meeting will be held on some particular day, say Wednesday, add to the database an assertion to that effect, suitably tagged as an assumption, and then proceed to find a time, checking along the way for any inconsistencies in people's schedules. If a conflict arises, the statement representing the assumption must be discarded and replaced by another, hopefully noncontradictory, one. But, of course, any statements that have been generated along the way that depend on the now-discarded assumption must also be discarded.

Of course, this kind of situation can be handled by a straightforward tree search with chronological backtracking. All assumptions, as well as the inferences drawn from them, are recorded at the search node that created them. When a node is determined to represent a contradiction, simply backtrack to the next node from which there remain unexplored paths. The assumptions and their inferences will disappear automatically. The drawback to this approach is illustrated in Fig. 7.4, which shows part of the search tree of a program that is trying to schedule a meeting. To do so, the program must solve a constraint satisfaction problem to find a day and time at which none of the participants is busy and at which there is a sufficiently large room available.

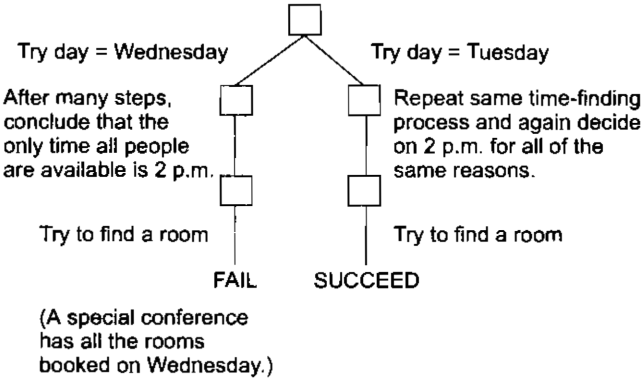

Try day = Wednesday
Try day = Tuesday
After many steps, conclude that the only time all people are available is 2 p.m.
Repeat same time-finding process and again decide on 2 p.m. for all of the same reasons.
Try to find a room
Try to find a room
FAIL
SUCCEED
(A special conference has all the rooms booked on Wednesday.)

Fig. 7.4 *Nondependency-Directed Backtracking*

In order to solve the problem, the system must try to satisfy one constraint at a time. Initially, there is little reason to choose one alternative over another, so it decides to schedule the meeting on Wednesday. That creates a new constraint that must be met by the rest of the solution. The assumption that the meeting will be held on Wednesday is stored at the node it generated. Next the program tries to select a time at which all participants are available. Among them, they have regularly scheduled daily meetings at all times except 2:00. So 2:00 is chosen as the meeting time. But it would not have mattered which day was chosen. Then the program discovers that on Wednesday there are no rooms available. So it backtracks past the assumption that the day would be Wednesday and tries another day, Tuesday. Now it must duplicate the chain of reasoning that led it to choose 2:00 as the time because that reasoning was lost when it backtracked to redo the choice of day. This occurred even though that reasoning did not depend in any way on the assumption that the day would be Wednesday. By withdrawing statements based on the order in which they were generated by the search process rather than on the basis of responsibility for inconsistency, we may waste a great deal of effort.

If we want to use dependency-directed backtracking instead, so that we do not waste this effort, then we need to do the following things:

- Associate with each node one or more justifications. Each justification corre sponds to a derivation process that led to the node. (Since it is possible to derive the same node in several different ways, we want to allow for the possibility of multiple justifications.) Each justification must contain a list of all the nodes (facts, rules, assumptions) on which its derivation depended.
- Provide a mechanism that, when given a contradiction node and its justification, computes the "no-good" set of assumptions that underlie the justification. The no-good set is defined to be the minimal set of assumptions such that if you remove any element from the set, the justification will no longer be valid and the inconsistent node will no longer be believed.

- Provide a mechanism for considering a no-good set and choosing an assumption to retract.
  Provide a mechanism for propagating the result of retracting an assumption. This mechanism must cause all of the justifications that depended, however indirectly, on the retracted assumption to become invalid.

In the next two sections, we will describe two approaches to providing such a system.

7.5.2 Justification-Based Truth Maintenance Systems

The idea of a truth maintenance system or TMS [Doyle, 1979] arose as a way of providing the ability to do dependency-directed backtracking and so to support nonmonotonic reasoning. There was a later attempt to rename it to Reason Maintenance System (a bit less pretentious), but since the old name has stuck, we use it here.

A TMS allows assertions to be connected via a spreadsheet-like network of dependencies. In this section, we describe a simple form of truth maintenance system, a justification-based truth maintenance system (or JTMS). In a JTMS (or just TMS for the rest of this section), the TMS itself does not know anything about the structure of the assertions themselves. (As a result, in our examples, we use an English-like shorthand for representing the contents of nodes.) The TMS's only role is to serve as a bookkeeper for a separate problem-solving system, which in turn provides it with both assertions and dependencies among assertions.

To see how a TMS works, let's return to the ABC Murder story. Initially, we might believe that Abbott is the primary suspect because he was a beneficiary of the deceased and he had no alibi. There are three assertions here, a specific combination of which we now believe, although we may change our beliefs later. We can represent these assertions in shorthand as follows:

- *Suspect Abbott* (Abbott is the primary murder suspect.)
- *Beneficiary Abbott* (Abbott is a beneficiary of the victim.)
- *Alibi Abbott* (Abbott was at an Albany hotel at the time.)

Our reason for possible belief that Abbott is the murderer is nonmonotonic. In the notation of Default Logic, we can state the rule that produced it as

$$\frac{Beneficiary(x) : \neg Alibi(x)}{Suspect(x)}$$

or we can write it as a backward rule as we did in Section 7.4.

If we currently believe that he is a beneficiary and we have no reason to believe he has a valid alibi, then we will believe that he is our suspect. But if later we come to believe that he does have a valid alibi, we will no longer believe Abbott is a suspect.

But how should belief be represented and how should this change in belief be enforced? There are various *ad hoc* ways we might do this in a rule-based system. But they would all require a developer to construct rules carefully for each possible change in belief. For instance, we would have to have a rule that said that if Abbott ever gets an alibi, then we should erase from the database the belief that Abbott is a suspect. But suppose that we later fire a rule that erases belief in Abbott's alibi. Then we need another rule that would reconclude that Abbott is a suspect. The task of creating a rule set that consistently maintains beliefs when new assertions are added to the database quickly becomes unmanageable. In contrast, a TMS dependency network offers a purely syntactic, domain-independent way to represent belief and change it consistently.

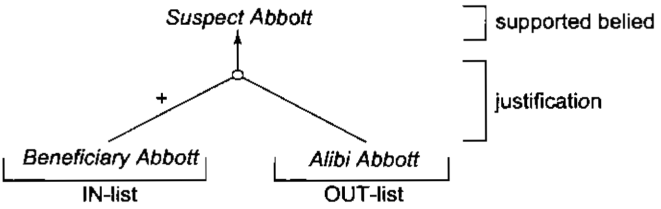


Fig. 7.5 *A Justification*

Figure 7.5 shows how these three facts would be represented in a dependency network, which can be created as a result of ap[illegible]ed TMS *justification.* Each justification consists of two parts: an *IN-list* and an *OUT-list.* In the figure, the assertions on the IN-list are connected to the justification by "+" links, those on the OUT-list by "–" links. The justification is connected by an arrow to the assertion that it supports. In the justification shown, there is exactly one assertion in each list. *Beneficiary Abbott* is in the IN-list and *Alibi Abbott* is in the OUT-list. Such a justification says that Abbott should be a suspect just when it is believed that he is a beneficiary and it is not believed that he has an alibi.

More generally, assertions (usually called nodes) in a TMS dependency network are believed when they have a valid justification. A justification is *valid* if every assertion in the IN-list is believed and none of those in the OUT-list is. A justification is nonmonotonic if its OUT-list is not empty, or, recursively, if any assertion in its IN-list has a nonmonotonic justification. Otherwise, it is monotonic. In a TMS network, nodes are labeled with a *belief status.* If the assertion corresponding to the node should be believed, then in the TMS it is labeled IN. If there is no good reason to believe the assertion, then it is labeled OUT. What does it mean that an assertion "should be believed" or has no "good" reason for belief?

A TMS answers these questions for a dependency network in a way that is independent of any interpretation of the assertions associated with the nodes. The *labeling* task of a TMS is to label each node so that two criteria about the dependency network structure are met. The first criterion is *consistency:* every node labeled IN is supported by at least one valid justification and all other nodes are labeled OUT. More specifically than before, a justification is valid if every node in its IN-list is labeled IN and every node in its OUT-list is labeled OUT. Notice that in Fig. 7.5, all of the assertions would have to be labeled OUT to be consistent. *Alibi Abbott* has no justification at all, much less a valid one, and so must be labeled OUT. But the same is true for *Beneficiary* Abbott, so it must be OUT as well. Then the justification for *Suspect Abbott* is invalid because an element of its IN-list is labeled OUT. *Suspect Abbott* would then be labeled OUT as well. Thus status labels correspond to our belief or lack of it in assertions, and justifications Correspond to our reasons for such belief, with valid justifications being our "good" reasons. Notice that the label OUT may indicate that we have specific reason to believe that a node represents an assertion that is not true, or it may mean simply that we have no information one way or the other.

But the state of affairs in Fig. 7.5 is incomplete. We are told that Abbott is a beneficiary. We have no further justification for this fact; we must simply accept it. For such facts, we give a *premise* justification: a justification with empty IN- and OUT-lists. Premise justifications are always valid. Figure 7.6 shows such a justification added to the network and a consistent labeling for that network, which shows *Suspect Abbott* labeled IN.

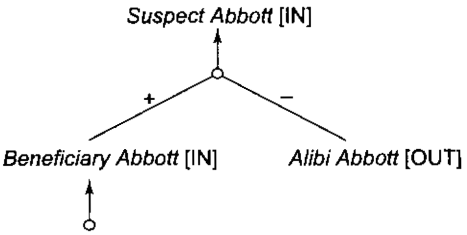


Fig. 7.6 *Labeled Nodes with Premise Justification*

That Abbot is the primary suspect represents an initial state of the murder investigation. Subsequently, the detective establishes that Abbott is listed on the register of a good Albany hotel on the day of the murder. This provides a valid reason to believe Abbott's alibi. Figure 7.7 shows the effect of adding such a justification to the network, assuming that we have used forward (data-driven) rules as shown in Fig. 7.3 for all of our reasoning except possibly establishing the top-level goal. That Abbott was registered at the hotel. *Registered Abbott,* was told to us and has a premise justification and so is labeled IN. That the hotel is far away is also asserted as a premise. The register might have been forged, but we have no good reason to believe it was. Thus *Register Forged* lacks any justification and is labeled OUT. That Abbott was on the register of a far away hotel and the lack of belief that the register was forged will cause the appropriate forward rule to fire and create a justification for *Alibi-Abbott,* which is thus labeled IN. This means that *Suspect Abbott* no longer has a valid justification and must be labeled OUT. Abbott is no longer a suspect.

Notice that such a TMS labeling carefully avoids saying that the register definitely was *not* forged. It only says that there is currently no good reason to believe that it was. Just like our original reason for believing that Abbott was a suspect, this is a nonmonotonic justification. Later, if we find that Abbott was secretly married to the desk clerk, we might add to this network a justification that would reverse some of the labeling. Babbitt will have a similar justification based upon lack of belief that his brother-in-law lied as shown in Fig. 7.8 (where *B-I-L* stands for "Brother-In-Law").

Abbott's changing state showed how consistency was maintained. There is another criterion that the TMS must meet in labeling a dependency network: *well-foundedness* (i.e., the proper grounding of a chain of justifications on a set of nodes that do not themselves depend on the nodes they support). To illustrate this, consider poor Cabot: Not only does he have fewer *bs* and *ts* in his name, he also lacks a valid justification for his alibi that he was at a ski show. We have only his word that he was. Ignoring the more complicated representation of lying, the simple dependency network in Fig. 7.9 illustrates the fact that the only support for the alibi of attending the ski show is that Cabot is telling the truth about being there. The only support for his telling the truth would be if we knew he was at the ski show. But this is a circular argument. Part of the task of a TMS is to disallow such arguments. In particular, if the support for a node only depends on an unbroken chain of positive links (IN-list links) leading back to itself then that node must be labeled OUT if the labeling is to be well-founded.

The TMS task of ensuring a consistent, well-founded labeling has now been outlined. The other major task of a TMS is resolving contradictions. In a TMS, a contradiction node does not represent a logical contradiction but rather a state of the database explicitly declared to be undesirable. (In the next section, we describe a slightly different kind of TMS in which this is not the case.) In our example, we

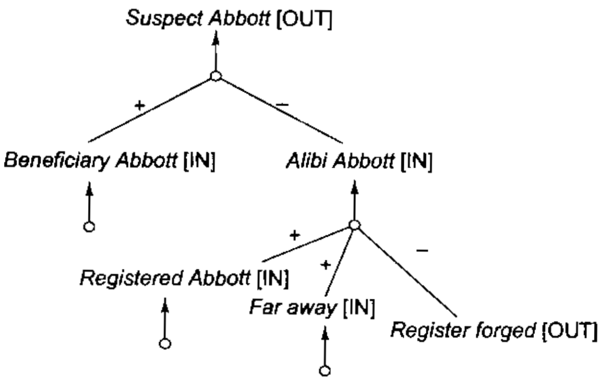


Fig. 7.7 *Changed Labeling*

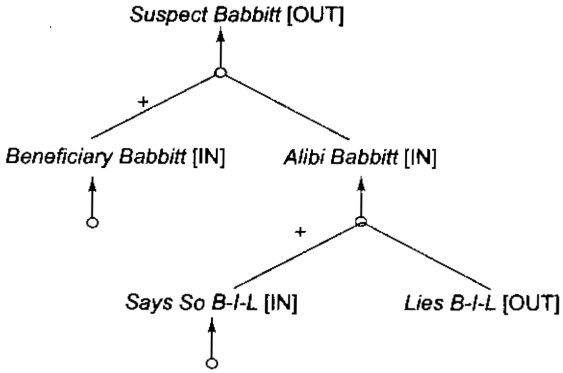


Fig. 7.8 *Babbitt's Justification*

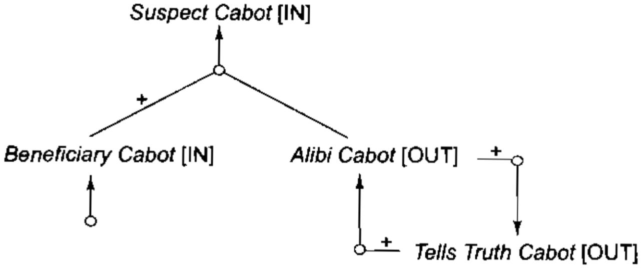


Fig. 7.9 *Cabot's justification*

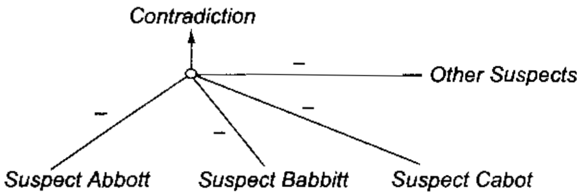


Fig. 7.10 *A Contradiction*

have a contradiction if we do not have at least one murder suspect. Thus a contradiction might have the justification shown in Fig. 7.10, where the node *Other Suspects* means that there are suspects other than Abbott, Babbitt, and Cabot. This is one way of explicitly representing an instance of the closed world assumption. Later, if we discover a long-lost relative, this will provide a valid justification for *Other Suspects*. But for now, it has none and must be labeled OUT. Fortunately, even though Abbott and Babbitt are not suspects, *Suspect Cabot* is labeled IN, invalidating the justification for the contradiction. While the contradiction is labeled OUT, there is no contradiction to resolve.

Now we learn that Cabot was seen on television attending the ski tournament. Adding this to the dependency network first illustrates the fact that nodes can have more than one justification as shown in Fig. 7.11. Not only does Cabot say he was at the ski slopes, but he was seen there on television, and we have no reason to believe that this was an elaborate forgery. This new valid justification of *Alibi Cabot* causes it to be labeled IN (which also causes *Tells Truth Cabot* to come IN). This change in state propagates to *Suspect Cabot,* which goes OUT. Now we have a problem.

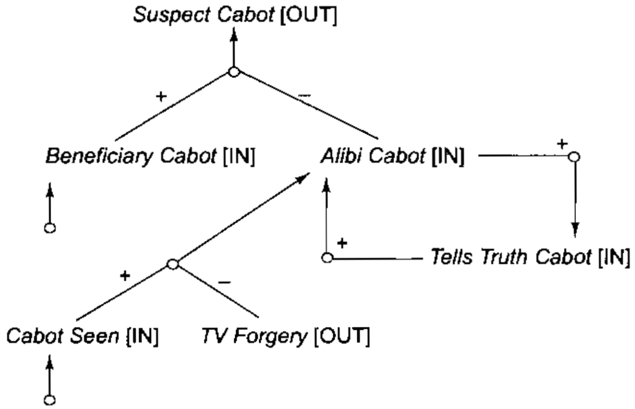


Fig. 7.11 *A Second Justification*

The justification for the contradiction is now valid and the contradiction is IN. The job of the TMS at this point is to determine how the contradiction can be made OUT again. In a TMS network, a node can be made OUT by causing all of its justifications to become invalid. Monotonic justifications cannot be made invalid without retracting explicit assertions that have been made to the network. Nonmonotonic justifications can, however, be invalidated by asserting some fact whose absence is required by the justification. We call assertions with nonmonotonic justifications *assumptions*. An assumption can be retracted by making IN some element of its justification's OUT-list (or recursively in some element of the OUT-list of the justification of some element in its IN-list). Unfortunately, there may be many such assumptions in a large dependency network. Fortunately, the network gives us a way to identify those that are relevant to the contradiction at hand. Dependency-directed backtracking algorithms, of the sort we described in Section 7.5.1, can use the dependency links to determine an AND/OR tree of assumptions that might be retracted and ways to retract them by justifying other beliefs.

In Fig. 7.10, we see that the contradiction itself is an assumption whenever its justification is valid. We might retract it by believing there were other suspects or by finding a way to believe again that either Abbott, Babbitt, or Cabot was a suspect. Each of the last three could be believed if we disbelieved their alibis, which in turn are assumptions. So if we believed that the hotel register was a forgery, that Babbitt's brother-in-law lied, or that the television pictures were faked, we would have a suspect again and the contradiction would go back OUT. So there are four things we might believe to resolve the contradiction. That is as far as DDB will take us. It reports there is an OR tree with four nodes. What should we do?

A TMS has no answer for this question. Early TMSs picked an answer at random. More recent architectures take the more reasonable position that this choice was a problem for the same problem-solving agent that created the dependencies in the first place. But suppose we do pick one. Suppose, in particular, that we choose to believe that Babbitt's brother-in-law lied. What should be the justification for that belief? If we believe it just because not believing it leads to a contradiction, then we should install a justification that should be valid only as long as it needs to be. If later we find another way that the contradiction can be labeled OUT, we will not want to continue in our abductive belief.

For instance, suppose that we believe that the brother-in-law lied, but later we discover that a long-lost relative, jilted by the family, was in town the day of the murder. We would no longer have to believe the brother-in-law lied just to avoid a contradiction. A TMS may also have algorithms to create such justifications, which we call abductive since they are created using abductive reasoning. If they have the property that they are not unnecessarily valid, they are said to be *complete*. Figure 7.12 shows a complete abductive justification for the belief that Babbitt's brother-in-law lied. If we come to believe that Abbott or Cabot is a suspect, or we find a long-lost relative, or we somehow come to believe that Babbitt's brother-in-law didn't really say Babbitt was at his house, then this justification for lying will become invalid.

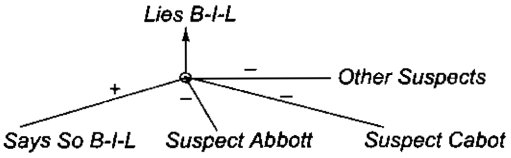


Fig. 7.12 *A Complete Abductive Justification*

At this point, we have described the key reasoning operations that are performed by a JTMS:

- consistent labeling
- contradiction resolution

We have also described a set of important reasoning operations that a JTMS does not perform, including:

- applying rules to derive conclusions
- creating justifications for the results of applying rules (although justifications are created as part of contradiction resolution)
- choosing among alternative ways of resolving a contradiction
- detecting contradictions

All of these operations must be performed by the problem-solving program that is using the JTMS. In the next section, we describe a slightly different kind of TMS, in which, although the first three of these operations must still be performed by the problem-solving system, the last can be performed by the TMS.

7.5.3 Logic-Based Truth Maintenance Systems

A *logic-based truth maintenance system* (LTMS) [McAllester, 1980] is very similar to a JTMS. It differs in one important way. In a JTMS, the nodes in the network are treated as atoms by the TMS, which assumes no relationships among them except the ones that are explicitly stated in the justifications. In particular, a JTMS has no problem simultaneously labeling both P and $\neg P$ IN. For example, we could have represented explicitly both *Lies B-I-L* and *Not Lies B-I-L* and labeled both of them IN. No contradiction will be detected automatically. In an LTMS, on the other hand, a contradiction would be asserted automatically in such a case. If we had constructed the ABC example in an LTMS system, we would not have created an explicit contradiction corresponding to the assertion that there was no suspect. Instead we would (replace the contradiction node by one that asserted something like *No Suspect*. Then we would assert *Suspect*. When *No Suspect* came IN, it would cause a contradiction to be asserted automatically.

7.6 IMPLEMENTATION: BREADTH-FIRST SEARCH

The *assumption-based truth maintenance system* (ATMS) [de Kleer, 1986] is an alternative way of implementing nonmonotonic reasoning. In both JTMS and LTMS systems, a single line of reasoning is pursued at a time, and dependency-directed backtracking occurs whenever it is necessary to change the system's assumptions. In an ATMS, alternative paths are maintained in parallel. Backtracking is avoided at the expense of maintaining multiple contexts, each of which corresponds to a set of consistent as-sumptions. As reasoning proceeds in an ATMS-based system, the universe of consistent contexts is pruned as contradictions are discovered. The remaining consistent contexts are used to label assertions, thus indicating the contexts in which each assertion has a valid justification. Assertions that do not have a valid justification in any consistent context can be pr [illegible] es the set of assertions that can consistently be believed by the problem solver. Essentially, an ATMS system works breadth-first, considering all possible contexts at once, while both JTMS and LTMS systems operate depth-first.

The ATMS, like the JTMS, is designed to be used in conjunction with a separate problem solver. The problem solver's job is to:

- Create nodes that correspond to assertions (both those that are given as axioms and those that are derived by the problem solver).
- Associate with each such node one or more justifications, each of which describes a reasoning chain that led to the node.
- Inform the ATMS of inconsistent contexts.

Notice that this is identical to the role of the problem solver that uses a JTMS, except that no explicit choices among paths to follow need be made as reasoning proceeds. Some decision may be necessary at the end, though, if more than one possible solution still has a consistent context.

The role of the ATMS system is then to:

- Propagate inconsistencies, thus ruling out contexts that include subcontexts (sets of assertions) that are known to be inconsistent.
- Label each problem solver node with the contexts in which it has a valid justification. This is done by combining contexts that correspond to the components of a justification. In particular, given a justification of the form

  $$A1 \wedge A2 \wedge \ldots \wedge An \rightarrow C$$

  assign as a context for the node corresponding to C the intersection of the contexts corresponding to the nodes $A1$ through An.

Contexts get eliminated as a result of the problem-solver asserting inconsistencies and the ATMS propagating them. Nodes get created by the problem-solver to represent possible components of a problem solution. They may then get pruned from consideration if all their context labels get pruned. Thus a choice among possible solution components gradually evolves in a process very much like the constraint satisfaction procedure that we examined in Section 3.5.

One problem with this approach is that given a set of n assumptions, the number of possible contexts that may have to be considered is 2". Fortunately, in many problem-solving scenarios, most of them can be pruned without ever looking at them. Further, the ATMS exploits an efficient labeling system that makes it possible to encode a set of contexts as a single context that delimits the set. To see how both of these things work, it is necessary to think of the set of contexts that are defined by a set of assumptions as forming a lattice, as shown for a simple example with four assumptions in Fig. 7.13. Lines going upward indicate a subset relationship.

The first thing this lattice does for us is to illustrate a simple mechanism by which contradictions (inconsistent contexts) can be propagated so that large parts of the space of 2^n contexts can be eliminated. Suppose that the

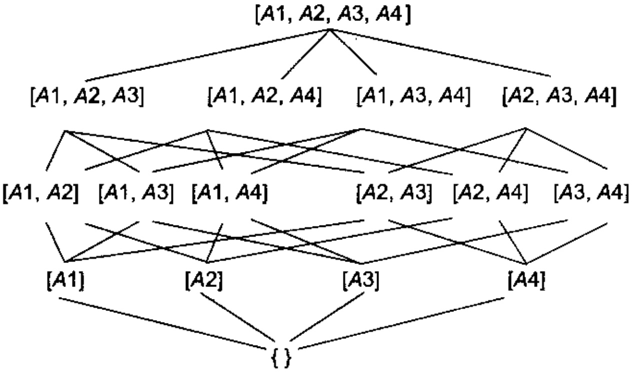


Fig. 7.13 *A Context Lattice*

context labeled $\{A2, A3\}$ is asserted to be Inconsistent. Then all contexts that include it (i.e., those that are above it) must also be inconsistent.

Now consider how a node can be labeled with all the contexts in which it has a valid justification. Suppose its justification depends on assumption $A1$. Then the context labeled $\{A1\}$ and all the contexts that include it are acceptable. But this can be indicated just by saying $\{A1\}$. It is not necessary to enumerate its supersets. In general, each node will be labeled with the greatest lower bounds of the contexts in which it should be believed.

Clearly, it is important that this lattice not be built explicitly but only used as an implicit structure as the ATMS proceeds.

As an example of how an ATMS-based problem-solver works, let's return to the ABC Murder story. Again, our goal is to find a primary suspect. We need (at least) the following assumptions:

- $A1$. Hotel register was forged.
- $A2$. Hotel register was not forged.
- $A3$. Babbitt's brother-in-law lied.
- $A4$. Babbitt's brother-in-law did not lie.
- $A5$. Cabot lied.
- $A6$. Cabot did not lie.
- $A7$. Abbott, Babbitt, and Cabot are the only possible suspects.
- $A8$. Abbott, Babbitt, and Cabot are not the only suspects.

The problem-solver could then generate the nodes and associated justifications shown in the first two columns of Fig. 7.14. In the figure, the justification for a node that corresponds to a decision to make assumption N is shown as $\{N\}$. Justifications for nodes that correspond to the result of applying reasoning rules are shown as the rule involved. Then the ATMS can assign labels to the nodes as shown in the second two columns. The first shows the label that would be generated for each justification taken by itself. The second shows the label (possibly containing multiple contexts) that is actually assigned to the node given all its current justifications. These columns are identical in simple cases, but they may differ in more complex situations as we see for nodes 12, 13, and 14 of our example.

	Nodes	Justifications		Node Labels
[1]	Register was not forged	{A2}	{A2}	{A2}.
[2]	Abbott at hotel	[1] → [2]	{A2}	{A2}
[3]	B-I-L didn't lie	{4}	{A4},	{A4}
[4]	Babbitt at B-I-L	[3] → [4]	{A4}	{A4}
[5]	Cabot didn't lie	{6}	{A6}	{A6}
[6]	Cabot at ski show	[5] → [6]	{A6}	{A6}
[7]	A, B, C only suspects	{A7}	{A7}	{A7}
[8]	Prime Suspect Abbott	[7] ∧ [13] ∧ [14] → [8]	{A7, A4, A6}	{A7, A4, A6}
[9]	Prime Suspect Babbitt	[7] ∧ [12] ∧ [14] → [9]	{A7, A2, A6}	{A7, A2, A6}
[10]	Prime Suspect Cabot	[7] ∧ [12] ∧ [13] → [10]	{A7, A2, A4}	{A7, A2, A4}
[11]	A, B,C not only suspects	{A8}	{A8}	{A8}
[12]	Not prime suspect Abbott	[2] → [12]	{A2}	{A2}, {A8}
		[11] → [12]	{A8}	
		[9] → [12]	{A7, A2, A6}	
		[10] → [12]	{A7, A2, A4}	
[13]	Not prime suspect Babbitt	[4] → [13]	{A4}	{A4}, {A8}
		[11] → [13]	{A8}	
		[8] → [13]	{A7, A4, A6}	
		[10] → [13]	{A7, A4, A2}	
[14]	Not prime suspect Cabot	[6] → [14]	{A6}	{A6}, {A8}
		[11] → [14]	{A8}	
		[8] → [14]	{A7, A4, A6}	
		[9] → [14]	{A7, A2, A6}	

Fig. 7.14 *Nodes and Their Justifications and Labels*

There are several things to notice about this example:

- Nodes may have several justifications if there are several possible reasons for believing them. This is the case for nodes 12, 13, and 14.
- Recall that when we were using a JTMS, a node was labeled IN if it had at least one valid justification. Using an ATMS, a node will end up being labeled with a consistent context if it has at least one justification that can occur in a consistent context.
- The label assignment process is sometimes complicated. We describe it in more detail below.

Suppose that a problem-solving program first created nodes 1 through 14, representing the various dependencies among them without committing to which of them it currently believes. It can indicate known contradictions by marking as no good the context:

- *A, B, C* are the only suspects; *A, B, C* are not the only suspects: {*A*7, *A*8}

The ATMS would then assign the labels shown in the figure. Let's consider the case of node 12. We generate four possible labels, one for each justification. But we want to assign to the node a label that contains just the greatest lower bounds of all the contexts in which it can,occur, since they implicitly encode the superset contexts. The label {*A*2} is the greatest lower bound of the first, third, and fourth label, and {*A*8} is the same for the second label. Thus those two contexts are all that are required as the label for the node. Now let's consider labeling node 8. Its label must be the union of the labels of nodes 7, 13, and 14. But nodes 13 and 14 have complex labels representing alternative justifications. So we must consider all ways of combining the labels of all three nodes. Fortunately, some of these combinations, namely those that contain both A7 and A8, can be eliminated because they are already known to be contradictory. Thus we are left with a single label as shown.

Now suppose the problem-solving program labels the context {*A*2} as no good, meaning that the assumption it contains (namely that the hotel register was not forged) conflicts with what it knows. Then many of the labels that we had disappear since they are now inconsistent. In particular, the labels for nodes 1, 2, 9, 10, and 12 disappear. At this point, the only suspect node that has a label is node 8. But node 12 (Not prime suspect Abbott) also still has a label that corresponds to the assumption that Abbott, Babbitt, and Cabot are not the only suspects. If this assumption is made, then Abbott would: not be a clear suspect even if the hotel register were forged. Further information or some choice process is still necessary to choose between these remaining nodes.

SUMMARY

In this chapter we have discussed several logical systems that provide a basis for nonmonotonic reasoning, including nonmonotonic logic, default logic, abduction, inheritance, the closed world assumption, and circumscription. We have also described a way in which the kind of rules that we discussed in Chapter 6 could be augmented to support nonmonotonic reasoning.

We then presented three kinds of TMS systems, all of which provide a basis for implementing nonmonotonic reasoning. We have considered two dimensions along which TMS systems can vary: whether they automatically detect logical contradictions and whether they maintain single or multiple contexts. The following table summarizes this discussion:

TMS Kinds	*single context*	*multiplecontext*
nonlogical	JTMS	ATMS
logical	LTMS	?

As can be seen in this table, there is currently no TMS with logical contradictions and multiple contexts.

These various TMS systems each have advantages and disadvantages with respect to each other. The major issues that distinguish JTMS and ATMS systems are:

- The JTMS is often better when only a single solution is desired since it does not need to consider alternatives; the ATMS is usually more efficient if all solutions are eventually going to be needed.
- To create the context lattice, the ATMS performs a global operation in which it considers all possible combinations of assumptions. As a result, either all assumptions must be known at the outset of problem solving or an expensive, recompilation process must occur whenever an assumption is added. In the JTMS, on the other hand, the gradual addition, of new assumptions poses no problem.
- The JTMS may spend a lot of time switching contexts when backtracking is necessary. Context switching does not happen in the ATMS.
- In an ATMS, inconsistent contexts disappear from consideration. If the initial problem description was overconstrained, then all nodes will end up with empty labels and there will be no problem-solving trace that can serve as a basis for relaxing one or more of the constraints. In a JTMS, on the other hand, the justification that is attached to a contradiction node provides exactly such a trace.
- The ATMS provides a natural way to answer questions of the form, "In what contexts is A true?" The only way to answer such questions using a JTMS is to try all the alternatives and record the ones in which A is labeled IN.

One way to get the best of both of these worlds is to combine an ATMS and a JTMS (or LTMS), letting each handle the part of the problem-solving process to which it is best suited.

The various nonmonotonic systems that we have described in this chapter have served as a basis for a variety of applications. One area of particular significance is diagnosis (for example, of faults in a physical device) [Reiter, 1987b; de Kleer and Williams, 1987]. Diagnosis is a natural application area for minimalist reasoning in particular, since one way to describe the diagnostic task is, "Find the smalles': set of abnormally behaving components that would account for the observed behavior." A second application area is reasoning about action, with a particular emphasis on addressing-the frame problem [Hanks and McDermott, 1986]. The frame problem is also natural for this kind of reasoning since it can be described as, "Assume that everything stays the same after an action except the things that necessarily change." A third application area is design [Steele *et al.*, 1989]. Here, nonmonotonic reasoning provides a basis for using common design principles to find a promising path quickly even in a huge design space while preserving the option to consider alternatives later if necessary. And yet another application area is in extracting intent from English expressions (see Chapter 15.)

In all the systems that we have discussed, we have assumed that belief status is a binary function. An assertion must eventually be either believed or not. Sometimes, this is too strong an assumption. In the next chapter, we present techniques for dealing with uncertainty without making that assumption. Instead, we allow for varying degrees of belief.

EXERCISES

1. Try to formulate the ABC Murder story in preduate logic and see how far you can get.
2. The classic example of nonmonotonic reasoning involves birds and flying. In particular, consider the following facts:
   - Most things do not fly.
   - Most birds do fly, unless they are too young or dead or have a broken wing.
   - Penguins and ostriches do not fly.
   - Magical ostriches fly.

- Tweety is a bird.
- Chirpy is either a penguin or an ostrich.
- Feathers is a magical ostrich.

Use one or more of the nonmonotonic reasoning systems we have discussed to answer the following questions:

- Does Tweety fly?
- Does Chirpy fly?
- Does Feathers fly?
- Does Paul fly?

3. Consider the missionaries and cannibals problem of Section 2.6. When you solved that problem, you used the CWA several times (probably without thinking about it). List some of the ways in which you used it.
4. A big technical problem that arises in defining circumscription precisely is the definition of a minimal model. Consider again the problem of Dick, the Quaker and Republican, which we can rewrite using a slightly different kind of *AB* predicate as:

$\forall x : Republican(x) \wedge \neg \mathrm{AB1}(x) \rightarrow \neg Pacifist(x)$
$\forall x : Quaker(x) \wedge \neg AB2(x) \rightarrow Pacifist(x)$
$Republican(x)$
$Quaker(x)$

(*a*) Write down the smallest models you can that describe the two extensions that we computed for that knowledge base.
(*b*) Does it make sense to say that either is smaller than the other?
(*c*) Prioritized circumscription [McCarthy, 1986] attempts to solve this problem by ranking predicates by the order in which they should be minimized. How could you use this idea to indicate a preference as to which extension to prefer?

5. Consider the problem of finding clothes to wear in the morning. To solve this problem, it is necessary to use knowledge such as:
- Wear jeans unless either they are dirty or you have a job interview today.
- Wear a sweater if it's cold.
- It's usually cold in the winter.
- Wear sandals if it's warm.
- It's usually warm in the summer.

(*a*) Build a JTMS-style database of the necessary facts to solve this problem.
(*b*) Show how the problem can be solved and how the solution changes as the relevant facts (such as time of year and dirtiness of jeans) change.

6. Show how a JTMS could be used in medical diagnosis. Consider rules such as, "If you have a runny nose, assume you have a cold unless it is allergy season."
7. Solve the same medical reasoning problem with an ATMS.
8. Show how a JTMS could be used to select a TV program to watch. Consider rules such as, "If it is 6:00, then watch the news on channel 2 unless there is a football game still going on."
9. TMSs are useful tools in solving constraint satisfaction problems since they fa cilitate the nonmonotonic reasoning that occurs during the search for a complete solution.

(*a*) Show how a JTMS could be used to solve the cryptarithmetic problems of Chapter 2.
(*b*) Show how an ATMS would solve the same problem.

10. We described informally the JTMS labeling process. Write a formal description of that algorithm.
11. Work through the details of the ATMS node labeling process whose results are shown in Fig. 7.14.

CHAPTER 8

STATISTICAL REASONING

Statistics can be made to prove anything-even the truth.

—Anonymous

So far, we have described several representation techniques that can be used to model belief systems in which, at any given point, a particular fact is believed to be true, believed to be false, or not considered one way or the other. For some kinds of problem solving, though, it is useful to be able to describe beliefs that are not certain but for which there is some supporting evidence. Let's consider two classes of such problems.

The first class contains problems in which there is genuine randomness in the world. Playing card games such as bridge and blackjack is a good example of this class. Although in these problems it is not possible to predict the world with certainty, some knowledge about the likelihood of various outcomes is available, and we would like to be able to exploit it.

The second class contains problems that could, in principle, be modeled using the techniques we described in the last chapter. In these problems, the relevant world is not random; it behaves "normally" unless there is some kind of exception. The difficulty is that there are many more possible exceptions than we care to enumerate explicitly (using techniques such as *AB* and UNLESS). Many common sense tasks fall into this category, as do many expert reasoning tasks such as medical diagnosis. For problems like this, statistical measures may serve a very useful function as summaries of the world; rather than enumerating all the possible exceptions, we can use a numerical summary that tells us how often an exception of some sort can be expected to occur.

In this chapter we explore several techniques that can be used to augment knowledge representation techniques with statistical measures that describe levels of evidence and belief.

8.1 PROBABILITY AND BAYES' THEOREM

An important goal for many problem-solving systems is to collect evidence as the system goes along and to modify its behavior on the basis of the evidence. To model this behavior, we need a statistical theory of evidence. Bayesian statistics is such a theory. The fundamental notion of Bayesian statistics is that of conditional probability:

$$P(H\backslash E)$$

Read this expression as the probability of hypothesis H given that we have observed evidence E. To compute this, we need to take into account the prior probability of H (the probability that we would assign to H if we had no evidence) and the extent to which E provides evidence of H. To do this, we need to define a universe that contains an exhaustive, mutually exclusive set of H_i's, among which we are trying to discriminate. Then, let

$P(H_i\backslash E)$ = the probability that hypothesis H_i is true given evidence E
$P(E\backslash H_i)$ = the probability that we will observe evidence E given that hypothesis i is true
$P(H_i)$ = the *a priori* probability that hypothesis i is true in the absence of any specific evidence. These probabilities are called prior probabilities or *prlors.*
k = the number of possible hypotheses

Bayes' theorem then states that

$$P(H_i\backslash E) = \frac{P(E \mid H_i) \cdot P(H_i)}{\sum_{n=1}^{k} P(E \mid H_n) \cdot P(H_n)}$$

Suppose, for example, that we are interested in examining the geological evidence at a particular location to determine whether that would be a good place to dig to find a desired mineral. If we know the prior probabilities of finding each of the various minerals and we know the probabilities that if a mineral is present then certain physical characteristics will be observed, then we can use Bayes' formula to compute, from the evidence we collect, how likely it is that the various minerals are present. This is, in fact, what is done by the PROSPECTOR program [Duda *et al.,* 1979], which has been used successfully to help locate deposits of several minerals, including copper and uranium.

The key to using Bayes' theorem as a basis for uncertain reasoning is to recognize exactly what it says. Specifically, when we say $P(A\backslash B)$, we are describing the conditional probability of A given that the only evidence we have is B. If there is also other relevant evidence, then it too must be considered. Suppose, for example, that we are solving a medical diagnosis problem. Consider the following assertions:

S: patient has spots
M: patient has measles
F. patient has high fever

Without any additional evidence, the presence of spots serves as evidence in favor of measles. It also serves as evidence of fever since measles would cause fever. But suppose we already know that the patient has measles. Then the additional evidence that he has spots actually tells us nothing about the likelihood of fever. Alternatively, either spots alone or fever alone would constitute evidence in favor of measles. If both are present, we need to take both into account in determining the total weight of evidence. But, since spots and fever are not independent events, we cannot just sum their effects. Instead, we need to represent explicitly the conditional probability that arises from their conjunction. In general, given a prior body of evidence e and some new observation E, we need to compute

$$P(H\backslash E, e) = P(H \mid E) \cdot \frac{P(e \mid E, H)}{P(e \mid E)}$$

Unfortunately, in an arbitrarily complex world, the size of the set of joint probabilities that we require in order to compute this function grows as 2^n if there are n different propositions being considered. This makes using Bayes' theorem intractable for several reasons:

- The knowledge acquisition problem is insurmountable: too many probabilities have to be provided. In addition, there is substantial empirical evidence (e.g., Tversky and Kahneman [1974] and Kahneman *et al.* [1982]) that people are very poor probability estimators.
- The space that would be required to store all the probabilities is too large.
- The time required to compute the probabilities is too large.

Despite these problems, though, Bayesian statistics provide an attractive basis for an uncertain reasoning system. As a result, several mechanisms for exploiting its power while at the same time making it tractable have been developed. In the rest of this chapter, we explore three of these:

- Attaching certainty factors to rules
- Bayesian networks
- Dempster-Shafer theory

We also mention one very different numerical approach to uncertainty, fuzzy logic.

There has been an active, strident debate for many years on the question of whether pure Bayesian statistics are adequate as a basis for the development of reasoning programs. (See, for example, Cheeseman [1985] for arguments that it is and Buchanan and Shortliffe [1984] for arguments that it is not.) On the one hand, non-Bayesian approaches have been shown to work well for some kinds of applications (as we see below). On the other hand, there are clear limitations to all known techniques. In essence, the jury is still out. So we sidestep the issue as much as possible and simply describe a set of methods and their characteristics.

8.2 CERTAINTY FACTORS AND RULE-BASED SYSTEMS

In this section we describe one practical way of compromising on a pure Bayesian system. The approach we discuss was pioneered in the MYCIN system [Shortliffe, 1976; Buchanan and Shortliffe, 1984; Shortliffe and Buchanan, 1975], which attempts to recommend appropriate therapies for patients with bacterial infections. It interacts with the physician to acquire the clinical data it needs. MYCIN is an example of an *expert system,* since it performs a task normally done by a human expert. Here we concentrate on the use of probabilistic reasoning; Chapter 20 provides a broader view of expert systems.

MYCIN represents most of its diagnostic knowledge as a set of rules. Each rule has associated with it a *certainty factor,* which is a measure of the extent to which the evidence that is described by the antecedent of the rule supports the conclusion that is given in the rule's consequent. A typical MYCIN rule looks like:

```
If:   (1)  the stain of the organism is gram-positive, and
      (2)  the morphology of the organism is coccus, and
      (3)  the growth conformation of the organism is clumps,
           then there is suggestive evidence (0.7) that
           the identity of the organism is staphylococcus.
```

This is the form in which the rules are stated to the user. They are actually represented internally in an easy-to-manipulate LISP list structure. The rule we just saw would be represented internally as

```
PREMISE:   ($AND    (SAME CNTXT GRAM GRAMPOS)
                    (SAME CNTXT MORPH COCCUS)
                    (SAME CNTXT CONFORM CLUMPS))
ACTION:             (CONCLUDE CNTXT IDENT STAPHYLOCOCCUS TALLY 0.7)
```

MYCIN uses these rules to reason backward to the clinical data available from its goal of finding significant disease-causing organisms. Once it finds the identities of such organisms, it then attempts to select a therapy by which the disease (s) may be treated. In order to understand how MYCIN exploits uncertain information, we need answers to two questions: "What do certainty factors mean?" and "How does MYCIN combine the estimates of certainty in each of its rules to produce a final estimate of the certainty of its conclusions?" A further question that we need to answer, given our observations about the intractability of pure Bayesian reasoning, is, "What compromises does the MYCIN technique make and what risks are associated with those compromises?" In the rest of this section we answer all these questions.

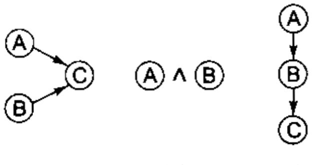
Let's start first with a simple answer to the first question (to which we return with a more detailed answer later). A certainty factor ($CF[h, e]$) is defined in terms of two components:

- $MB[h, e]$—a measure (between 0 and 1) of belief in hypothesis h given the evidence e. *MB* measures the extent to which the evidence supports the hypothesis. It is zero if the evidence fails to support the hypothesis.
- $MD[h,e]$—a measure (between 0 and 1) of disbelief in hypothesis h given the evidence e. *MD* measures the extent to which the evidence supports the negation of the hypothesis. It is zero if the evidence supports the hypothesis.

From these two measures, we can define the certainty factor as

$$CF[h, e] = MB[h, e] - MD[h, e]$$

Since any particular piece of evidence either supports or denies a hypothesis (but not both), and since each MYCIN rule corresponds to one piece of evidence (although it may be a compound piece of evidence), a single number suffices for each rule to define both the *MB* and *MD* and thus the *CF*.



(a) (b) (c)

Fig. 8.1 *Combining Uncertain Rules*

The *CF's* of MYCIN's rules are provided by the experts who write the rules. They reflect the experts' assessments of the strength of the evidence in support of the hypothesis. As MYCIN reasons, however, these *CF's* need to be combined to reflect the operation of multiple pieces of evidence and multiple rules applied to a problem. Figure 8.1 illustrates three combination scenarios that we need to consider. In Fig. 8.1(a), several rules all provide evidence that relates to a single hypothesis. In Fig. 8.1(b), we need to consider our belief in a collection of several propositions taken together. In Fig. 8.1(c), the output of one rule provides the input to another.

What formulas should be used to perform these combinations? Before we answer that question, we need first to describe some properties that we would like th combining functions to satisfy:

- Since the order in which evidence is collected is arbitrary, the combining functions should be commutative and associative.
- Until certainty is reached, additional confirming evidence should increase *MB* (and similarly for disconfirming evidence and *MD*).
- If uncertain inferences are chained together, then the result should be less certain than either of the inferences alone.

Having accepted the desirability of these properties, let's first consider the scenario in Fig. 8.1(a), in which several pieces of evidence are combined to determine the *CF* of one hypothesis. The measures of belief and disbelief of a hypothesis given two observations s_1 and s_2 are computed from:

$$MB[h,s_1 \wedge s_2] = \begin{cases} 0 & \text{if } MD[h, s_1 \wedge s_2] = 1 \\ MB[h, s_1] + MB[h, s_2] \cdot (1 - MB[h, s_1]) & \text{otherwise} \end{cases}$$

$$MD[h, s_1 \wedge s_2] = \begin{cases} 0 & \text{if } MB[h, s_1 \wedge s_2] = 1 \\ MD[h, s_1] + MD[h, s_2] \cdot (1 - MD[h, s_1]) & \text{otherwise} \end{cases}$$

One way to state these formulas in English is that the measure of belief in h is 0 if h is disbelieved with certainty. Otherwise, the measure of belief in h given two observations is the measure of belief given only one observation plus some increment for the second observation. This increment is computed by first taking the difference between 1 (certainty) and the belief given only the first observation. This difference is the most that can be added by the second ohservation. The difference is then scaled by the belief in h given only the second observation. A corresponding explanation can be given, then, for the formula for computing disbelief. From MB and MD, CF can be computed. Notice that if several sources of corroborating evidence are pooled, the absolute value of CF will increase. If conflicting evidence is introduced, the absolute value of CF will decrease.

A simple example shows how these functions operate. Suppose we make an initial observation that confirms our belief in h with $MB = 0.3$. Then $MD[h,s_1] = 0$ and $CF[h, s_1] = 0.3$. Now we make a second observation, which also confirms h, with $MB[h,s_2] = 0.2$. Now:

$$\begin{aligned} MB[h,s_1 \wedge s_2] &= 0.3 + 0.2 \cdot 0.7 \\ &= 0.44 \\ MD[h,s_1 \wedge s_2] &= 0.0 \\ CF[h,s_1 \wedge s_2] &= 0.44 \end{aligned}$$

You can see from this example how slight confirmatory evidence can accumulate to produce increasingly larger certainty factors.

Next let's consider the scenario of Fig. 8.1(b), in which we need to compute the certainty factor of a combination of hypotheses. In particular, this is necessary when we need to know the certainty factor of a rule antecedent that contains several clauses (as, for example, in the staphylococcus rule given above). The combination certainty factor can be computed from its MB and MD. The formulas MYCIN uses for the MB of the conjunction and the disjunction of two hypotheses are:

$$MB[h_1 \wedge h_2, e] = \min(MB[h_1, e], MB[h_2, e])$$

$$MB[h_1 \wedge h_2, e] = max(MB[h_1,e], MB[h_2,e])$$

MD can be computed analogously.

Finally, we need to consider the scenario in Fig. 8.1(c), in which rules are chained together with the result that the uncertain outcome of one rule must provide the input to another. Our solution to this problem will also handle the case in which we must assign a measure of uncertainty to initial inputs. This could easily happen in situations where the evidence is the outcome of an experiment or a laboratory test whose results are not completely accurate. In such a case, the certainty factor of the hypothesis must take into account both the strength with which the evidence suggests the hypothesis and the level of confidence in the evidence. MYCIN provides a chaining rule that is defined as follows. Let $MB'[h, s]$ be the measure of belief in h given that we are absolutely sure of the validity of s. Let e be the evidence that led us to believe in s (for example, the actual readings of the laboratory instruments or the results of applying other rules). Then:

$$MB[h, e] = MB'[h, e] \cdot \max(0, CF[s, e])$$

Since initial *CF*'s in MYCIN are estimates that are given by experts who write the rules, it is not really necessary to state a more precise definition of what a *CF* means than the one we have already given. The original work did, however, provide one by defining *MB* (which can be thought of as a proportionate decrease in disbelief in *h* as a result of *e*) as:

$$MB[h, e] = \begin{cases} 1 & \text{if } P(h) = 1 \\ \dfrac{\max[P(h \mid e), P(h)] - P(h)}{1 - P(h)} & \text{otherwise} \end{cases}$$

Similarly, the *MD* is the proportionate decrease in belief in *h* as a result of *e*:

$$MD[h, e] = \begin{cases} 1 & \text{if } P(h) = 0 \\ \dfrac{\min[P(h \mid e), P(h)] - P(h)}{-P(h)} & \text{otherwise} \end{cases}$$

It turns out that these definitions are incompatible with a Bayesian view of conditional probability. Small cbanges to them, however, make them compatible [Heckerman, 1986]. In particular, we can redefine *MB* as

$$MB[h, e] = \begin{cases} 1 & \text{if } P(h) = 1 \\ \dfrac{\max[P(h \mid e), P(h)] - P(h)}{(1 - P(h) \cdot P(h \mid e)} & \text{otherwise} \end{cases}$$

The definition of *MD* must also be changed similarly.

With these reinterpretations, there ceases to be any fundamental conflict between MYCIN's techniques and those suggested by Bayesian statistics. We argued at the end of the last section that pure Bayesian statistics usually leads to intractable systems. But MYCIN works [Buchanan and Shortliffe, 1984]. Why?

Each *CF* in a MYCIN rule represents the contribution of an individual rule to MYCIN's belief in a hypothesis. In some sense then, it represents a conditional probability, *P*(*H**E*). But recall that in a pure Bayesian system, *P*(*H**E*) describes the conditional probability of *H* given that the only relevant evidence is *E*. If there is other evidence, joint probabilities need to be considered. This is where MYCIN diverges from a pure Bayesian system, with the result that it is easier to write and more efficient to execute, but witb the corresponding risk, that its behavior will be counterintuitive. In particular, the MYCIN formulas for all three combination scenarios of Fig. 8.1 make the assumption that all rules are independent. The burden of guaranteeing independence (at least to the extent that it matters) is on the rule writer. Each of the combination scenarios is vulnerable when this independence assumption is violated.

Let's first consider the scenario in Fig. 8.1(a). Our example rule has three antecedents with a single *CF* rather than three separate rules; this makes the combination rules unnecessary. The rule writer did this because the three antecedents are not independent. To see how much difference MYCIN's independence assumption can make, suppose for a moment that we had instead had three separate rules and that the *CF* of each was 0.6. This could happen and still be consistent with the combined *CF* of 0.7 if the three conditions overlap substantially. If we apply the MYCIN combination formula to the three separate rules, we get

$$\begin{aligned} MB[h,s \wedge s_2] &= 0.6 + (0.6 \cdot 0.4) \\ &= 0.84 \\ MB[h,(s_1 \wedge s_2) \wedge s_3] &= 0.84 + (0.6 \cdot 0.16) \\ &= 0.936 \end{aligned}$$

This is a substantially different result than the true value, as expressed by the expert, of 0.7.

Now let's consider what happens when independence assumptions are violated in the scenario of Fig. 8.1(c). Let's consider a concrete example in which:

S: sprinkler was on last night
W: grass is wet
R: it rained last night

We can write MYCIN-style rules that describe predictive relationships among these three events:

```
If:  the sprinkler was on last night
then there is suggestive evidence (0.9) that
     the grass will be wet this morning
```

Taken alone, this rule may accurately describe the world. But now consider a second rule:

```
If:  the grass is wet this morning
then there is suggestive evidence (0.8) that
     it rained last night
```

Taken alone, this rule makes sense when rain is the most common source of water on the grass. But if the two rules are applied together, using MYCIN's rule for chaining, we get

$MB[W,S] = 0.8$	{sprinkler suggests wet}
$MB[R,W] = 0.8 \cdot 0.9 = 0.72$	{wet suggests rains}

In other words, we believe that it rained because we believe the sprinkler was on. We get this despite the fact that if the sprinkler is known to have been on and to be the cause of the grass being wet, then there is actually almost no evidence for rain (because the wet grass has been explained some other way). One of the major advantages of the, modularity of the MYCIN rule system is that it allows us to consider individual antecedent/consequent relationships independently of others. In particular, it lets us talk about the implications of a proposition without going back and considering the evidence that supported it. Unfortunately, this example shows that there is a danger in this approach whenever the justifications of a belief are important to determining its consequences. In this case, we need to know why we believe the grass is wet (e.g., because we observed it to be wet as opposed to because we know the sprinkler was on) in order to determine whether the wet grass is evidence for it having just rained.

It is worth pointing out here that this example illustrates one specific rule structure that almost always causes trouble and should be avoided. Notice that our first rule describes a causal relationship (sprinkler causes wet grass). The second rule, although it looks the same, actually describes an inverse causality relationship (wet grass is caused by rain and thus is evidence for its cause). Although one can derive evidence for a symptom from its cause and for a cause from observing its symptom, it is important that evidence that is derived one way not be used again to go back the other way with no new information. To avoid this problem, many rule-based systems either limit their rules to one structure or clearly partition the two kinds so that they cannot interfere with each other. When we discuss Bayesian networks in the next section, we describe a systematic solution to this problem.

We can summarize this discussion of certainty factors and rule-based systems as follows. The approach makes strong independence assumptions that make it relatively easy to use; at the same time assumptions create dangers if rules are not written carefully so that important dependencies are captured. The approach can serve as the basis of practical application programs. It did so in MYCIN. It has done so in a broad array of other systems that have been built on the EMYCIN platform [van Melle *et al.*, 1981], which is a generalization (often called a *shell*) of MYCIN with all the domain-specifie rules stripped out. One reason that this framework is useful, despite its limitations, is that it appears that in an otherwise robust system the exact numbers that are used do not matter very much. The other reason is that the rules were carefully designed to avoid the major pitfalls we have just described. One other interesting thing about this approach is that it appears to mimic quite well [Shultz *et al.*, 1989] the way people manipulate certainties.

8.3 BAYESIAN NETWORKS

In the last section, we described *CF's* as a mechanism for reducing the complexity of a Bayesian reasoning system by making some approximations to the formalism. In this section, we describe an alternative approach, *Bayesian networks* [Pearl, 1988], in which we preserve the formalism and rely instead on the modularity of the world we are trying to model. The main idea is that to describe the real world, it is not necessary to use a huge joint probabililify table in which we list the probabilities of all conceivable combinations of events. Most events are conditionally independent of most other ones, so their interactions need not be considered. Instead, we can use a more local representation in which we will describe clusters of events that interact.

Recall that in Fig. 8.1 we used a network notation to describe the various kinds of constraints on likelihoods that propositions can have on each other. The idea of constraint networks turns out to be very powerful. We expand on it in this section as a way to represent interactions among events; we also return to it later in Sections 11.3.1 and 14.3, where we talk about other ways of representing knowledge as sets of constraints.

Let's return to the example of the sprinkler, rain, and grass that we introduced in the last section. Figure 8.2(a) shows the flow of constraints we described in MYCIN-style rules. But recall that the problem that we encountered with that example was that the constraints flowed incorrectly from "sprinkler on" to "rained last night." The problem was that we failed to make a distinction that turned out to be critical. There are two different ways that propositions can influence the likelihood of each other. The first is that causes influence the likelihood of their symptoms; the second is that observing a symptom affects the likelihood of all of its possible causes. The idea behind the Bayesian network structure is to make a clear distinction between these two kinds of influence.

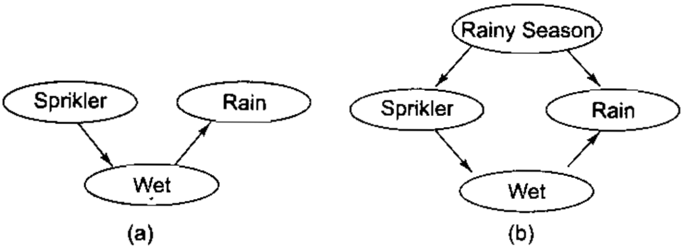


Fig. 8.2 *Representing Causality Uniformly*

Specifically, we construct a directed acyclic graph (DAG) that represents causality relationships among variables. The idea of a causality graph (or network) has proved to be very useful in several systems, particularly medical diagnosis systems such as CAS- NET [Weiss *et al.*, 1978] and INTERNIST/CADUCEUS [Pople, 1982]. The variables in such a graph may be propositional (in which case they can take on the values TRUE and FALSE) or they may be variables that take on values of some other type (e.g., a specific disease, a body temperature, or a reading taken by some other diagnostic device). In Fig. 8.2(*b*), we show a causality graph for the wet grass example. In addition to the three nodes we have been talking about, the graph contains a new node corresponding to the propositional variable that tells us whether it is currently the rainy season.

A DAG, such as the one we have just drawn, illustrates the causality relationships that occur among the nodes it contains. In order to use it as a basis for probabilistic reasoning, however, we need more information. In particular, we need to know, for each value of a parent node, what evidence is provided about the values that the child node can take on. We can state this in a table in which the conditional probabilities are provided. We show such a table for our example in Fig. 8.3. For example. from the table we see that the prior probability of the rainy season is 0.5. "Then, if it is the rainy season, the probability of rain on a given night is 0.9; if it is not, the probability is only 0.1.

Attribute	*Probability*
p(*Wet\Sprinkler, Rain*)	0.95
P(*Wet\Sprinkler,* ¬*Rain*)	0.9
p(*Wet\¬Sprinkler, Rain*)	0.8
p(*Wet\¬Sprinkler,* ¬*Rain*)	0.1
p(*Sprinkler\RainySeason*)	0.0
p(*Sprinkler\¬RainySeason*)	1.0
p(*Rain \RainySeason*)	0.9
p(*Rain \ ¬RainySeason*)	0.1
p(*RainySeason*)	0.5

Fig. 8.3 *Conditional Probabilities for a Bayesian Network*

To be useful as a basis for problem solving, we need a mechanism for computing the influence of any arbitrary node on any other. For example, suppose that we have observed that it rained last night. What does that tell us about the probability that it is the rainy season? To answer this question requires that the initial DAG be converted to an undirected graph in which the arcs can be used to transmit probabilities in either direction, depending on where the evidence is coming from. We also require a mechanism for using the graph that guarantees that probabilities are transmitted correctly. For example, while it is true that observing wet grass may be evidence for rain, and observing rain is evidence for wet grass, we must guarantee that no cycle is ever traversed in such a way that wet grass is evidence for rain, which is then taken as evidence for wet grass, and so forth.

There are three broad classes of algorithms for doing these computations: a message-passing method [Pearl, 1988], a clique triangulation method [Lauritzen and Spiegelhalter, 1988], and a variety of stochastic algorithms. The idea behind these methods is to take advantage of the fact that nodes have limited domains of influence. Thus, although in principle the task of updating probabilities consistently throughout the network is intractable, in practice it may not be. In the clique triangulation method, for example, explicit arcs are introduced between pairs of nodes that share a common- descendent. For the case shown in Fig. 8.2(b), a link would be introduced between *Sprinkler* and *Rain.* This explicit link supports assessing the impact of the observation *Sprinkler* on the hypothesis *Rain.* This is important since wet grass could be evidence of either of them, but wet grass plus one of its causes is not evidence for the competing cause since an alternative explanation for the observed phenomenon already exists.

The message-passing approach is based on the observation that to compute the probability of a node A given what is known about other nodes in the network, it is necessary to know three things:

- π-the total support arriving at A from its parent nodes (which represent its causes).
- λ-the total support arriving at A from its children (which represent its symptoms).
- The entry in the fixed conditional probability matrix that relates A to its causes.

Several methods for propagating π and λ messages and updating the probabilities at the nodes have been developed. The structure of the network determines what approach can be used. For example, in singly connected networks (those in which there is only a single path between every pair of nodes), a simpler algorithm can be used than in the case of multiply connected ones. For details, see Pearl [1988].

Finally, there are stochastic, or randomized algorithms for updating belief networks. One such algorithm [Chavez, 1989] transforms an arbitrary network into a Markov chain. The idea is to shield a given node probabilistically from most of the other nodes in the network: Stochastic algorithms run fast in practice, but may not yield absolutely correct results.

8.4 DEMPSTER-SHAFER THEORY

So far, we have described several techniques, all of which consider individual propositions and assign to each of them a point estimate (i.e., a single number) of the degree of belief that is warranted given the evidence. In this section, we consider an alternative technique, called *Dempster-Shafer theory* [Dempster, 1968; Shafer, 1976]. This new approach considers sets of propositions and assigns to each of them an interval

$$[Belief,\ Plausibility]$$

in which the degree of belief must lie. Belief (usually denoted *Bel*) measures the strength of the evidence in favor of a set of propositions. It ranges from 0 (indicating no evidence) to 1 (denoting certainty).

Plausibility (*Pl*) is denned to be

$$Pl(s) = 1 - Bel(\neg s)$$

It also ranges from 0 to 1 and measures the extent to which evidence in favor of $\neg s$ leaves room for belief in s. In particular, if we have certain evidence in favor of $\neg s$, then $Bel(\neg s)$ will be 1 and $Pl(s)$ will be 0. This tells us that the only possible value for $Bel(s)$ is also 0.

The belief-plausibility interval we have just defined measures not only our level of belief in some propositions, but also the amount of information we have. Suppose that we are currently considering three competing hypotheses: *A, B,* and *C*. If we have no information, we represent that by saying, for each of them, that the true likelihood is in the range [0,1]. As evidence is accumulated, this interval can be expected to shrink, representing increased confidence that we know how likely each hypothesis is. Note that this contrasts with a pure Bayesian approach, in which we would probably begin by distributing the prior probability equally among the hypotheses and thus assert for each that $P(h) = 0.33$. The interval approach makes it clear that we have no information when we start. The Bayesian approach does not, since we could end up with the same probability values if we collected volumes of evidence, which taken together suggest that the three values occur equally often. This difference can matter if one of the decisions that our program needs to make is whether to collect more evidence or to act on the basis of the evidence it already has.

So far we have talked intuitively about *Bel* as a measure of our belief in some „ hypothesis given some evidence. Let's now define it more precisely. To do this, we need to start, just as with Bayes' theorem, with an exhaustive universe of mutually exclusive hypotheses. We'll call this *the frame of discernment* and we'll write it as Θ. For example, in a simplified diagnosis problem, Θ might consist of the set $\{All, Flu, Cold, Pneu\}$:

All: allergy
Flu: flu
Cold: cold
Pneu: pneumonia

Our goal is to attach some measure of belief to elements of Θ. However, not all evidence is directly supportive of individual elements. Often it supports sets of elements (i.e., subsets of Θ). For example, in our diagnosis problem, fever might support {*Flu, Cold, Pneu*}. In addition, since the elements of Θ are mutually exclusive, evidence in favor of some may have an affect on our belief in the others. In a purely Bayesian system, we can handle both of these phenomena by listing all of the combinations of conditional probabilities. But our goal is not to have to do that. Dempster-Shafer theory lets us handle interactions by manipulating sets of hypotheses directly.

The key function we use is a probability density function, which we denote as m. The function m is defined not just for elements of Θ but for all subsets of it (including singleton subsets, which correspond to individual elements). The quantity $m(p)$ measures the amount of belief that is currently assigned to exactly the set p of hypotheses. If Θ contains n elements, then there are 2^n subsets of Θ. We must assign m so that the sum of all the m values assigned to the subsets of Θ is 1. Although dealing with 2^n values may appear intractable, it Usually turns out that many of the subsets will never need to be considered because they have no significance in the problem domain (and so their associated value of m will be 0).

Let us see how m works for our diagnosis problem. Assume that we have no information about how to choose among the four hypotheses when we start the diagnosis task. Then we define m as:

$\{\Theta\}$	(1.0)

All other values of m are thus 0. Although this means that the actual value must be some one element *All, Flu, Cold,* or *Pneu,* we do not have any information that allows us to assign belief in any other way than to say that we are sure the answer is somewhere in the whole set. Now suppose we acquire a piece of evidence that suggests (at a level of 0.6) that the correct diagnosis is in the set {*Flu, Cold,Pneu*}. Fever might be such a piece of evidence. We update m as follows:

{*Flu,Cold,Pneu*}	(0.6)
$\{\Theta\}$	(0.4)

At this point, we have assigned to the set {*Flu, Cold,Pneu*} the appropriate belief. The remainder of our belief still resides in the larger set Θ. Notice that we do not make the commitment that the remainder must be assigned to the complement of {*Flu,Cold, Pneu*}.

Having defined m, we can now define *Bel(p)* for a set p as the sum of the values of m for p and for all of its subsets. Thus *Bel(p)* is our overall belief that the correct answer lies somewhere in the set p.

In order to be able to use m (and thus *Bel* and *Pl*) in reasoning programs, we need to define functions that enable us to combine m's that arise from multiple sources of evidence.

Recall that in our discussion of *CF's,* we considered three combination scenarios, which we illustrated in Fig. 8.1. When we use Dempster-Shafer theory, on the other hand, we do not need an explicit combining function for the scenario in Fig. 8.1(b) since we have that capability already in our ability to assign a value of m to a set of hypotheses. But we do need a mechanism for performing the combinations of scenarios (a) and (c). Dempster's rule of combination serves both these functions. It allows us to combine any two belief functions (whether they represent multiple sources of evidence for a single hypothesis or multiple sources of evidence for different hypotheses).

Suppose we are given two belief functions m_1 and m_2. Let X be the set of subsets of Θ to which m_1 assigns a nonzero value and let Y be the corresponding set for m_2. We define the combination m_3 of m_1 and m_2 to be

$$m_3(Z) = \frac{\sum_{X \cap Y = Z} m_1(X) \cdot m_2(Y)}{1 - \sum_{X \cap Y = \phi} m_1(X) \cdot m_2(Y)}$$

This gives us a new belief function that we can apply to any subset Z of Θ. We can describe what this formula is doing by looking first at the simple case in which all ways of intersecting elements of *X* and elements of *Y* generate nonempty sets. For example, suppose m_1 corresponds to our belief after observing fever:

{*Flu, Cold, Pneu*}	(0.6)
Θ	(0.4)

Suppose m_2 corresponds to our belief after observing a runny nose:

{*All, Flu,Cold*}	(0.8)
Θ	(0.2)

Then we can compute their combination m_3 using the following table (in which we further abbreviate disease names), which we can derive using the numerator of the combination rule:

		{*A, F, C*}	(0.8)	*Θ*	(0.2)
{*F, C, P*}	(0.6)	{*F, C*}	(0.48)	{*F, C, P*}	(0.12)
Θ	(0.4)	{*A, F, C*}	(0.32)	*Θ*	(0.08)

The four sets that are generated by taking all ways of intersecting an element of *X* and an element of *Y* are shown in the body of the table. The value of m_3 that the combination rule associates with each of them is computed by multiplying the values of m_1 and m_2 associated with the elements from which they were derived. Although it did not happen in this simple case, it is possible for the same set to be derived in more than one way during this intersection process. If that does occur, then to compute m_3 for that set, it is necessary to compute the sum of all the individual values that are generated for all the distinct ways in which the set is produced (thus the summation sign in the numerator of the combination formula).

A slightly more complex situation arises when some of the subsets created by the intersection operation are empty. Notice that we are guaranteed by the way we compute m_3 that the sum of all its individual values is 1 (assuming that the sums of all the values of m_1 and m_2 are 1). If some empty subsets are created, though, then some of m_3 will be assigned to them. But from the fact that we assumed that *0* is exhaustive, we know that the true value of the hypothesis must be contained in some nonempty subset of *0*. So we need to redistribute any belief that ends up in the empty subset proportionately across the nonempty ones. We do that with the scaling factor shown in the denominator of the combination formula. If no nonempty subsets are created, the scaling factor is 1, so we were able to ignore it in our first example. But to see how it works, let's add a new piece of evidence to our example. As a result of applying m_1 and m_2, we produced *my*.

{*Flu, Cold*}	(0.48)
{*All, Flu,Cold*}	(0.32)
{*Flu,Cold,Pneu*}	(0.12)
Θ	(0.08)

Now, let m_4 correspond to our belief given just the evidence that the problem goes away when the patient goes on a trip:

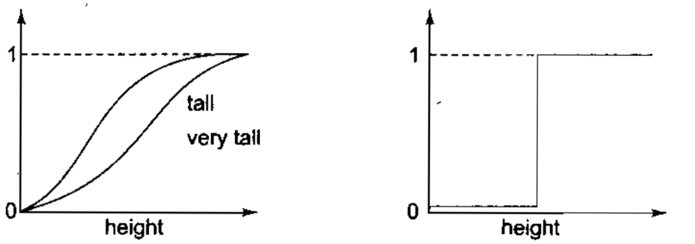
{*All*}	(0.9)
Θ	(0.1)

We can apply the numerator of the combination rule to produce (where * denotes the empty set):

		{*A*}	(0.9)	*Θ*	(0.1)
{*F*, *C*}	(0.48)	ϕ	(0.432)	{*F*, *C*}	(0.048)
{*A*, *F*, *C*}	(0.32)	{*A*,*F*,*C*}	(0.288)	{*A*, *F*, *C*}	(0.032)
{*F*, *C*, *P*}	(0.12)	ϕ	(0.108)	{*F*, *C*, *P*}	(0.012)
Θ	(0.08)	{*A*}	(0.072)	*Θ*	(0.008)

But there is now a total belief of 0.54 associated with ϕ; only 0.45 is associated with outcomes that are in fact possible. So we need to scale the remaining values by the factor 1 — 0.54 = 0.46. If we do this, and also combine alternative ways of generating the set {*All*, *Flu*, *Cold*}, then we get the final combined belief function, m_5.

{*Flu*, *Cold*}	(0.104)
{*All*, *Flu*, *Cold*}	(0.696)
{*Flu*,*Cold*,*Pneu*}	(0.026)
{*All*}	(0.157)
Θ	(0:017)



(a) Fuzzy Membership (b) Conventional Membership

Fig. 8.4 *Fuzzy versus Conventional Set Membership*

In this example, the percentage of m_5 that was initially assigned to the empty set was large (over half). This happens whenever there is conflicting evidence (as in this case between m_1 and m_4).

8.5 FUZZY LOGIC

In the techniques we have discussed so far, we have hot modified the mathematical underpinnings provided by set theory and logic. We have instead augmented those ideas with additional constructs provided by probability theory. In this section, we take a different approach and briefly consider what happens if we make fundamental changes to our idea of set membership and corresponding changes to our definitions of logical operations.

The motivation for fuzzy sets is provided by the need to represent such propositions as:

John is very tall.
Mary is slightly ill.
Sue and Linda are close friends.
Exceptions to the rule are nearly impossible.
Most Frenchmen are not very tall.

While traditional set theory defines set membership as a boolean predicate, fuzzy set theory allows us to represent set membership as a possibility distribution, such as the ones shown in Fig. 8.4(*a*) for the set of tall people and the set of very tall people. Notice how this contrasts with the standard Boolean definition for tall people shown in Fig. 8.4(b). In the latter, one is either tall or not and there must be a specific height that defines the boundary. The same is true for very tall. In the former, one's tallness increases with one's height until the value of 1 is reached.

Once set membership has been redefined in this way, it is possible to define a reasoning system based on techniques for combining distributions [Zadeh, 1979] (or see the papers in the journal *Fuzzy Sets and Systems*). Such reasoners have been applied in control systems for devices as diverse as trains and washing machines. A typical fuzzy logic control system has been described in Chapter 22.

SUMMARY

In this chapter we have shown that Bayesian statistics provide a good basis for reasoning under various kinds of uncertainty. We have also, though, talked about its weaknesses in complex real tasks, and so we have talked about ways in which it can be modified to work in practical domains. The thing that all of these modifications have in common is that they substitute, for the huge joint probability matrix that a pure Bayesian approach requires, a more structured representation of the facts that are relevant to a particular problem. They typically do this by combining probabilistic information with knowledge that is represented using one or more other representational mechanisms, such as rules or constraint networks.

Comparing these approaches for use in a particular problem-solving program is not always straightforward, since they differ along several dimensions, for example:

- They provide different mechanisms for describing the ways in which propositions are not independent of each other.
- They provide different techniques for representing ignorance.
- They differ substantially in the ease with which systems that use them can be built and in the computational complexity that the resulting systems exhibit.

We have also presented fuzzy logic as an alternative for representing some kinds of uncertain knowledge. Although there remain many arguments about the relative overall merits of the Bayesian and the fuzzy approaches, there is some evidence that they may both be useful in capturing different kinds of information. As an example, consider the proposition

> John was pretty sure that Mary was seriously ill.

Bayesian approaches naturally capture John's degree of certainty, while fuzzy techniques ran describe the degree of Mary's illness.

Throughout all of this discussion, it is important to keep in mind the fact that although we have been discussing techniques for representing knowledge, there is another perspective from which what we have really been doing is describing ways of representing *lack* of knowledge. In this sense, the techniques we have described in this chapter are fundamentally different from the ones we talked about earlier. For example, the truth values that we manipulate in a logical system characterize the formulas that we write; certainty measures, on the other hand, describe the exceptions — the facts that do not appear anywhere in the formulas that we have written. The consequences of this distinction show up in the ways that we can interpret and manipulate the formulas that we write. The most important difference is that logical formulas can be treated as though they represent independent propositions. As we have seen throughout this chapter, uncertain assertions cannot. As a result, for example, while implication is transitive in logical systems, we often get into trouble in uncertain systems if we treat it as though it were (as we saw in our first treatment of the sprinkler and grass example). Another difference is that in logical systems it is necessary to find only a single proof to be able to assert the truth value of a proposition. All other proofs, if there are any, can safely be ignored. In uncertain systems, on the other hand, computing belief in a proposition requires that all available reasoning paths be followed and combined.

One final comment is in order before we end this discussion. You may have noticed throughout this chapter that we have not maintained a clear distinction among such concepts as probability, certainty, and belief. This is because although there has been a great deal of philosophical debate over the meaning of these various terms, there is no clear argreement on how best to interpret them if our goal is to create working programs. Although the idea that probability should be viewed as a measure of belief rather than as a summary of past experience is now quite widely held, we have chosen to avoid the debate in this presentation. Instead, we have used all those words with their everyday, undifferentiated meaning, and we have concentrated on providing simple descriptions of how several algorithms actually work. If you are interested in the philosophical issues, see, for example, Shafer [1976] and Pearl [1988].

Unfortunately, although in the last two chapters we have presented several important approaches to the problem of uncertainty management, we have barely scraped the surface of this area. For more information, see Kanal and Lemmer [1986], Kanal and Lemmer [1988], Kanal *et al.* [1989], Shafer and Pearl [1990], Clark [1990]. In particular, our list of specific techniques is by no means complete. For example, you may wish to look into probabilistic logic [Nilsson, 1986; Halpern, 1989], in which probability theory is combined with logic so that the truth value of a formula is a probability value (between 0 and 1) rather than a boolean value (TRUE or FALSE). Or you may wish to ask not what statistics can do for AI but rather what AI can do for statistics. In that case, see Gale [1986].

EXERCISES

1. Consider the following puzzle:

   > A pea is placed under one of three shells, and the shells are then manipulated in such a fashion that all three appear to be equally likely to contain the pea. Nevertheless, you win a prize if you guess the correct shell, so you make a guess. The person running the game does know the correct shell, however, and uncovers one of the shells that you did not choose and that is empty. Thus, what remains are two shells: one you chose and one you did not choose. Furthermore, since the uncovered shell did not contain the pea, one of the two remaining shells does contain it. You are offered the opportunity to change your selection to the other shell. Should you?

   Work through the conditional probabilities mentioned in this problem using Bayes' theorem. What do the results tell about what you should do'
2. Using MYCIN's rules for inexact reasoning, compute *CF, MB,* and *MD* of h_1 given three observations where

$$CF(h_1, o_1) = 0.5$$
$$CF(h_1, o_2) = 0.3$$
$$CF(h_1, o_3) = -0.2$$

3. Show that MYCIN's combining rules satisfy the three properties we gave for them.
4. Consider the following set of propositions:

   patient has spots
   patient has measles

patient has high fever
patient has Rocky Mountain Spotted Fever
patient has previously been innoculated against measles
patient was recently bitten by a tick
patient has an allergy

(a) Create a network that defines the causal connections among these nodes.
(b) Make it a Bayesian network by constructing the necessary conditional probability matrix.

5. Consider the same propositions again, and assume our task is to identify the patient's disease using Dempster-Shafer theory.
   (a) What is Θ?
   (b) Define a set of m functions that describe the dependencies among sources of evidence and elements of Θ.'
   (c) Suppose we have observed spots, fever, and a tick bite. In that case, what is our *Bel*({*RockyMountainSpottedFever*})?
6. Define fuzzy sets that can be used to represent the list of propositions that we gave at the beginning of Section 8.5.
7. Consider again the ABC Murder story from Chapter 7. In our discussion of it there, we focused on the use of symbolic techniques for representing and using uncertain knowledge. Let's now explore the use of numeric techniques to solve the same problem. For each part below, show how knowledge could be represented. Whenever possible, show how it can be combined to produce a prediction of who committed the murder given at least one possible configuration of the evidence.
   (a) Use MYCIN-style rules and *CF's*. Example rules might include:

```
If  (1) relative (x,y), and
    (2) on speaking terms (x,y),
then there is suggestive evidence (0.7) that
   will-lie-for (x,y)
```

   (b) Use Bayesian networks. Represent as nodes such propositions as brother- in-law-lied, Cabot-at-ski-meet, and so forth.
   (c) Use Dempster-Shafer theory. Examples of/w's might be:

$m_1 = \{Abbott,\ Babbitt\}$	(0.8)	{*beneficiaries in will*}
Θ (0.2)		
$m_2 = \{Abbott,\ Cabot\}$	(0.7)	{*in line for his job*}
Θ (0.3)		

   (d) Use fuzzy logic. For example, you might want to define such fuzzy sets as honest people or greedy people and describe Abbott, Babbitt, and Cabot's memberships in those sets.
   (e) What kinds of information arc easiest (and hardest) to represent in each of these frameworks?

CHAPTER 9

WEAK SLOT-AND-FILLER STRUCTURES

Speech is the representation of the mind, and writing is the representation of speech

—Aristotle
(384 BC – 322 BC), Greek philosopher

In this chapter, we continue the discussion we began in Chapter 4 of slot-and-filler structures. Recall that we originally introduced them as a device to support property inheritance along *isa* and *instance* links. This is an important aspect of these structures. Monotonic inheritance can be performed substantially more efficiently with such structures than with pure logic, and nonmonotonic inheritance is easily supported. The reason that inheritance is easy is that the knowledge in slot-and-filler systems is structured as a set of entities and their attributes. This structure turns out to be a useful one for other reasons besides the support of inheritance, though, including:

- It indexes assertions by the entities they describe. More formally, it indexes binary predicates [such as *team*{*Three-Finger-Brown, Chicago-Cubs*) by their first argument. As a result, retrieving the value for an attribute of an entity is fast.
- It makes it easy to describe properties of relations. To do this in a purely logical system requires some higher-order mechanisms.
- It is a form of object-oriented programming and has the advantages that such systems normally have, including modularity and ease of viewing by people.

We describe two views of this kind of structure: semantic nets and frames. We talk about the representations themselves and about techniques for reasoning with them. We do not say much, though, about the specific knowledge that the structures should contain. We call these "knowledge-poor" structures "weak," by analogy with the weak methods for problem solving that we discussed in Chapter 3. In the next chapter, we expand this discussion to include "strong" slot-and-filler structures, in which specific commitments to the content of the representation are made.

9.1 SEMANTIC NETS

The main idea behind semantic nets is that the meaning of a concept comes from the ways in which it is connected to other concepts. In a semantic net, information is represented as a set of nodes connected to each other by a set of labeled arcs, which represent relationships among the nodes. A fragment of a typical semantic net is shown in Fig. 9.1.

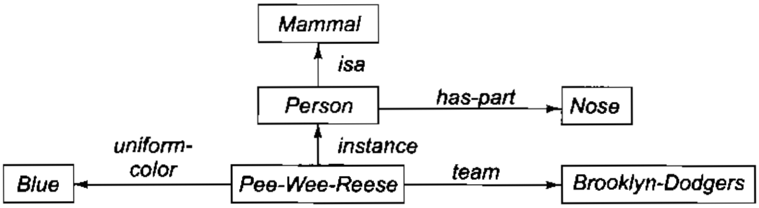


Fig. 9.1 *A Semantic Network*

This network contains examples of both the *isa* and *instance* relations, as well as some other, more domain-specific relations like *team* and *uniform-color.* In this network, we could use inheritance to derive the additional relation

has-part (*Pee-Wee-Reese, Nose*)

9.1.1 Intersection Search

One of the early ways that semantic nets were used was to find relationships among objects by spreading activation out from each of two nodes and seeing where the activation met. This process is called *intersection search* [Quillian, 1968]. Using this process, it is possible to use the network of Fig. 9.1 to answer questions such as "What is the connection between the Brooklyn Dodgers and blue?"[1] This kind of reasoning exploits one of the important advantages that slot-and-filler structures have over purely logical representations because it takes advantage of the entity-based organization of knowledge that slot-and-fiiler representations provide.

To answer more structured questions, however, requires networks that are themselves more highly structured. In the next few sections we expand and refine our notion of a network in order to support more sophisticated reasoning.

9.1.2 Representing Nonbinary Predicates

Semantic nets are a natural way to represent relationships that would appear as ground instances of binary predicates in predicate logic. For example, some of the arcs from Fig. 9.1 could be represented in logic as

isa(*Person, Mammal*)
instance(*Pee-Wee-Reese, Person*)
team(*Pee-Wee-Reese, Brooklyn-Dodgers*)
uniform-color(*Pee-Wee-Reese, Blue*)

But the knowledge expressed by predicates of other arities can also be expressed in semantic nets. We have already seen that many unary predicates in logic can be thought of as binary predicates using some very general-purpose predicates, such as *isa* and *instance.* So, for example,

man(*Marcus*)

[1] Actually, to do this we need to assume that the inverses of the links we have shown also exist.

could be rewritten as

instance(*Marcus, Man*)

thereby making it easy to represent in a semantic net.

Three or more place predicates can also be converted to a binary form by creating one new object representing the entire predicate statement and then introducing binary predicates to describe the relationship to this new object of each of the original arguments. For example, suppose we know that

score(*Cubs, Dodgers,* 5-3)

This can be represented in a semantic net by creating a node to represent the specific game and then relating each of the three pieces of information to it. Doing this produces the network shown in Fig. 9.2.

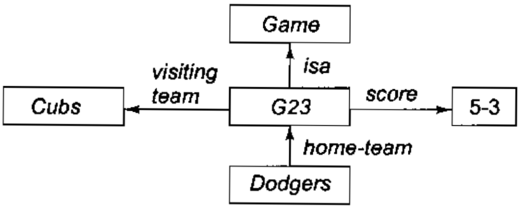


Fig. 9.2 *A Semantic Net for an n-Place Predicate*

This technique is particularly useful for representing the contents of a typical declarative sentence that describes several aspects of a particular event. The sentence

John gave the book to Mary.

could be represented by the network shown in Fig. 9.3.[2] In fact, several of the earliest uses of semantic nets were in English-understanding programs.

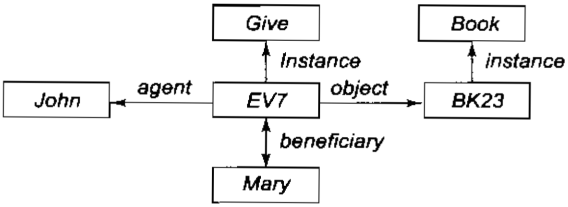
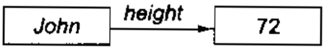


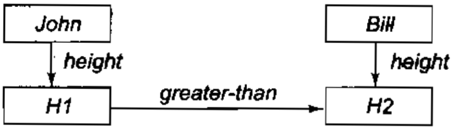
Fig. 9.3 *A Semantic Net Representing a Sentence*

9.1.3 Making Some Important Distinctions

In the networks we have described so far, we have glossed over some distinctions that are important in reasoning. For example, there should be a difference between a link that defines a new entity and one that relates two existing entities. Consider the net



Both nodes represent objects that exist independently of their relationship to each other. But now suppose we want to represent the fact that John is taller than Bill, using the net

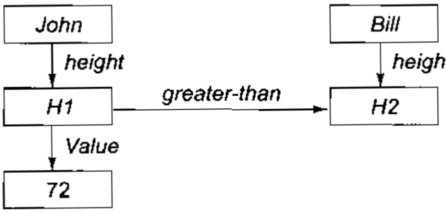


The nodes *H*1 and *H*2 are new concepts representing John's height and Bill's height, respectively. They are defined by their relationships to the nodes *John* and *Bill.* Using these defined concepts, it is possible to

[2] The node labeled *BK23* represents the particular book that was referred to by the phrase "the book." Discovering which particular book was meant by that phrase is similar to the problem of deciding on the correct referent for a pronoun, and it can be a very hard problem. These issues are discussed in Section 15.4.

represent such facts as that John's height increased, which we could not do before. (The number 72 increased?)

Sometimes it is useful to introduce the arc *value* to make this distinction clear. Thus we might use the following net to represent the fact that John is 6 feet tall and that he is taller than Bill:



The procedures that operate on nets such as this can exploit the fact that some arcs, such as *height*, define new entities, while others, such as *greater-than* and *value*, merely describe relationships among existing entities.

Another example of an important distinction we have missed is the difference between the properties of a node itself and the properties that a node simply holds and passes on to its instances. For example, it is a property of the node *Person* that it is a subclass of the node *Mammal*. But the node *Person* does not have as one of its parts a nose. Instances of the node *Person* do, and we want them to inherit it.

It is difficult to capture these distinctions without assigning more structure to our notions of node, link, and value. In the next section, when we talk about frame systems, we do that. But first, we discuss a network-oriented solution to a simpler problem; this solution illustrates what can be done in the network model but at what price in complexity.

9.1.4 Partitioned Semantic Nets

Suppose we want to represent simple quantified expressions in semantic nets. One way to do this is to *partition* the semantic net into a hierarchical set of *spaces*, each of which corresponds to the scope of one or more variables [Hendrix, 1977]. To see how this works, consider first the simple net shown in Fig. 9.4(a). This net corresponds to the statement

The dog bit the mail carrier

The nodes *Dot's*, *Bite*, and *Mail-Carrier* represent the classes of dogs, bitings, and mail carriers, respectively, while the nodes *d*, *b*, and *m* represent a particular dog, a particular biting, and a particular mail carrier. This fact can easily be represented by a single net with no partitioning.

But now suppose that we want to represent the fact

Every dog has bitten a mail carrier,

or, in logic:

$$\forall x : Dog(x) \rightarrow \exists y : Mail\text{-}Carrier\ (y) \wedge Bite \wedge (x, y)$$

To represent this fact, it is necessary to encode the scope of the universally quantified variable *x*. This can be done using partitioning as shown in Fig. 9.4(b). The node *g* stands for the assertion given above. Node *g* is an instance of the special class *GS* of general statements about the world (i.e., those with universal quantifiers). Every element of *GS* has at least two attributes: a *form*, which states the relation that is being asserted, and one or more ∀ connections, one for each of the universally quantified variables. In this example, there is only one such variable *d*, which can stand for any element of the class *Dogs*. The other two variables in the form, *b* and *m*, are understood to be existentially quantified. In other words, for every dog *d*, there exists a biting event *b*, and a mail carrier *m*, such that *d* is the assailant of *b* and *m* is the victim.

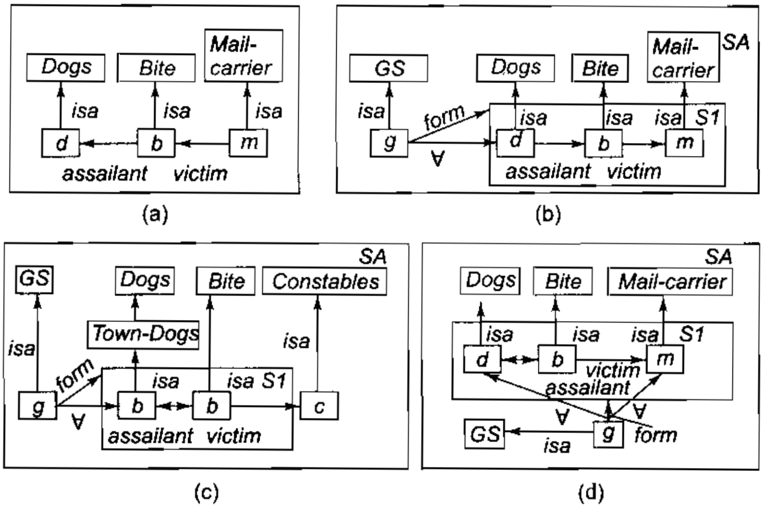


Fig. 9.4 *Using Partitioned Semantic Nets*

To see how partitioning makes variable quantification explicit, consider next the similar sentence:

> Every dog in town has bitten the constable.

The representation of this sentence is shown in Fig. 9.4(c). In this net, the node *c* representing the victim lies outside the form of the general statement. Thus it is not viewed as an existentially quantified variable whose value may depend on the value of *d*. Instead it is interpreted as standing for a specific entity (in this case, a particular constable), just as do other nodes in a standard, nonpartitioned net.

Figure 9.4(d) shows how yet another similar sentence:

> Every dog has bitten every mail carrier.

would be represented. In this case, *g* has two ∀ links, one pointing to *d*, which represents any dog, and one pointing to *m*, representing any mail carrier.

The spaces of a partitioned semantic net are related to each other by an inclusion hierarchy. For example, in Fig. 9.4(d), space *S*1 is included in space *SA*. Whenever a search process operates in a partitioned semantic net, it can explore nodes and arcs in the space from which it starts and in other spaces that contain the starting point, but it cannot go downward, except in special circumstances, such as when *a form* arc is being traversed. So, returning to Fig. 9.4(d), from node *d* it can be determined that *d* must be a dog. But if we were to start at the node *Dogs* and search for all known instances of dogs by traversing *isa* links, we would not find *d* since it and the link to it are in the space *S*1, which is at a lower level than space *SA*, which contains *Dogs*. This is important, since *d* does not stand for a particular dog; it is merely a variable that can be instantiated with a value that represents a dog.

9.1.5 The Evolution into Frames

The idea of a semantic net started out simply as a way to represent labeled connections among entities. But, as we have just seen, as we expand the range of problem-solving tasks that the representation must support, the representation itself necessarily begins to become more complex. In particular, it becomes useful to assign more structure to nodes as well as to links. Although there is no clear distinction between a semantic net and a frame system, the more structure the system has, the more likely it is to be termed a frame system. In the next section we continue our discussion of structured slot-and-filler representations by describing some of the most important capabilities that frame systems offer.

9.2 FRAMES

A frame is a collection of attributes (usually called slots) and associated values (and possibly constraints on values) that describe some entity in the world. Sometimes a frame describes an entity in some absolute sense; sometimes it represents the entity from a particular point of view (as it did in the vision system proposal [Minsky, 1975] in which the term *frame* was first introduced). A single frame taken alone is rarely useful. Instead, we build frame systems out of collections of frames that are connected to each other by virtue of the fact that tbe value of an attribute of one frame may be another frame. In the rest of this section, we expand on this simple definition and explore ways that frame systems can be used to encode knowledge and support reasoning

9.2.1 Frames as Sets and Instances

The Set theory provides a good basis for understanding frame systems. Although not all frame systems are defined this way, we do so here. In this view, each frame represents either a class (a set) or an instance (an element of a class). To see how this works, consider the frame system shown in Fig. 9.5, which is a slightly modified form of the network we showed in Fig. 9.5. In this example, the frames *Person, Adult-Male, ML-Baseball-Player* (corresponding to major league baseball players), *Pitcher,* and *ML-Baseball-Team* (for major league baseball team) are all classes. The frames *Pee-Wee-Reese* and *Brooklyn-Dodgers* are instances.

The *isa* relation that we have been using without a precise definition is in fact the *subset* relation. The set of adult males is a subset of the set of people. The set of major league baseball players is a subset of the set of adult males, and so forth. Our *instance* relation corresponds to the relation *element-of.* Pee Wee Reese is an element of the set of fielders. Thus he is also an element of all of the supersets of fielders, including major league baseball players and people. The transitivity of *isa* that we have taken for granted in our description of property inheritance follows directly from the transitivity of the subset relation.

Both the *isa* and *instance* relations have inverse attributes, which we call *subclasses* and *all-instances.* We do not bother to write them explicitly in our examples unless we need to refer to them. We assume that the frame system maintains them automatically, either explicitly or by computing them if necessary.

Because a class represents a set, there are two kinds of attributes that can be associated with it. There are attributes about the set itself, and there are attributes that are to be inherited by each element of the set. We indicate the difference between these two by prefixing the latter with an asterisk (*). For example, consider the class *ML-Baseball-Player.* We have shown only two properties of it as a set: It is a subset of the set of adult males. And it has cardinality 624 (i.e., there are 624 major league baseball players). We have listed five properties that all major league baseball players have (*height, bats, batting-average, team,* and *uniform-color*), and we have specified default values for the first three of them. By providing both kinds of slots, we allow a class both to define a set of objects and to describe a prototypical object of the set.

Sometimes, the distinction between a set and an individual instance may not seem clear. For example, the team *Brooklyn-Dodgers,* which we have described as an instance of the class of major league baseball teams, could be thought of as a set of players. In fact, notice that the value of the slot *players* is a set. Suppose, instead, that we want to represent the Dodgers as a class instead of an instance. Then its instances would be the individual players. It cannot stay where it is in the *isa* hierarchy; it cannot be a subclass of *ML-Baseball-Team,* because if it were, then its elements, namely the players, would also, by the transitivity of subclass, be elements of *ML-Baseball-Team,* which is not what we want to say. We have to put it somewhere else in the *isa* hierarchy. For example, we could make it a subclass of major league baseball players. Then its elements, the players, are also elements of *ML-Baseball-Player, Adult-Male,* and *Person.* That is acceptable. But if we do that, we lose the ability to inherit properties of the Dodgers from general information about baseball teams. We can still inherit attributes for the elements of the team, but we cannot inherit properties of the team as a whole, i.e., of the set of players. For example, we might like to know what the default size of the team is, that it has a manager, and so on. The easiest way to allow for this is to go back to the idea of the Dodgers as an instance of *ML-Baseball-Team,* with the set of players given as a slot value.

```
Person
    isa :                 Mammal
    cardinality :         6,000,000,000
    * handed :            Right
Adult-Male
    isa :                 Person
    cardinality :         2,000,000,000
    * height :            5-10
ML-Baseball-Player
    isa :                 Adult-Male
    cardinality :         624
    *height :             6-1
    * bats :              equal to handed
    * batting-average :   .252
    * team :
    * uniform-color :
Fielder
    isa :                 ML-Baseball-Player
    cardinality :         376
    *batting-average :    .262
Pee-Wee-Reese
    instance :            Fielder
    height :              5-10
    bats :                Right
    batting-average :     .309
    team :                Brooklyn-Dodgers
    uniform-color :       Blue
ML-Baseball-Team
    isa:                  Team
    cardinality :         26
    * team-size :         24
    * manager :
Brooklyn-Dodgers
    instance :            ML-Baseball-Team
    team-size :           24
    manager :             Leo-Durocher
    players :             {Pee-Wee-Reese,...}
```

Fig. 9.5 *A Simplified Frame System*

But what we have encountered here is an example of a more general problem. A class is a set, and we want to be able to talk about properties that its elements possess. We want to use inheritance to infer those properties from general knowledge about the set. But a class is also an entity in itself. It may possess properties that belong not to the individual instances but rather to the class as a whole. In the case of *Brooklyn-Dodgers*, such properties included team size and the existence of a manager. We may even want to inherit some of these properties from a more general kind of set. For example, the Dodgers can inherit a default team size from the set of all major league baseball teams. To support this, we need to view a class as two things simultaneously: a subset (*isa*) of a larger class that also contains its elements and an instance (*instance*) of a class of sets, from which it inherits its set-level properties.

To make this distinction clear, it is useful to distinguish between regular classes, whose elements are individual entities, and *metaclasses*, which are special classes whose elements are themselves classes. A class is now an element of (*instance*) some class (or classes) as well as a subclass (*isa*) of one or more classes. A class inherits properties from the class of which it is an instance, just as any instance does. In addition, a class passes inheritable properties down from its superclasses to its instances.

Let us consider an example. Figure 9.6 shows how we could represent teams as classes using this distinction. Figure 9.7 shows a graphic view of the same classes. The most basic metaclass is the class *Class*. It represents the set of all classes. All classes are instances of it, either directly or through one of its subclasses. In the example, *Team* is a subclass (subset) of *Class* and *ML-Baseball-Team* is a subclass of *Team*. The class *Class* introduces the attribute *cardinality*, which is to be inherited by all instances of *Class* (including itself). This makes sense since all the instances of *Class* are sets and all sets have a cardinality.

```
Class
    instance :          Class
    isa :               Class
    * cardinality :

Team
    instance :          Class
    isa :               Class
    cardinality :       {the number of teams that exist}
    *team-size :        {each team has a size}

ML-Baseball-Team

    isa :               Mammal
    instance :          Class
    isa :                Team
    cardi nality :      26 {the number of baseball teams that exist}
    * team-size :       24 {default 24 players on a team}
    * manager :

Brooklyn-Dodgers
    instance :          ML-Baseball-Team
    isa :               ML-Baseball-Player
    team-size :         24
    manager :           Leo-Durocher
    * uniform-color :   Blue

Pee-Wee-Reese
    instance :          Brooklyn-Dodgers
    instance :          Fielder
    uniform-color :     Blue
    batting-average :   .309
```

Fig. 9.6 *Representing the Class of All Teams as a Metaclass*

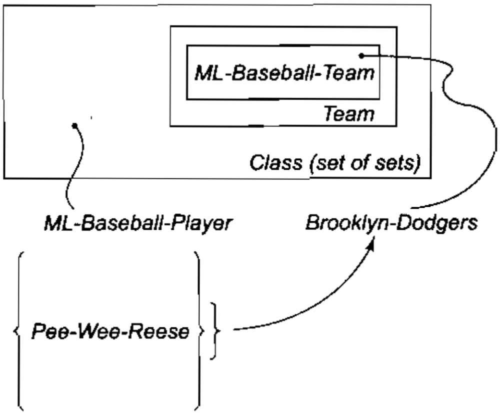


Fig. 9.7 *Classes and Metaclasses*

Team represents a subset of the set of all sets, namely those whose elements are sets of players on a team. It inherits the property of having a cardinality from *Class*. *Team* introduces the attribute *team-size,* which all its elements possess. Notice that *team-size* is like *cardinality* in that it measures the size of a set. But it applies to something different; *cardinality* applies to sets of sets and is inherited by all elements of *Class*. The slot *team-size* applies to the elements of those sets that happen to be teams. Those elements are sets of individuals.

ML-Baseball-Team is also an instance of *Class*, since it is a set. It inherits the property of having a cardinality from the set of which it is an instance, namely *Class*. But it is a subset of *Team*. All of its instances will have the property of having a *team-size* since they are also instances of the superclass *Team*. We have added at this level the additional fact that the default team size is 24, so all instances of *ML-Baseball-Team* will inherit that as well. In addition, we have added the inheritable slot *manager.*

Brooklyn-Dodgers is an instance of a *ML-Baseball-Team*. It is not an instance of *Class* because its elements are individuals, not sets. *Brooklyn-Dodgers* is a subclass of *ML-Baseball-Player* since all of its elements are also elements of that set. Since it is an instance of a *ML-Baseball-Team,* it inherits the properties *team-size* and *manager,* as well as their default values. It specifies a new attribute *uniform-color,* which is to be inherited by all of its instances (who will be individual players).

Finally, *Pee-Wee-Reese* is an instance of *Brooklyn-Dodgers*. That makes him also, by transitivity up *isa* links, an instance of *ML-Baseball-Player.* But recall that in our earlier example we also used the class *Fielder,* to which we attached the fact that fielders have above-average batting averages. To allow that here, we simply make Pee Wee an instance of *Fielder* as well. He will thus inherit properties from both *Brooklyn-Dodgers* and from *Fielder,* as well as from the classes above these. We need to guarantee that when multiple inheritance occurs, as it does here, that it works correctly. Specifically, in this case, we need to assure that *batting-average* gets inherited from *Fielder* and not from *ML-Baseball-Player* through *Brooklyn-Dodgers*. We return to this issue in Section 9.2.5.

In all the frame systems we illustrate, all classes are instances of the metaclass *Class*. As a result, they all have the attribute *cardinality.* We leave the class *Class*, the *isa* links to it, and the attribute *cardinality* out of our descriptions of our examples, though, unless there is some particular reason to include them.

Every class is a set. But not every set should be described as a class. A class describes a set of entities that share significant properties. In particular, the default information associated with a class can be used as a basis for inferring values for the properties of its individual elements. So there is an advantage to representing as a class those sets for which membership serves as a basis for nonmonotonic inheritance. Typically, these are sets in which membership is not highly ephemeral. Instead, membership is based on some fundamental structural or functional properties. To see the difference, consider the following sets:

- People
- People who are major league baseball players
- People who are on my plane to New York

The first two sets can be advantageously represented as classes, with which a substantial number of inheritable attributes can be associated. The last, though, is different. The only properties that all the elements of that set probably share are the definition of the set itself and some other properties that follow from the definition (e.g., they are being transported from one place to another). A simple set, with some associated assertions, is adequate to represent these facts; nonmonotonic inheritance is not necessary.

9.2.2 Other Ways of Relating Classes to Each Other

We have talked up to this point about two ways in which classes (sets) can be related to each other. $Class_1$ can be a subset of $Class_2$. Or, if $Class_2$ is a metaclass, then $Class_1$ can be an instance of $Class_2$. But there are other ways that classes can be related to each other, corresponding to ways that sets of objects in the world can be related.

One such relationship is *mutually-disjoint-with,* which relates a class to one or more other classes that are guaranteed to have no elements in common with it. Another important relationship is *is-covered-by* which relates a class to a set of subclasses, the union of which is equal to it. If a class *is-covered-by* a set S of mutually disjoint classes, then S is called *a partition* of the class.

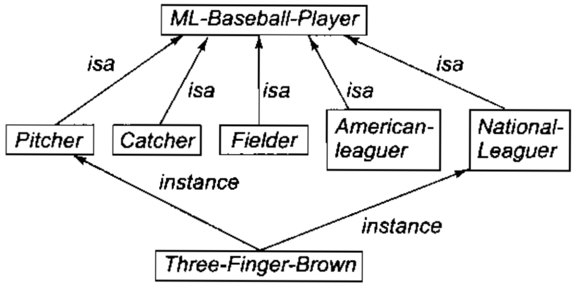
For examples of these relationships, consider the classes shown in Fig. 9.8, which represent two orthogonal ways of decomposing the class of major league baseball players. Everyone is either a pitcher, a catcher, or a fielder (and no one is more than one of these). In addition, everyone plays in either the National League or the American League, but not both.

9.2.3 Slots as Full-Fledged Objects

So far, we have provided a way to describe sets of objects and individual objects, both in terms of attributes and values. Thus we have made extensive use of attributes, which we have represented as slots attached to frames. But it turns out that there are several reasons why we would like to be able to represent attributes explicitly and describe their properties. Some of the properties we would like to be able to represent and use in reasoning include:

- The classes to which the attribute can be attached, i.e. for what classes does it make sense? For example, weight makes sense for physical objects but not for conceptual ones (except in some metaphorical sense).
- Constraints on either the type or the value of the attribute. For example, the age of a person must be a numeric quantity measured in some time frame, and it must be less than the ages of the person's biological parents.
- A value that all instances of a class must have by the definition of the class.
- A default value for the attribute.
- Rules for inheriting values for the attribute. The usual rule is to inherit down *isa* and *instance* links. But some attributes inherit in other ways. For example, *last-name* inherits down the *child-of* link.
- Rules for computing a value separately from inheritance. One extreme form of such a rule is a procedure written in some procedural programming language such as LISP.
- An inverse attribute.
- Whether the slot is single-valued or multivalued.

In order to be able to represent these attributes of attributes, we need to describe attributes (slots) as frames. These frames will be organized into an *isa* hierarchy, just as any other frames are, and that hierarchy can then be used to support inheritance of values for attributes of slots. Before we can describe such a hierarchy in detail, we need to formalize our notion of a slot.



ML-Baseball-Player	
is-covered-by :	*{Pitcher, Catcher, Fielder}* *{American-Leaguer, National-Leaguer}*
Pitcher	
isa :	*ML-Baseball-Player*
mutually-disjoint-with :	*{Catcher, Fielder}*
Catcher	
isa :	*ML-Baseball-Player*
mutually-disjoint-with:	*{Pitcher, Fielder}*
Fielder	
isa :	*ML-Baseball-Player*
mutually-disjoint-with :	*{Pitcher, Catcher}*
American-Leaguer	
isa :	*ML-Baseball-Player*
mutually-disjoint-with :	*{National-Leaguer}*
National-Leaguer	
isa :	*ML-Baseball-Player*
mutually-disjoint-with :	*{American-Leaguer}*
Three-Finger-Brown	
instance :	*Pitcher*
instance :	*National-Leaguer*

Fig. 9.8 *Representing Relationships among Classes*

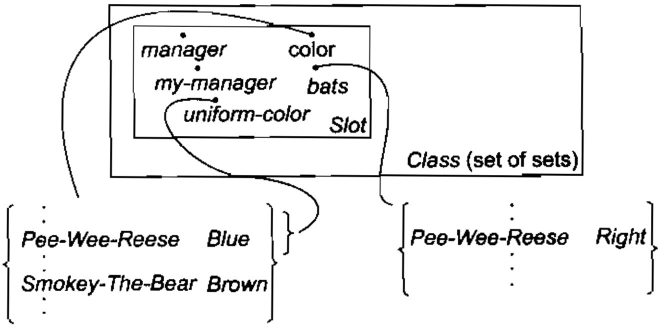
A slot is a relation. It maps from elements of its domain (the classes for which it makes sense) to elements of its range (its possible values). A relation is a set of ordered pairs. Thus it makes sense to say that one relation (R_1) is a subset of another (R_2). In that case, R_1 is a specialization of R_2, so in our terminology *isa* (R_1, R_2). Since a slot is a set, the set of all slots, which we will call *Slot,* is a metaclass. Its instances are slots, which may have subslots.

Figures 9.9 and 9.10 illustrate several examples of slots represented as frames. *Slot* is a metaclass. Its instances are slots (each of which is a set of ordered pairs). Associated with the metaclass are attributes that each instance (i.e., each actual slot) will inherit. Each slot, since it is a relation, has a domain and a range. We represent the domain in the slot labeled *domain.* We break up the representation of the range into two parts: *range* gives the class of which elements of the rangemust be elements; *range-constraint* contains a logical expression that further constrains the range to be elements of *range* that also satisfy the constraint. *If range-constraint* is absent, it is taken to be TRUE. The advantage to breaking the description apart into these two pieces is that type checking is much cheaper than is arbitrary constraint checking, so it is useful to be able to do it separately and early during some reasoning processes.

The other slots do what you would expect from their names. If there is a value for *definition,* it must be propagated to all instances of the slot. If there is a value for *default,* that value is inherited to all instances of the slot unless there is an overriding value. The attribute *transfers-through* lists other slots from which values for this slot can be derived through inheritance. The *to-compute* slot contains a procedure for deriving its value. The *inverse* attribute contains the inverse of the slot. Although in principle all slots have inverses, sometimes they are not useful enough in reasoning to be worth representing. And *single-valued* is used to mark the special cases in which the slot is a function and so can have only one value.

Of course, there is no advantage to representing these properties of slots if there is no reasoning mechanism that exploits them. In the rest of our discussion, we assume that the frame-system interpreter knows how to reason with all of these slots of slots as part of its built-in reasoning capability. In particular, we assume that it is capable of performing the following reasoning actions:

- Consistency checking to verify that when a slot value is added to a frame
  - The slot makes sense for the frame. This relies on the *domain* attribute of the slot.
  - The value is a legal value for the slot. This relies on the *range* and *range-constraints* attributes.
- Maintenance of consistency between the values for slots and their inverses when ever one is updated.
- Propagation of *definition* values along *isa* and *instance* links.
- Inheritance of *default* values along *isa* and *instance* links.



Slot

isa :	*Class*
instance :	*Class*
* *domain* :	
* *range* :	
* *range-constraint* :	
* *definition* :	
* *default* :	
* *transfers-through* :	
* *to-compute* :	
* *inverse* :	
* *single-valued* :	

manager

instance :	*Slot*
domain :	*ML-Baseball-Team*
range :	*Person*
range-constraint :	λx (*baseball-experience* x.*manager*)
default :	
inverse :	*manager-of*
single-valued :	TRUE

Fig. 9.9 *Representing Slots as Frames, I*

```
my-manager
    instance :            Slot
    domain :              ML-Baseball-Player
    range :               Person
    range-constraint :    λx (baseball-experience x.my-manager)
    to-compute :          λx (x.team).manager
    single-valued :       TRUE
color
    instance :            Slot
    domain :              Physical-Object
    range :               Color-Set
    transfers-through :   top-level-part-of
    visual-salience :     High
    single-valued :       FALSE
uniform-color
    instance :            Slot
    isa :                 color
    domain :              team-player
    range :               Color-Set
    range-constraint :    not Pink
    visual-salience :     High
    single-valued :       FALSE
bats
    instance :            Slot
    domain :              ML-Baseball-Player
    range :               {Left, Right, Switch}
    to-compute :          λx x.handed
    single-valued :       TRUE
```

Fig. 9.10 *Representing Slots as Frames, II*

- Computation of a value of a slot as needed. This relies on the *to-compute* and *transfers-through* attributes.
- Checking that only a single value is asserted *for single-valued* slots. This is usually done by replacing an old value by the new one when it is asserted. An alternative is to force explicit retraction of the old value and to signal a contradiction if a new value is asserted when another is already there.

There is something slightly counterintuitive about this way of defining slots. We have defined the properties *range-constraint* and *default* as parts of a slot. But we often think of them as being properties of a slot associated with a particular class. For example, in Fig. 9.5, we listed two defaults for the *batting-average* slot, one associated with major league baseball players and one associated with fielders. Figure 9.11 shows how

```
batting-average
    instance :            Slot
    domain :              ML-Baseball-Player
    range :               Number
    range-constraint :    λx (0 ≤ x.range-constraint ≤ 1)
    default :             .252
    single-valued :       TRUE
fielder-batting-average
    instance :            Slot
    isa :                 batting-average
    domain :              Fielder
    range :               Number
    range-constraint :    λx (0 ≤ x.range-constraint ≤ 1)
    default :             .262
    single-valued :       TRUE
```

Fig. 9.11 *Associating Defaults with Slots*

this can be represented correctly by creating a specialization of *fielding-average* that can be associated with a specialization of *ML-Baseball-Player* to represent the more specific information that is known about the specialized class. This seems cumbersome. It is natural, though, given our definition of a slot as a relation. There are really two relations here, one a specialization of the other. And below we will define inheritance so that it looks for values of either the slot it is given or any of that slot's generalizations.

Unfortunately, although this model of slots is simple and it is internally consistent, it is not easy to use. So we introduce some notational shorthand that allows the four most important properties of a slot (domain, range, definition, and default) to be defined implicitly by how the slot is used in the definitions of the classes in its domain. We describe the domain implicitly to be the class where the slot appears. We describe the range and any range constraints with the clause MUST BE, as the value of an inherited slot. Figure 9.12 shows an example of this notation. And we describe the definition and the default, if they are present, by inserting them as the value of the slot when it appears. The two will be distinguished by prefixing a definitional value with an asterisk (*). We then let the underlying bookkeeping of the frame system create the frames that represent slots as they are needed.

```
ML-Baseball-Player
    bats :                    MUST BE {Left, Right, Switch}
```

Fig. 9.12 *A Shorthand Notation for Slot-Range Specification*

Now let's look at examples of how these slots can be used. The slots *bats* and *my-manager* illustrate the use of the *to-compute* attribute of a slot. The variable x will be bound to the frame to which the slot is attached. We use the dot notation to specify the value of a slot of a frame. Specifically, $x.y$ describes the value(s) of the y slot o^r frame x. So we know that to compute a frame's value for *my-manager,* it is necessary to find the frame's value for *team,* then find the resulting team's manager. We have simply composed two slots to form a new one.[3] Computing the value of the *bats* slot is even simpler. Just go get the value of the *handed* slot.

The *manager* slot illustrates the use of a range constraint. It is stated in terms of a variable x, which is bound to the frame whose *manager* slot is being described. It requires that any manager be not only a person but someone with baseball experience. It relies on the domain-specific function *baseball-experience,* which must be defined somewhere in the system.

The slots *color* and *uniform-color* illustrate the arrangement of slots in an *isa* hierarchy. The relation *color* is a fairly general one that holds between physical objects and colors. The attribute *uniform-color* is a restricted form of *color* that applies only between team players and the colors that are allowed for team uniforms (anything but pink). Arranging slots in a hierarchy is useful for the same reason that arranging any thing else in a hierarchy is: it supports inheritance. In this example, the general slot *color* is known to have high visual salience. The more specific slot *uniform-color* then inherits this property, so it too is known to have high visual salience.

The slot *color* also illustrates the use of the *transfers-through* slot, which defines a way of computing a slot's value by retrieving it from the same slot of a related object. In this example, we used *transfers-through* to capture the fact that if you take an object and chop it up into several top level parts (in other words, parts that are not contained inside each other), then they will all be the same color. For example, the arm of a sofa is the same color as the sofa. Formally, what *transfers-through* means in this example is

$$color\,(x,\,y) \wedge top\text{-}level\text{-}part\text{-}of\,(z,\,x) \rightarrow color(z,\,y)$$

In addition to these domain-independent slot attributes, slots may have domain-specific properties that support problem solving in a particular domain. Since these slots are not treated explicitly by the frame-system interpreter, they will be useful precisely to the extent that the domain problem solver exploits them.

[3] Notice that since slots are relations rather than functions, their composition may return a set of values.

9.2.4 Slot-Values as Objects

In the last section, we reified the notion of a slot by making it an explicit object that we could make assertions about. In some sense this was not necessary. A finite relation can be completely described by listing its elements. But in practical knowledge-based systems one often does not have that list. So it can be very important to be able to make assertions about the list without knowing all of its elements. Reification gave us a way to do this.

The next step along this path is to do the same thing to a particular attribute-value (an instance of a relation) that we did to the relation itself. We can reify it and make it an object about which assertions can be made. To see why we might want to do this, let us return to the example of John and Bill's height that we discussed in Section 9.1.3. Figure 9.13 shows a frame-based representation of some of the facts. We could easily record Bill's height if we knew it. Suppose, though, that we do not know it. All we know is that John is taller than Bill. We need a way to make an assertion about the value of a slot without knowing what that value is. To do that, we need to view the slot and its value as an object.

```
John
    height :        72
Bill
    height :
```

Fig. 9.13 *Representing Slot-Values*

We could attempt to do this the same way we made slots themselves into objects, namely by representing them explicitly as frames. There seems little advantage to doing that in this case, though, because the main advantage of frames does not apply to slot values: frames are organized into an *isa* hierarchy and thus support inheritance. There is no basis for such an organization of slot values. So instead, we augment our value representation language to allow the value of a slot to be stated as either or both of:

- A value of the type required by the slot.
- A logical constraint on the value. This constraint may relate the slot's value to he values of other slots or to domain constants.

If we do this to the frames of Fig. 9.13, then we get the frames of Fig. 9.14. We again use the lambda notation as a way to pick up the name of the frame that is being described.

```
John
    height :        72; λx (x.height > Bill.height)
Bill
    height :        λx (x.height < John.height)
```

Fig. 9.14 *Representing Slot-Values with Lambda Notation*

9.2.5 Inheritance Revisited

In Chapter 4, we presented a simple algorithm for inheritance. But that algorithm assumed that the *isa* hierarchy was a tree. This is often not the case. To support flexible representations of knowledge about the world, it is necessary to allow the hierarchy to be an arbitrary directed acyclic graph (DAG). We know that acyclic graphs are adequate because *isa* corresponds to the subset relation. Hierarchies that are not trees are called *tangled hierarchies*. Tangled hierarchies require a new inheritance algorithm. In the rest of this section, we discuss an algorithm for inheriting values for single-valued slots in a tangled hierarchy. We leave the problem of inheriting multivalued slots as an exercise.

Consider the two examples shown in Fig. 9.15 (in which we return to a network notation to make it easy to visualize the *isa* structure). In Fig. 9.15(a), we want to decide whether *Fifi* can fly. The correct answer is no.

Although birds in general can fly, the subset of birds, ostriches, does not. Although the class *Pet-Bird* provides a path from *Fifi* to *Bird* and thus to the answer that *Fifi* can fly, it provides no information that conflicts with the special case knowledge associated with the class *Ostrich,* so it should have no affect on the answer. To handle this case correctly, we need an algorithm for traversing the *isa* hierarchy that guarantees that specific knowledge will always dominate more general facts.

In Fig. 9.15(b), we return to a problem we discussed in Section 7.2.1, namely determining whether Dick is a pacifist. Again, we must traverse multiple *instance* links, and more than one answer can be found along the paths. But in this case, there is no well-founded basis for choosing one answer over the other. The classes that are associated with the candidate answers are incommensurate with each other in the partial ordering that is defined by the DAG formed by the *isa* hierarchy. Just as we found that in Default Logic this theory had two extensions and there was no principled basis for choosing between them, what we need here is an inheritance algorithm that reports the ambiguity; we do not want an algorithm that finds one answer (arbitrarily) and stops without noticing the other.

One possible basis for a new inheritance algorithm is path length. This can be implemented by executing a breadth-first search, starting with the frame for which a slot value is needed. Follow its *instance* links, then follow *isa* links upward. If a path produces a value, it can be terminated, as can all other paths once their length exceeds that of the successful path. This algorithm works for both of the examples in Fig. 9.15. In (a), it finds a value at *Ostrich.* It continues the other path to the same length (*Pet-Bird*), fails to find any other answers, and then halts. In the case of (b), it finds two competing answers at the same level, so it can report the contradiction.

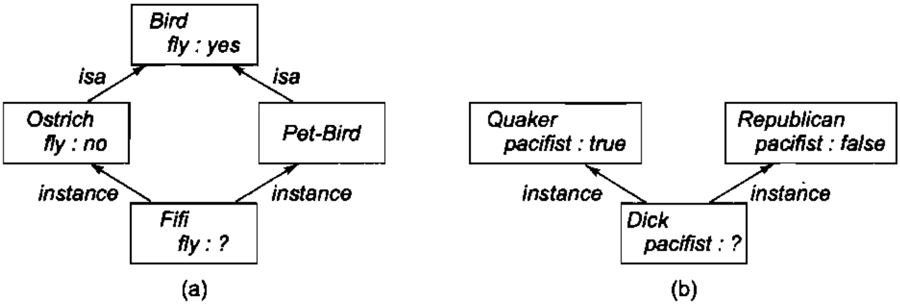


Fig. 9.15 *Tangled Hierarchies*

But now consider the examples shown in Fig. 9.16. In the case of (a), our new algorithm reaches *Bird* (via *Pet-Bird*) before it reaches *Ostrich.* So it reports that *Fifi* can fly. In the case of (b), the algorithm reaches *Quaker* and stops without noticing a contradiction. The problem is that path length does not always correspond to the level of generality of a class. Sometimes what it really corresponds to is the degree of elaboration of classes in the knowledge base. If some regions of the knowledge base have been elaborated more fully than others, then their paths will tend to be longer. But this should not influence the result of inheritance if no new information about the desired attribute has been added.

The solution to this problem is to base our inheritance algorithm not on path length but on the notion of *inferential distance* [Touretzky, 1986], which can be defined as follows:

$Class_1$ is closer to $Class_2$ than to $Class_3$, if and only if $Class_1$ has an inference path through $Class_2$ to $Class_3$ (in other words, $Class_2$ is between $Class_1$ and $Class_3$).

Notice that inferential distance defines only a partial ordering. Some classes are incommensurate with each other under it.

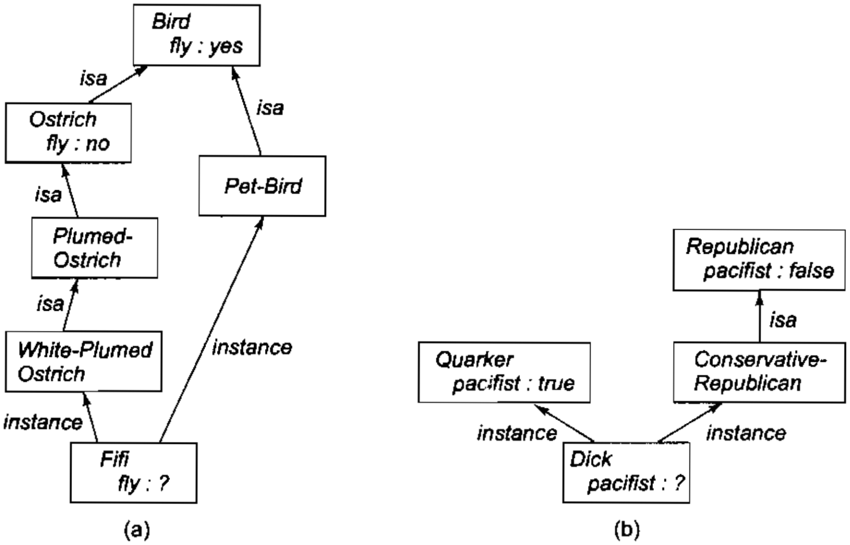


Fig. 9.16 *More Tangled Hierarchies*

We can now define the result of inheritance as follows: The set of competing values for a slot *S* in a frame *F* contains all those values that

- Can be derived from some frame *X* that is above *F* in the *isa* hierarchy
- Are not contradicted by some frame *Y* that has a shorter inferential distance to *F* than *X* does

Notice that under this definition competing values that are derived from incommensurate frames continue to compete.

Using this definition, let us return to our examples. For Fig. 9.15(a), we had two candidate classes from which to get an answer. But *Ostrich* has a shorter inferential distance to *Fifi* than *Bird* does, so we get the single answer no. For Fig. 9.15(b), we get two answers, and neither is closer to *Dick* than the other, so we correctly identify a contradiction. For Fig. 9.16(a), we get two answers, but again *Ostrich* has a shorter inferential distance to *Fifi* than *Bird* does. The significant thing about the way we have defined inferential distance is that as long as *Ostrich* is a subclass of *Bird*, it will be closer to all its instances than *Bird* is, no matter how many other classes are added to the system. For Fig. 9.16(b), we again get two answers and again neither is closer to *Dick* than the other.

There are several ways that this definition can be implemented as an inheritance algorithm. We present a simple one. It can be made more efficient by caching paths in the hierarchy, but we do not do that here.

Algorithm: Property Inheritance

To retrieve a value *V* for slot S of an instance *F* do:

1. Set *CANDIDATES* to empty.
2. Do breadth-first or depth-first search up the *isa* hierarchy from *F*, following all *instance* and *isa* links. At each step, see if a value for *S* or one f its generalizations is stored.
   (a) If a value is found, add it to *CANDIDATES* and terminate that branch of the search.
   (b) If no value is found but there are *instance* or *isa* links upward, follow them.
   (c) Otherwise, terminate the branch.

3. For each element *C* of *CANDIDATES* do:
   (a) See if there is any other element of *CANDIDATES* that was derived from a class closer to *F* than the class from which *C* came.
   (b) If there is, then, remove *C* from *CANDIDATES.*
4. Check the cardinality of *CANDIDATES:*
   (a) If it is 0, then report that no value was found.
   (b) If it is 1, then return the single element of *CANDIDATES* as *V*.
   (c) If it is greater than 1, report a contradiction.

This algorithm is guaranteed to terminate because the *isa* hierarchy is represented as an acyclic graph.

9.2.6 Frame Languages

The idea of a frame system as a way to represent declarative knowledge has been encapsulated in a series of frame-oriented knowledge representation languages, whose features have evolved and been driven by an increased understanding of the sort of representation issues we have been discussing. Examples of such languages include KRL [Bobrow and Winograd, 1977], FRL [Roberts and Goldstein, 1977], RLL [Greiner and Lenat, 1980], KL-ONE [Brachman, 1979; Brachman and Schmolze, 1985], KRYPTON [Brachman *et al.*, 1985], NIKL [Kaczmarek *et al.*, 1986], CYCL [Lenat and Guha, 1990], conceptual graphs [Sowa, 1984], THEO [Mitchell *et al.*, 1989], and FRAMEKIT [Nyherg, 1988]. Although not all of these systems support all of the capabilities that we have discussed, the more modern of these systems permit elaborate and efficient representation of many kinds of knowledge. Their reasoning methods include most of the ones described here, plus many more, including subsumption checking, automatic classification, and various methods for consistency maintenance.

EXERCISES

1. Construct semantic net representations for the following:
   (a) *Pompeian*(*Marcus*), *Blacksmith*(*Marcus*)
   (b) Mary gave the green flowered vase to her favorite cousin.
2. Suppose we want to use a semantic net to discover relationships that could help in disambiguating the word "bank" in the sentence

   John went downtown to deposit his money in the bank.

   The financial institution meaning for bank should be preferred over the river bank meaning.
   (a) Construct a semantie net that contains representations for the relevant concepts.
   (b) Show how intersection seareh could be used to find the connection between the correct meaning for bank and the rest of the sentence more easily than it can find a connection with the incorrect meaning.
3. Construct partitioned semantic net representations for the following:
   (a) Every batter hit a ball.
   (b) All the batters like the pitcher.
4. Construct one consistent frame representation of all the baseball knowledge that was described in this chapter. You will need to choose between the two representations for team that we considered.
5. Modify the property inheritance algorithm of Section 9.2 to work for multiple-valued attributes, sueh as the attribute *believes-in-principles*, defined as follows:

believes-in-principles
instance : *Slot*
domain : *Person*
range : *Philosophical-Principles*
single-valued : FALSE

6. Define the value of a multiple-valued slot *S* of class *C* to be the union of the values that are found for *S* and all its generalizations at *C* and all its generalizations. Modify your technique to allow a class to exclude specific values that are associated with one or more of its superclasses.
7. Pick a problem area and represent some knowledge about it the way we represented baseball knowledge in this chapter.
8. How would you classify and represent the various types of triangles?