signals—depends only on its *type*. We can use a function to refer to the type of the gate.[11] For example, we can write $Type(X_1) = XOR$. This introduces the constant XOR for a particular type of gate; the other constants will be called OR, AND, and NOT. The $Type$ function is not the only way to encode the ontological distinction. We could have used a binary predicate, $Type(X_1, XOR)$, or several individual type predicates, such as $XOR(X_1)$. Either of these choices would work fine, but by choosing the function $Type$, we avoid the need for an axiom which says that each individual gate can have only one type. The semantics of functions already guarantees this.

Next we consider terminals. A gate or circuit can have one or more input terminals and one or more output terminals. We could simply name each one with a constant, just as we named gates. Thus, gate X_1 could have terminals named X_1In_1, X_1In_2, and X_1Out_1. The tendency to generate long compound names should be avoided, however. Calling something X_1In_1 does not make it the first input of X_1; we would still need to say this using an explicit assertion. It is probably better to name the gate using a function, just as we named King John's left leg $LeftLeg(John)$. Thus, let $In(1, X_1)$ denote the first input terminal for gate X_1. A similar function Out is used for output terminals.

The connectivity between gates can be represented by the predicate $Connected$, which takes two terminals as arguments, as in $Connected(Out(1, X_1), In(1, X_2))$.

Finally, we need to know whether a signal is on or off. One possibility is to use a unary predicate, On, which is true when the signal at a terminal is on. This makes it a little difficult, however, to pose questions such as "What are all the possible values of the signals at the output terminals of circuit C1 ?" We will therefore introduce as objects two "signal values" 1 and 0, and a function $Signal$ that takes a terminal as argument and denotes the signal value for that terminal.

Encode general knowledge of the domain

One sign that we have a good ontology is that there are very few general rules which need to be specified. A sign that we have a good vocabulary is that each rule can be stated clearly and concisely. With our example, we need only seven simple rules to describe everything we need to know about circuits:

1. If two terminals are connected, then they have the same signal:
   $\forall t_1, t_2 \;\; Connected(t_1, t_2) \Rightarrow Signal(t_1) = Signal(t_2)$
2. The signal at every terminal is either 1 or 0 (but not both):
   $\forall t \;\; Signal(t) = 1 \vee Signal(t) = 0$
   $1 \neq 0$
3. Connected is a commutative predicate:
   $\forall t_1, t_2 \;\; Connected(t_1, t_2) \Leftrightarrow Connected(t_2, t_1)$
4. An OR gate's output is 1 if and only if any of its inputs is 1:
   $\forall g \;\; Type(g) = OR \Rightarrow$
   $Signal(Out(1, g)) = 1 \Leftrightarrow \exists n \;\; Signal(In(n, g)) = 1$

[11] Note that we have used names beginning with appropriate letters—A_1, X_1, and so on—purely to make the example easier to read. The knowledge base must still contain type information for the gates.

5. An AND gate's output is 0 if and only if any of its inputs is 0:
$$\forall g \ \ Type(g) = AND \Rightarrow Signal(Out(1,g)) = 0 \Leftrightarrow \exists n \ \ Signal(In(n,g)) = 0$$
6. An XOR gate's output is 1 if and only if its inputs are different:
$$\forall g \ \ Type(g) = XOR \Rightarrow Signal(Out(1,g)) = 1 \Leftrightarrow Signal(In(1,g)) \neq Signal(In(2,g))$$
7. A NOT gate's output is different from its input:
$$\forall g \ \ (Type(g) = NOT) \Rightarrow Signal(Out(1,g)) \neq Signal(In(1,g))$$

Encode the specific problem instance

The circuit shown in Figure 8.4 is encoded as circuit C_1 with the following description. First, we categorize the gates:

$$Type(X_1) = XOR \quad Type(X_2) = XOR$$
$$Type(A_1) = AND \quad Type(A_2) = AND$$
$$Type(O_1) = OR$$

Then, we show the connections between them:

$$Connected(Out(1,X_1), In(1,X_2)) \quad Connected(In(1,C_1), In(1,X_1))$$
$$Connected(Out(1,X_1), In(2,A_2)) \quad Connected(In(1,C_1), In(1,A_1))$$
$$Connected(Out(1,A_2), In(1,O_1)) \quad Connected(In(2,C_1), In(2,X_1))$$
$$Connected(Out(1,A_1), In(2,O_1)) \quad Connected(In(2,C_1), In(2,A_1))$$
$$Connected(Out(1,X_2), Out(1,C_1)) \quad Connected(In(3,C_1), In(2,X_2))$$
$$Connected(Out(1,O_1), Out(2,C_1)) \quad Connected(In(3,C_1), In(1,A_2)) .$$

Pose queries to the inference procedure

What combinations of inputs would cause the first output of C_1 (the sum bit) to be 0 and the second output of C_1 (the carry bit) to be 1?

$$\exists i_1, i_2, i_3 \ \ Signal(In(1,C_1)) = i_1 \wedge Signal(In(2,C_1)) = i_2 \wedge Signal(In(3,C_1)) = i_3 \wedge Signal(Out(1,C_1)) = 0 \wedge Signal(Out(2,C_1)) = 1 .$$

The answers are substitutions for the variables i_1, i_2, and i_3 such that the resulting sentence is entailed by the knowledge base. There are three such substitutions:

$$\{i_1/1, \ i_2/1, \ i_3/0\} \quad \{i_1/1, \ i_2/0, \ i_3/1\} \quad \{i_1/0, \ i_2/1, \ i_3/1\} .$$

What are the possible sets of values of all the terminals for the adder circuit?

$$\exists i_1, i_2, i_3, o_1, o_2 \ \ Signal(In(1,C_1)) = i_1 \wedge Signal(In(2,C_1)) = i_2 \wedge Signal(In(3,C_1)) = i_3 \wedge Signal(Out(1,C_1)) = o_1 \wedge Signal(Out(2,C_1)) = o_2 .$$

This final query will return a complete input–output table for the device, which can be used to check that it does in fact add its inputs correctly. This is a simple example of **circuit verification**. We can also use the definition of the circuit to build larger digital systems, for which the same kind of verification procedure can be carried out. (See Exercise 8.17.) Many domains are amenable to the same kind of structured knowledge-base development, in which more complex concepts are defined on top of simpler concepts.

CIRCUIT VERIFICATION

Debug the knowledge base

We can perturb the knowledge base in various ways to see what kinds of erroneous behaviors emerge. For example, suppose we omit the assertion that $1 \neq 0$.[12] Suddenly, the system will be unable to prove any outputs for the circuit, except for the input cases 000 and 110. We can pinpoint the problem by asking for the outputs of each gate. For example, we can ask

$$\exists i_1, i_2, o \;\; Signal(In(1, C_1)) = i_1 \wedge Signal(In(2, C_1)) = i_2 \wedge Signal(Out(1, X_1))$$

which reveals that no outputs are known at X_1 for the input cases 10 and 01. Then, we look at the axiom for XOR gates, as applied to X_1:

$$Signal(Out(1, X_1)) = 1 \;\Leftrightarrow\; Signal(In(1, X_1)) \neq Signal(In(2, X_1)) \,.$$

If the inputs are known to be, say, 1 and 0, then this reduces to

$$Signal(Out(1, X_1)) = 1 \;\Leftrightarrow\; 1 \neq 0 \,.$$

Now the problem is apparent: the system is unable to infer that $Signal(Out(1, X_1)) = 1$, so we need to tell it that $1 \neq 0$.

8.5 SUMMARY

This chapter has introduced **first-order logic**, a representation language that is far more powerful than propositional logic. The important points are as follows:

- Knowledge representation languages should be declarative, compositional, expressive, context-independent, and unambiguous.
- Logics differ in their **ontological commitments** and **epistemological commitments**. While propositional logic commits only to the existence of facts, first-order logic commits to the existence of objects and relations and thereby gains expressive power.
- A **possible world**, or **model**, for first-order logic is defined by a set of objects, the relations among them, and the functions that can be applied to them.
- **Constant symbols** name objects, **predicate symbols** name relations, and **function symbols** name functions. An **interpretation** specifies a mapping from symbols to the model. **Complex terms** apply function symbols to terms to name an object. Given an interpretation and a model, the truth of a sentence is determined.
- An **atomic sentence** consists of a predicate applied to one or more terms; it is true just when the relation named by the predicate holds between the objects named by the terms. **Complex sentences** use connectives just like propositional logic, and **quantified sentences** allow the expression of general rules.
- Developing a knowledge base in first-order logic requires a careful process of analyzing the domain, choosing a vocabulary, and encoding the axioms required to support the desired inferences.

[12] This kind of omission is quite common because humans typically assume that different names refer to different things. Logic programming systems, described in Chapter 9, also make this assumption.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Although even Aristotle's logic deals with generalizations over objects, true first-order logic dates from the introduction of quantifiers in Gottlob Frege's (1879) *Begriffschrift* ("Concept Writing" or "Conceptual Notation"). Frege's ability to nest quantifiers was a big step forward, but he used an awkward notation. (An example appears on the front cover of this book.) The present notation for first-order logic is due substantially to Giuseppe Peano (1889), but the semantics is virtually identical to Frege's. Oddly enough, Peano's axioms were due in large measure to Grassmann (1861) and Dedekind (1888).

A major barrier to the development of first-order logic had been the concentration on one-place predicates to the exclusion of many-place relational predicates. This fixation on one-place predicates had been nearly universal in logical systems from Aristotle up to and including Boole. The first systematic treatment of relations was given by Augustus De Morgan (1864), who cited the following example to show the sorts of inferences that Aristotle's logic could not handle: "All horses are animals; therefore, the head of a horse is the head of an animal." This inference is inaccessible to Aristotle because any valid rule that can support this inference must first analyze the sentence using the two-place predicate "x is the head of y." The logic of relations was studied in depth by Charles Sanders Peirce (1870), who also developed first-order logic independently of Frege, although slightly later (Peirce, 1883).

Leopold Löwenheim (1915) gave a systematic treatment of model theory for first-order logic in 1915. This paper also treated the equality symbol as an integral part of logic. Löwenheim's results were further extended by Thoralf Skolem (1920). Alfred Tarski (1935, 1956) gave an explicit definition of truth and model-theoretic satisfaction in first-order logic, using set theory.

McCarthy (1958) was primarily responsible for the introduction of first-order logic as a tool for building AI systems. The prospects for logic-based AI were advanced significantly by Robinson's (1965) development of resolution, a complete procedure for first-order inference described in Chapter 9. The logicist approach took root at Stanford. Cordell Green (1969a, 1969b) developed a first-order reasoning system, QA3, leading to the first attempts to build a logical robot at SRI (Fikes and Nilsson, 1971). First-order logic was applied by Zohar Manna and Richard Waldinger (1971) for reasoning about programs and later by Michael Genesereth (1984) for reasoning about circuits. In Europe, logic programming (a restricted form of first-order reasoning) was developed for linguistic analysis (Colmerauer *et al.*, 1973) and for general declarative systems (Kowalski, 1974). Computational logic was also well entrenched at Edinburgh through the LCF (Logic for Computable Functions) project (Gordon *et al.*, 1979). These developments are chronicled further in Chapters 9 and 10.

There are a number of good modern introductory texts on first-order logic. Quine (1982) is one of the most readable. Enderton (1972) gives a more mathematically oriented perspective. A highly formal treatment of first-order logic, along with many more advanced topics in logic, is provided by Bell and Machover (1977). Manna and Waldinger (1985) give a readable introduction to logic from a computer science perspective. Gallier (1986) provides an extremely rigorous mathematical exposition of first-order logic, along with a great deal

of material on its use in automated reasoning. *Logical Foundations of Artificial Intelligence* (Genesereth and Nilsson, 1987) provides both a solid introduction to logic and the first systematic treatment of logical agents with percepts and actions.

EXERCISES

8.1 A logical knowledge base represents the world using a set of sentences with no explicit structure. An **analogical** representation, on the other hand, has physical structure that corresponds directly to the structure of the thing represented. Consider a road map of your country as an analogical representation of facts about the country. The two-dimensional structure of the map corresponds to the two-dimensional surface of the area.

a. Give five examples of *symbols* in the map language.

b. An *explicit* sentence is a sentence that the creator of the representation actually writes down. An *implicit* sentence is a sentence that results from explicit sentences because of properties of the analogical representation. Give three examples each of *implicit* and *explicit* sentences in the map language.

c. Give three examples of facts about the physical structure of your country that cannot be represented in the map language.

d. Give two examples of facts that are much easier to express in the map language than in first-order logic.

e. Give two other examples of useful analogical representations. What are the advantages and disadvantages of each of these languages?

8.2 Consider a knowledge base containing just two sentences: $P(a)$ and $P(b)$. Does this knowledge base entail $\forall x\ P(x)$? Explain your answer in terms of models.

8.3 Is the sentence $\exists x, y\ \ x = y$ valid? Explain.

8.4 Write down a logical sentence such that every world in which it is true contains exactly one object.

8.5 Consider a symbol vocabulary that contains c constant symbols, p_k predicate symbols of each arity k, and f_k function symbols of each arity k, where $1 \leq k \leq A$. Let the domain size be fixed at D. For any given interpretation–model combination, each predicate or function symbol is mapped onto a relation or function, respectively, of the same arity. You may assume that the functions in the model allow some input tuples to have no value for the function (i.e., the value is the invisible object). Derive a formula for the number of possible interpretation–model combinations for a domain with D elements. Don't worry about eliminating redundant combinations.

8.6 Represent the following sentences in first-order logic, using a consistent vocabulary (which you must define):

a. Some students took French in spring 2001.

b. Every student who takes French passes it.

c. Only one student took Greek in spring 2001.

d. The best score in Greek is always higher than the best score in French.

e. Every person who buys a policy is smart.

f. No person buys an expensive policy.

g. There is an agent who sells policies only to people who are not insured.

h. There is a barber who shaves all men in town who do not shave themselves.

i. A person born in the UK, each of whose parents is a UK citizen or a UK resident, is a UK citizen by birth.

j. A person born outside the UK, one of whose parents is a UK citizen by birth, is a UK citizen by descent.

k. Politicians can fool some of the people all of the time, and they can fool all of the people some of the time, but they can't fool all of the people all of the time.

8.7 Represent the sentence "All Germans speak the same languages" in predicate calculus. Use $Speaks(x, l)$, meaning that person x speaks language l.

8.8 What axiom is needed to infer the fact $Female(Laura)$ given the facts $Male(Jim)$ and $Spouse(Jim, Laura)$?

8.9 Write a general set of facts and axioms to represent the assertion "Wellington heard about Napoleon's death" and to correctly answer the question "Did Napoleon hear about Wellington's death?"

8.10 Rewrite the propositional wumpus world facts from Section 7.5 into first-order logic. How much more compact is this version?

8.11 Write axioms describing the predicates $GrandChild$, $GreatGrandparent$, $Brother$, $Sister$, $Daughter$, Son, $Aunt$, $Uncle$, $BrotherInLaw$, $SisterInLaw$, and $FirstCousin$. Find out the proper definition of mth cousin n times removed, and write the definition in first-order logic.

Now write down the basic facts depicted in the family tree in Figure 8.5. Using a suitable logical reasoning system, TELL it all the sentences you have written down, and ASK it who are Elizabeth's grandchildren, Diana's brothers-in-law, and Zara's great-grandparents.

8.12 Write down a sentence asserting that + is a commutative function. Does your sentence follow from the Peano axioms? If so, explain why; if not, give a model in which the axioms are true and your sentence is false.

8.13 Explain what is wrong with the following proposed definition of the set membership predicate $\in$:

$$\forall x, s \quad x \in \{x|s\}$$
$$\forall x, s \quad x \in s \Rightarrow \forall y \quad x \in \{y|s\} .$$

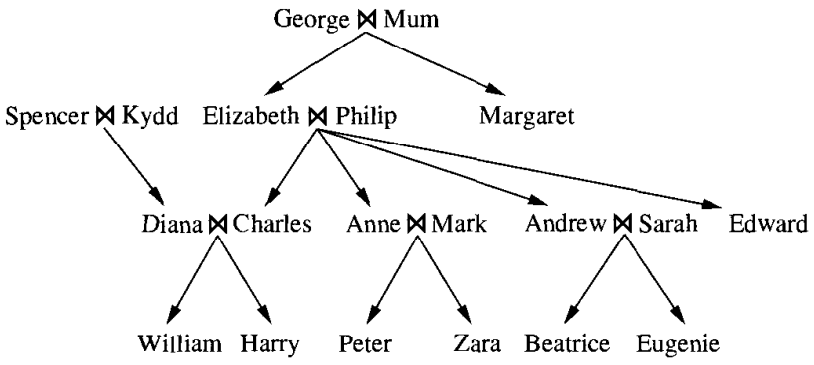


Figure 8.5 A typical family tree. The symbol "=" connects spouses and arrows point to children.

8.14 Using the set axioms as examples, write axioms for the list domain, including all the constants, functions, and predicates mentioned in the chapter.

8.15 Explain what is wrong with the following proposed definition of adjacent squares in the wumpus world:

$$\forall x, y \;\; Adjacent([x, y], [x + 1, y]) \wedge Adjacent([x, y], [x, y + 1]) .$$

8.16 Write out the axioms required for reasoning about the wumpus's location, using a constant symbol *Wumpus* and a binary predicate $In(Wumpus, Location)$. Remember that there is only one wumpus.

8.17 Extend the vocabulary from Section 8.4 to define addition for n-bit binary numbers. Then encode the description of the four-bit adder in Figure 8.6, and pose the queries needed to verify that it is in fact correct.

8.18 The circuit representation in the chapter is more detailed than necessary if we care only about circuit functionality. A simpler formulation describes any m-input, n-output gate or circuit using a predicate with $m+n$ arguments, such that the predicate is true exactly when

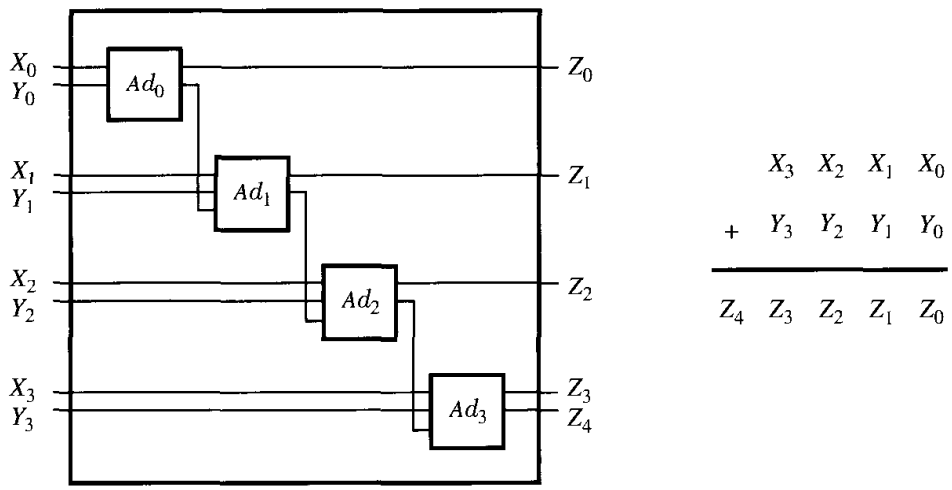


Figure 8.6 A four-bit adder.

the inputs and outputs are consistent. For example, NOT-gates are described by the binary predicate $NOT(i, o)$, for which $NOT(0, 1)$ and $NOT(1, 0)$ are known. Compositions of gates are defined by conjunctions of gate predicates in which shared variables indicate direct connections. For example, a NAND circuit can be composed from ANDs and NOTs:

$$\forall i_1, i_2, o_a, o \;\; NAND(i_1, i_2, o) \Leftarrow AND(i_1, i_2, o_a) \wedge NOT(o_a, o) \; .$$

Using this representation, define the one-bit adder in Figure 8.4 and the four-bit adder in Figure 8.6, and explain what queries you would use to verify the designs. What kinds of queries are *not* supported by this representation that *are* supported by the representation in Section 8.4?

8.19 Obtain a passport application for your country, identify the rules determining eligibility for a passport, and translate them into first-order logic, following the steps outlined in Section 8.4.

9 INFERENCE IN FIRST-ORDER LOGIC

In which we define effective procedures for answering questions posed in first-order logic.

Chapter 7 defined the notion of **inference** and showed how sound and complete inference can be achieved for propositional logic. In this chapter, we extend those results to obtain algorithms that can answer any answerable question stated in first-order logic. This is significant, because more or less anything can be stated in first-order logic if you work hard enough at it.

Section 9.1 introduces inference rules for quantifiers and shows how to reduce first-order inference to propositional inference, albeit at great expense. Section 9.2 describes the idea of **unification**, showing how it can be used to construct inference rules that work directly with first-order sentences. We then discuss three major families of first-order inference algorithms: **forward chaining** and its applications to **deductive databases** and **production systems** are covered in Section 9.3; **backward chaining** and **logic programming** systems are developed in Section 9.4; and resolution-based **theorem-proving** systems are described in Section 9.5. In general, one tries to use the most efficient method that can accommodate the facts and axioms that need to be expressed. Reasoning with fully general first-order sentences using resolution is usually less efficient than reasoning with definite clauses using forward or backward chaining.

9.1 PROPOSITIONAL VS. FIRST-ORDER INFERENCE

This section and the next introduce the ideas underlying modern logical inference systems. We begin with some simple inference rules that can be applied to sentences with quantifiers to obtain sentences without quantifiers. These rules lead naturally to the idea that *first-order* inference can be done by converting the knowledge base to *propositional* logic and using *propositional* inference, which we already know how to do. The next section points out an obvious shortcut, leading to inference methods that manipulate first-order sentences directly.

Inference rules for quantifiers

Let us begin with universal quantifiers. Suppose our knowledge base contains the standard folkloric axiom stating that all greedy kings are evil:

$$\forall x \;\; King(x) \wedge Greedy(x) \;\Rightarrow\; Evil(x) \,.$$

Then it seems quite permissible to infer any of the following sentences:

$$\begin{aligned} &King(John) \wedge Greedy(John) \;\Rightarrow\; Evil(John) \,. \\ &King(Richard) \wedge Greedy(Richard) \;\Rightarrow\; Evil(Richard) \,. \\ &King(Father(John)) \wedge Greedy(Father(John)) \;\Rightarrow\; Evil(Father(John)) \,. \\ &\;\;\vdots \end{aligned}$$

UNIVERSAL INSTANTIATION

The rule of **Universal Instantiation** (UI for short) says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable. [1] To write out the inference rule formally, we use the notion of **substitutions** introduced in Section 8.3. Let $\text{SUBST}(\theta, \alpha)$ denote the result of applying the substitution θ to the sentence α. Then the rule is written

$$\frac{\forall v \;\; \alpha}{\text{SUBST}(\{v/g\}, \alpha)}$$

for any variable v and ground term g. For example, the three sentences given earlier are obtained with the substitutions $\{x/John\}$, $\{x/Richard\}$, and $\{x/Father(John)\}$.

EXISTENTIAL INSTANTIATION

The corresponding **Existential Instantiation** rule for the existential quantifier is slightly more complicated. For any sentence α, variable v, and constant symbol k that does not appear elsewhere in the knowledge base,

$$\frac{\exists v \;\; \alpha}{\text{SUBST}(\{v/k\}, \alpha)} \,.$$

For example, from the sentence

$$\exists x \;\; Crown(x) \wedge OnHead(x, John)$$

we can infer the sentence

$$Crown(C_1) \wedge OnHead(C_1, John)$$

as long as C_1 does not appear elsewhere in the knowledge base. Basically, the existential sentence says there is some object satisfying a condition, and the instantiation process is just giving a name to that object. Naturally, that name must not already belong to another object. Mathematics provides a nice example: suppose we discover that there is a number that is a little bigger than 2.71828 and that satisfies the equation $d(x^y)/dy = x^y$ for x. We can give this number a name, such as e, but it would be a mistake to give it the name of an existing object, such as π. In logic, the new name is called a **Skolem constant**. Existential Instantiation is a special case of a more general process called **skolemization**, which we cover in Section 9.5.

SKOLEM CONSTANT

[1] Do not confuse these substitutions with the extended interpretations used to define the semantics of quantifiers. The substitution replaces a variable with a term (a piece of syntax) to produce a new sentence, whereas an interpretation maps a variable to an object in the domain.

As well as being more complicated than Universal Instantiation, Existential Instantiation plays a slightly different role in inference. Whereas Universal Instantiation can be applied many times to produce many different consequences, Existential Instantiation can be applied once, and then the existentially quantified sentence can be discarded. For example, once we have added the sentence $Kill(Murderer, Victim)$, we no longer need the sentence $\exists x\ Kill(x, Victim)$. Strictly speaking, the new knowledge base is not logically equivalent to the old, but it can be shown to be **inferentially equivalent** in the sense that it is satisfiable exactly when the original knowledge base is satisfiable.

INFERENTIAL EQUIVALENCE

Reduction to propositional inference

Once we have rules for inferring nonquantified sentences from quantified sentences, it becomes possible to reduce first-order inference to propositional inference. In this section we will give the main ideas; the details are given in Section 9.5.

The first idea is that, just as an existentially quantified sentence can be replaced by one instantiation, a universally quantified sentence can be replaced by the set of *all possible* instantiations. For example, suppose our knowledge base contains just the sentences

$$\begin{array}{l} \forall x\ \ King(x) \wedge Greedy(x) \Rightarrow Evil(x) \\ King(John) \\ Greedy(John) \\ Brother(Richard, John)\ . \end{array} \tag{9.1}$$

Then we apply UI to the first sentence using all possible ground term substitutions from the vocabulary of the knowledge base—in this case, $\{x/John\}$ and $\{x/Richard\}$. We obtain

$$\begin{array}{l} King(John) \wedge Greedy(John) \Rightarrow Evil(John)\ , \\ King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)\ , \end{array}$$

and we discard the universally quantified sentence. Now, the knowledge base is essentially propositional if we view the ground atomic sentences—$King(John)$, $Greedy(John)$, and so on—as proposition symbols. Therefore, we can apply any of the complete propositional algorithms in Chapter 7 to obtain conclusions such as $Evil(John)$.

PROPOSITIONALIZATION

This technique of **propositionalization** can be made completely general, as we show in Section 9.5; that is, every first-order knowledge base and query can be propositionalized in such a way that entailment is preserved. Thus, we have a complete decision procedure for entailment ... or perhaps not. There is a problem: When the knowledge base includes a function symbol, the set of possible ground term substitutions is infinite! For example, if the knowledge base mentions the $Father$ symbol, then infinitely many nested terms such as $Father(Father(Father(John)))$ can be constructed. Our propositional algorithms will have difficulty with an infinitely large set of sentences.

Fortunately, there is a famous theorem due to Jacques Herbrand (1930) to the effect that if a sentence is entailed by the original, first-order knowledge base, then there is a proof involving just a *finite* subset of the propositionalized knowledge base. Since any such subset has a maximum depth of nesting among its ground terms, we can find the subset by first generating all the instantiations with constant symbols ($Richard$ and $John$), then all terms of

depth 1 ($Father(Richard)$ and $Father(John)$), then all terms of depth 2, and so on, until we are able to construct a propositional proof of the entailed sentence.

We have sketched an approach to first-order inference via propositionalization that is **complete**—that is, any entailed sentence can be proved. This is a major achievement, given that the space of possible models is infinite. On the other hand, we do not know until the proof is done that the sentence *is* entailed! What happens when the sentence is *not* entailed? Can we tell? Well, for first-order logic, it turns out that we cannot. Our proof procedure can go on and on, generating more and more deeply nested terms, but we will not know whether it is stuck in a hopeless loop or whether the proof is just about to pop out. This is very much like the halting problem for Turing machines. Alan Turing (1936) and Alonzo Church (1936) both proved, in rather different ways, the inevitability of this state of affairs. *The question of entailment for first-order logic is* ***semidecidable****—that is, algorithms exist that say yes to every entailed sentence, but no algorithm exists that also says no to every nonentailed sentence.*

9.2 Unification and Lifting

The preceding section described the understanding of first-order inference that existed up to the early 1960s. The sharp-eyed reader (and certainly the computational logicians of the early 1960s) will have noticed that the propositionalization approach is rather inefficient. For example, given the query $Evil(x)$ and the knowledge base in Equation (9.1), it seems perverse to generate sentences such as $King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$. Indeed, the inference of $Evil(John)$ from the sentences

$$\forall x \;\; King(x) \wedge Greedy(x) \Rightarrow Evil(x)$$
$$King(John)$$
$$Greedy(John)$$

seems completely obvious to a human being. We now show how to make it completely obvious to a computer.

A first-order inference rule

The inference that John is evil works like this: find some x such that x is a king and x is greedy, and then infer that this x is evil. More generally, if there is some substitution θ that makes the premise of the implication identical to sentences already in the knowledge base, then we can assert the conclusion of the implication, after applying θ. In this case, the substitution $\{x/John\}$ achieves that aim.

We can actually make the inference step do even more work. Suppose that instead of knowing $Greedy(John)$, we know that *everyone* is greedy:

$$\forall y \;\; Greedy(y) \; . \tag{9.2}$$

Then we would still like to be able to conclude that $Evil(John)$, because we know that John is a king (given) and John is greedy (because everyone is greedy). What we need for this to work is find a substitution both for the variables in the implication sentence

and for the variables in the sentences to be matched. In this case, applying the substitution $\{x/John, y/John\}$ to the implication premises $King(x)$ and $Greedy(x)$ and the knowledge base sentences $King(John)$ and $Greedy(y)$ will make them identical. Thus, we can infer the conclusion of the implication.

GENERALIZED MODUS PONENS

This inference process can be captured as a single inference rule that we call **Generalized Modus Ponens**: For atomic sentences p_i, p_i', and q, where there is a substitution θ such that $\text{SUBST}(\theta, p_i') = \text{SUBST}(\theta, p_i)$, for all i,

$$\frac{p_1', \ p_2', \ \ldots, \ p_n', \ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)} .$$

There are $n+1$ premises to this rule: the n atomic sentences p_i' and the one implication. The conclusion is the result of applying the substitution θ to the consequent q. For our example:

p_1' is $King(John)$	p_1 is $King(x)$
p_2' is $Greedy(y)$	p_2 is $Greedy(x)$
θ is $\{x/John, y/John\}$	q is $Evil(x)$
$\text{SUBST}(\theta, q)$ is $Evil(John)$.	

It is easy to show that Generalized Modus Ponens is a sound inference rule. First, we observe that, for any sentence p (whose variables are assumed to be universally quantified) and for any substitution θ,

$$p \models \text{SUBST}(\theta, p) .$$

This holds for the same reasons that the Universal Instantiation rule holds. It holds in particular for a θ that satisfies the conditions of the Generalized Modus Ponens rule. Thus, from $p_1', \ldots, p_n'$ we can infer

$$\text{SUBST}(\theta, p_1') \wedge \ldots \wedge \text{SUBST}(\theta, p_n')$$

and from the implication $p_1 \wedge \ldots \wedge p_n \Rightarrow q$ we can infer

$$\text{SUBST}(\theta, p_1) \wedge \ldots \wedge \text{SUBST}(\theta, p_n) \Rightarrow \text{SUBST}(\theta, q) .$$

Now, θ in Generalized Modus Ponens is defined so that $\text{SUBST}(\theta, p_i') = \text{SUBST}(\theta, p_i)$, for all i; therefore the first of these two sentences matches the premise of the second exactly. Hence, $\text{SUBST}(\theta, q)$ follows by Modus Ponens.

LIFTING

Generalized Modus Ponens is a **lifted** version of Modus Ponens—it raises Modus Ponens from propositional to first-order logic. We will see in the rest of the chapter that we can develop lifted versions of the forward chaining, backward chaining, and resolution algorithms introduced in Chapter 7. The key advantage of lifted inference rules over propositionalization is that they make only those substitutions which are required to allow particular inferences to proceed. One potentially confusing point is that in one sense Generalized Modus Ponens is less general than Modus Ponens (page 211): Modus Ponens allows any single α on the left-hand side of the implication, while Generalized Modus Ponens requires a special format for this sentence. It is generalized in the sense that it allows any number of P_i'.

Unification

UNIFICATION

Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called **unification** and is a key component of all first-order

UNIFIER

inference algorithms. The UNIFY algorithm takes two sentences and returns a **unifier** for them if one exists:

$$\text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q) .$$

Let us look at some examples of how UNIFY should behave. Suppose we have a query $Knows(John, x)$: whom does John know? Some answers to this query can be found by finding all sentences in the knowledge base that unify with $Knows(John, x)$. Here are the results of unification with four different sentences that might be in the knowledge base.

$$\begin{aligned}
&\text{UNIFY}(Knows(John, x), Knows(John, Jane)) = \{x/Jane\} \\
&\text{UNIFY}(Knows(John, x), Knows(y, Bill)) = \{x/Bill, y/John\} \\
&\text{UNIFY}(Knows(John, x), Knows(y, Mother(y))) = \{y/John, x/Mother(John)\} \\
&\text{UNIFY}(Knows(John, x), Knows(x, Elizabeth)) = fail .
\end{aligned}$$

The last unification fails because x cannot take on the values $John$ and $Elizabeth$ at the same time. Now, remember that $Knows(x, Elizabeth)$ means "Everyone knows Elizabeth," so we *should* be able to infer that John knows Elizabeth. The problem arises only because the two sentences happen to use the same variable name, x. The problem can be avoided by **standardizing apart** one of the two sentences being unified, which means renaming its variables to avoid name clashes. For example, we can rename x in $Knows(x, Elizabeth)$ to z_{17} (a new variable name) without changing its meaning. Now the unification will work:

STANDARDIZING APART

$$\text{UNIFY}(Knows(John, x), Knows(z_{17}, Elizabeth)) = \{x/Elizabeth, z_{17}/John\} .$$

Exercise 9.7 delves further into the need for standardizing apart.

There is one more complication: we said that UNIFY should return a substitution that makes the two arguments look the same. But there could be more than one such unifier. For example, $\text{UNIFY}(Knows(John, x), Knows(y, z))$ could return $\{y/John, x/z\}$ or $\{y/John, x/John, z/John\}$. The first unifier gives $Knows(John, z)$ as the result of unification, whereas the second gives $Knows(John, John)$. The second result could be obtained from the first by an additional substitution $\{z/John\}$; we say that the first unifier is *more general* than the second, because it places fewer restrictions on the values of the variables. It turns out that, for every unifiable pair of expressions, there is a single **most general unifier** (or MGU) that is unique up to renaming of variables. In this case it is $\{y/John, x/z\}$.

MOST GENERAL UNIFIER

An algorithm for computing most general unifiers is shown in Figure 9.1. The process is very simple: recursively explore the two expressions simultaneously "side by side," building up a unifier along the way, but failing if two corresponding points in the structures do not match. There is one expensive step: when matching a variable against a complex term, one must check whether the variable itself occurs inside the term; if it does, the match fails because no consistent unifier can be constructed. This so-called **occur check** makes the complexity of the entire algorithm quadratic in the size of the expressions being unified. Some systems, including all logic programming systems, simply omit the occur check and sometimes make unsound inferences as a result; other systems use more complex algorithms with linear-time complexity.

OCCUR CHECK

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound
          y, a variable, constant, list, or compound
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(ARGS[x], ARGS[y], UNIFY(OP[x], OP[y], θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y], θ))
  else return failure
```

```
function UNIFY-VAR(var, x, θ) returns a substitution
  inputs: var, a variable
          x, any expression
          θ, the substitution built up so far

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression, such as $F(A, B)$, the function OP picks out the function symbol F and the function ARGS picks out the argument list (A, B).

Storage and retrieval

Underlying the TELL and ASK functions used to inform and interrogate a knowledge base are the more primitive STORE and FETCH functions. STORE(s) stores a sentence s into the knowledge base and FETCH(q) returns all unifiers such that the query q unifies with some sentence in the knowledge base. The problem we used to illustrate unification—finding all facts that unify with $Knows(John, x)$—is an instance of FETCHing.

The simplest way to implement STORE and FETCH is to keep all the facts in the knowledge base in one long list; then, given a query q, call UNIFY(q, s) for every sentence s in the list. Such a process is inefficient, but it works, and it's all you need to understand the rest of the chapter. The remainder of this section outlines ways to make retrieval more efficient, and can be skipped on first reading.

We can make FETCH more efficient by ensuring that unifications are attempted only with sentences that have *some* chance of unifying. For example, there is no point in trying

to unify $Knows(John, x)$ with $Brother(Richard, John)$. We can avoid such unifications by **indexing** the facts in the knowledge base. A simple scheme called **predicate indexing** puts all the *Knows* facts in one bucket and all the *Brother* facts in another. The buckets can be stored in a hash table[2] for efficient access.

INDEXING
PREDICATE INDEXING

Predicate indexing is useful when there are many predicate symbols but only a few clauses for each symbol. In some applications, however, there are many clauses for a given predicate symbol. For example, suppose that the tax authorities want to keep track of who employs whom, using a predicate $Employs(x, y)$. This would be a very large bucket with perhaps millions of employers and tens of millions of employees. Answering a query such as $Employs(x, Richard)$ with predicate indexing would require scanning the entire bucket.

For this particular query, it would help if facts were indexed both by predicate and by second argument, perhaps using a combined hash table key. Then we could simply construct the key from the query and retrieve exactly those facts that unify with the query. For other queries, such as $Employs(AIMA.org, y)$, we would need to have indexed the facts by combining the predicate with the first argument. Therefore, facts can be stored under multiple index keys, rendering them instantly accessible to various queries that they might unify with.

Given a sentence to be stored, it is possible to construct indices for *all possible* queries that unify with it. For the fact $Employs(AIMA.org, Richard)$, the queries are

$Employs(AIMA.org, Richard)$	Does AIMA.org employ Richard?
$Employs(x, Richard)$	Who employs Richard?
$Employs(AIMA.org, y)$	Whom does AIMA.org employ?
$Employs(x, y)$	Who employs whom?

SUBSUMPTION LATTICE

These queries form a **subsumption lattice**, as shown in Figure 9.2(a). The lattice has some interesting properties. For example, the child of any node in the lattice is obtained from its parent by a single substitution; and the "highest" common descendant of any two nodes is the result of applying their most general unifier. The portion of the lattice above any ground fact can be constructed systematically (Exercise 9.5). A sentence with repeated constants has a slightly different lattice, as shown in Figure 9.2(b). Function symbols and variables in the sentences to be stored introduce still more interesting lattice structures.

The scheme we have described works very well whenever the lattice contains a small number of nodes. For a predicate with n arguments, the lattice contains $O(2^n)$ nodes. If function symbols are allowed, the number of nodes is also exponential in the size of the terms in the sentence to be stored. This can lead to a huge number of indices. At some point, the benefits of indexing are outweighed by the costs of storing and maintaining all the indices. We can respond by adopting a fixed policy, such as maintaining indices only on keys composed of a predicate plus each argument, or by using an adaptive policy that creates indices to meet the demands of the kinds of queries being asked. For most AI systems, the number of facts to be stored is small enough that efficient indexing is considered a solved problem. For industrial and commercial databases, the problem has received substantial technology development.

[2] A hash table is a data structure for storing and retrieving information indexed by fixed *keys*. For practical purposes, a hash table can be considered to have constant storage and retrieval times, even when the table contains a very large number of items.

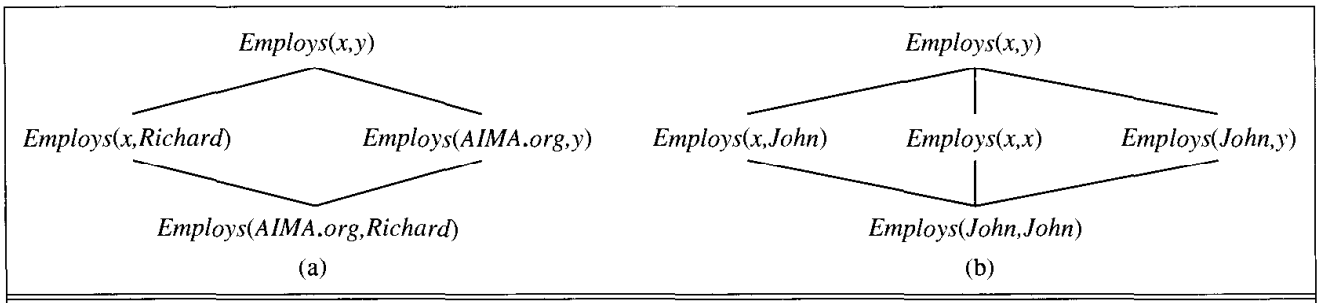


Figure 9.2 (a) The subsumption lattice whose lowest node is the sentence $Employs(AIMA.org, Richard)$. (b) The subsumption lattice for the sentence $Employs(John, John)$.

9.3 Forward Chaining

A forward-chaining algorithm for propositional definite clauses was given in Section 7.5. The idea is simple: start with the atomic sentences in the knowledge base and apply Modus Ponens in the forward direction, adding new atomic sentences, until no further inferences can be made. Here, we explain how the algorithm is applied to first-order definite clauses and how it can be implemented efficiently. Definite clauses such as $Situation \Rightarrow Response$ are especially useful for systems that make inferences in response to newly arrived information. Many systems can be defined this way, and reasoning with forward chaining can be much more efficient than resolution theorem proving. Therefore it is often worthwhile to try to build a knowledge base using only definite clauses so that the cost of resolution can be avoided.

First-order definite clauses

First-order definite clauses closely resemble propositional definite clauses (page 217): they are disjunctions of literals of which *exactly one is positive*. A definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. The following are first-order definite clauses:

$$King(x) \wedge Greedy(x) \Rightarrow Evil(x) .$$
$$King(John) .$$
$$Greedy(y) .$$

Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified. (Typically, we omit universal quantifiers when writing definite clauses.) Definite clauses are a suitable normal form for use with Generalized Modus Ponens.

Not every knowledge base can be converted into a set of definite clauses, because of the single-positive-literal restriction, but many can. Consider the following problem:

> The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

We will prove that West is a criminal. First, we will represent these facts as first-order definite clauses. The next section shows how the forward-chaining algorithm solves the problem.

"... it is a crime for an American to sell weapons to hostile nations":

$$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x) . \quad (9.3)$$

"Nono ... has some missiles." The sentence $\exists x\ Owns(Nono, x) \wedge Missile(x)$ is transformed into two definite clauses by Existential Elimination, introducing a new constant M_1:

$$Owns(Nono, M_1) \quad (9.4)$$

$$Missile(M_1) \quad (9.5)$$

"All of its missiles were sold to it by Colonel West":

$$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono) . \quad (9.6)$$

We will also need to know that missiles are weapons:

$$Missile(x) \Rightarrow Weapon(x) \quad (9.7)$$

and we must know that an enemy of America counts as "hostile":

$$Enemy(x, America) \Rightarrow Hostile(x) . \quad (9.8)$$

"West, who is American ...":

$$American(West) . \quad (9.9)$$

"The country Nono, an enemy of America ...":

$$Enemy(Nono, America) . \quad (9.10)$$

This knowledge base contains no function symbols and is therefore an instance of the class of **Datalog** knowledge bases—that is, sets of first-order definite clauses with no function symbols. We will see that the absence of function symbols makes inference much easier.

DATALOG

A simple forward-chaining algorithm

The first forward chaining algorithm we will consider is a very simple one, as shown in Figure 9.3. Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions to the known facts. The process repeats until the query is answered (assuming that just one answer is required) or no new facts are added. Notice that a fact is not "new" if it is just a **renaming** of a known fact. One sentence is a renaming of another if they are identical except for the names of the variables. For example, $Likes(x, IceCream)$ and $Likes(y, IceCream)$ are renamings of each other because they differ only in the choice of x or y; their meanings are identical: everyone likes ice cream.

RENAMING

We will use our crime problem to illustrate how FOL-FC-ASK works. The implication sentences are (9.3), (9.6), (9.7), and (9.8). Two iterations are required:

- On the first iteration, rule (9.3) has unsatisfied premises.
  Rule (9.6) is satisfied with $\{x/M_1\}$, and $Sells(West, M_1, Nono)$ is added.
  Rule (9.7) is satisfied with $\{x/M_1\}$, and $Weapon(M_1)$ is added.
  Rule (9.8) is satisfied with $\{x/Nono\}$, and $Hostile(Nono)$ is added.

```
function FOL-FC-ASK(KB, α) returns a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
          α, the query, an atomic sentence
  local variables: new, the new sentences inferred on each iteration

  repeat until new is empty
    new ← { }
    for each sentence r in KB do
      (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-APART(r)
      for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
                  for some p'1, ..., p'n in KB
        q' ← SUBST(θ, q)
        if q' is not a renaming of some sentence already in KB or new then do
          add q' to new
          φ ← UNIFY(q', α)
          if φ is not fail then return φ
    add new to KB
  return false
```

Figure 9.3 A conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB.

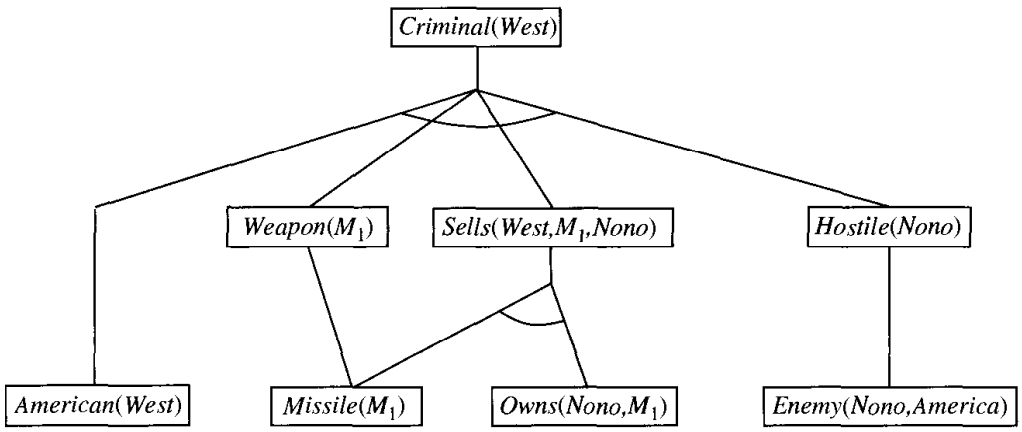


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

- On the second iteration, rule (9.3) is satisfied with $\{x/West, y/M_1, z/Nono\}$, and $Criminal(West)$ is added.

Figure 9.4 shows the proof tree that is generated. Notice that no new inferences are possible at this point because every sentence that could be concluded by forward chaining is already contained explicitly in the KB. Such a knowledge base is called a **fixed point** of the inference process. Fixed points reached by forward chaining with first-order definite clauses are similar

to those for propositional forward chaining (page 219); the principal difference is that a first-order fixed point can include universally quantified atomic sentences.

FOL-FC-ASK is easy to analyze. First, it is **sound**, because every inference is just an application of Generalized Modus Ponens, which is sound. Second, it is **complete** for definite clause knowledge bases; that is, it answers every query whose answers are entailed by any knowledge base of definite clauses. For Datalog knowledge bases, which contain no function symbols, the proof of completeness is fairly easy. We begin by counting the number of possible facts that can be added, which determines the maximum number of iterations. Let k be the maximum **arity** (number of arguments) of any predicate, p be the number of predicates, and n be the number of constant symbols. Clearly, there can be no more than pn^k distinct ground facts, so after this many iterations the algorithm must have reached a fixed point. Then we can make an argument very similar to the proof of completeness for propositional forward chaining. (See page 219.) The details of how to make the transition from propositional to first-order completeness are given for the resolution algorithm in Section 9.5.

For general definite clauses with function symbols, FOL-FC-ASK can generate infinitely many new facts, so we need to be more careful. For the case in which an answer to the query sentence q is entailed, we must appeal to Herbrand's theorem to establish that the algorithm will find a proof. (See Section 9.5 for the resolution case.) If the query has no answer, the algorithm could fail to terminate in some cases. For example, if the knowledge base includes the Peano axioms

$$NatNum(0)$$
$$\forall n \;\; NatNum(n) \Rightarrow NatNum(S(n))$$

then forward chaining adds $NatNum(S(0))$, $NatNum(S(S(0)))$, $NatNum(S(S(S(0))))$, and so on. This problem is unavoidable in general. As with general first-order logic, entailment with definite clauses is semidecidable.

Efficient forward chaining

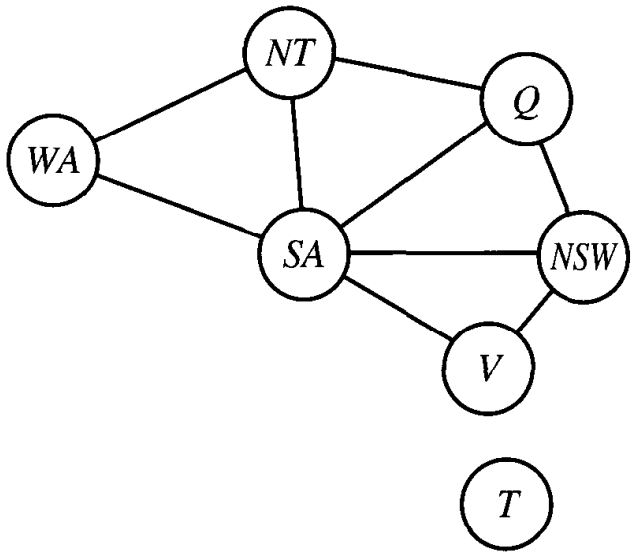
The forward chaining algorithm in Figure 9.3 is designed for ease of understanding rather than for efficiency of operation. There are three possible sources of complexity. First, the "inner loop" of the algorithm involves finding all possible unifiers such that the premise of a rule unifies with a suitable set of facts in the knowledge base. This is often called **pattern matching** and can be very expensive. Second, the algorithm rechecks every rule on every iteration to see whether its premises are satisfied, even if very few additions are made to the knowledge base on each iteration. Finally, the algorithm might generate many facts that are irrelevant to the goal. We will address each of these sources in turn.

PATTERN MATCHING

Matching rules against known facts

The problem of matching the premise of a rule against the facts in the knowledge base might seem simple enough. For example, suppose we want to apply the rule

$$Missile(x) \Rightarrow Weapon(x) \,.$$



(a)

$$Diff(wa, nt) \wedge Diff(wa, sa) \wedge$$
$$Diff(nt, q) \wedge Diff(nt, sa) \wedge$$
$$Diff(q, nsw) \wedge Diff(q, sa) \wedge$$
$$Diff(nsw, v) \wedge Diff(nsw, sa) \wedge$$
$$Diff(v, sa) \Rightarrow Colorable()$$

$$Diff(Red, Blue) \quad Diff(Red, Green)$$
$$Diff(Green, Red) \quad Diff(Green, Blue)$$
$$Diff(Blue, Red) \quad Diff(Blue, Green)$$

(b)

Figure 9.5 (a) Constraint graph for coloring the map of Australia (from Figure5.1). (b) The map-coloring CSP expressed as a single definite clause. Note that the domains of the variables are defined implicitly by the constants given in the ground facts for *Diff*.

Then we need to find all the facts that unify with $Missile(x)$; in a suitably indexed knowledge base, this can be done in constant time per fact. Now consider a rule such as

$$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono) .$$

Again, we can find all the objects owned by Nono in constant time per object; then, for each object, we could check whether it is a missile. If the knowledge base contains many objects owned by Nono and very few missiles, however, it would be better to find all the missiles first and then check whether they are owned by Nono. This is the **conjunct ordering** problem: find an ordering to solve the conjuncts of the rule premise so that the total cost is minimized. It turns out that finding the optimal ordering is NP-hard, but good heuristics are available. For example, the **most constrained variable** heuristic used for CSPs in Chapter 5 would suggest ordering the conjuncts to look for missiles first if there are fewer missiles than objects that are owned by Nono.

CONJUNCT ORDERING

The connection between pattern matching and constraint satisfaction is actually very close. We can view each conjunct as a constraint on the variables that it contains—for example, $Missile(x)$ is a unary constraint on x. Extending this idea, *we can express every finite-domain CSP as a single definite clause together with some associated ground facts.* Consider the map-coloring problem from Figure 5.1, shown again in Figure 9.5(a). An equivalent formulation as a single definite clause is given in Figure 9.5(b). Clearly, the conclusion $Colorable()$ can be inferred only if the CSP has a solution. Because CSPs in general include 3SAT problems as special cases, we can conclude that *matching a definite clause against a set of facts is NP-hard.*

It might seem rather depressing that forward chaining has an NP-hard matching problem in its inner loop. There are three ways to cheer ourselves up:

- We can remind ourselves that most rules in real-world knowledge bases are small and simple (like the rules in our crime example) rather than large and complex (like the CSP formulation in Figure 9.5). It is common in the database world to assume that both the sizes of rules and the arities of predicates are bounded by a constant and to worry only about **data complexity**—that is, the complexity of inference as a function of the number of ground facts in the database. It is easy to show that the data complexity of forward chaining is polynomial.

DATA COMPLEXITY

- We can consider subclasses of rules for which matching is efficient. Essentially every Datalog clause can be viewed as defining a CSP, so matching will be tractable just when the corresponding CSP is tractable. Chapter 5 describes several tractable families of CSPs. For example, if the constraint graph (the graph whose nodes are variables and whose links are constraints) forms a tree, then the CSP can be solved in linear time. Exactly the same result holds for rule matching. For instance, if we remove South Australia from the map in Figure 9.5, the resulting clause is

$$Diff(wa, nt) \wedge Diff(nt, q) \wedge Diff(q, nsw) \wedge Diff(nsw, v) \Rightarrow Colorable()$$

  which corresponds to the reduced CSP shown in Figure 5.11. Algorithms for solving tree-structured CSPs can be applied directly to the problem of rule matching.
- We can work hard to eliminate redundant rule matching attempts in the forward chaining algorithm, which is the subject of the next section.

Incremental forward chaining

When we showed how forward chaining works on the crime example, we cheated; in particular, we omitted some of the rule matching done by the algorithm shown in Figure 9.3. For example, on the second iteration, the rule

$$Missile(x) \Rightarrow Weapon(x)$$

matches against $Missile(M_1)$ (again), and of course the conclusion $Weapon(M_1)$ is already known so nothing happens. Such redundant rule matching can be avoided if we make the following observation: *Every new fact inferred on iteration t must be derived from at least one new fact inferred on iteration $t-1$.* This is true because any inference that does not require a new fact from iteration $t-1$ could have been done at iteration $t-1$ already.

This observation leads naturally to an incremental forward chaining algorithm where, at iteration t, we check a rule only if its premise includes a conjunct p_i that unifies with a fact p_i' newly inferred at iteration $t-1$. The rule matching step then fixes p_i to match with p_i', but allows the other conjuncts of the rule to match with facts from any previous iteration. This algorithm generates exactly the same facts at each iteration as the algorithm in Figure 9.3, but is much more efficient.

With suitable indexing, it is easy to identify all the rules that can be triggered by any given fact, and indeed many real systems operate in an "update" mode wherein forward chaining occurs in response to each new fact that is TELLed to the system. Inferences cascade through the set of rules until the fixed point is reached, and then the process begins again for the next new fact.

Typically, only a small fraction of the rules in the knowledge base are actually triggered by the addition of a given fact. This means that a great deal of redundant work is done in constructing partial matches repeatedly that have some unsatisfied premises. Our crime example is rather too small to show this effectively, but notice that a partial match is constructed on the first iteration between the rule

$$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$$

and the fact $American(West)$. This partial match is then discarded and rebuilt on the second iteration (when the rule succeeds). It would be better to retain and gradually complete the partial matches as new facts arrive, rather than discarding them.

RETE

The **rete** algorithm[3] was the first to address this problem seriously. The algorithm preprocesses the set of rules in the knowledge base to construct a sort of dataflow network in which each node is a literal from a rule premise. Variable bindings flow through the network and are filtered out when they fail to match a literal. If two literals in a rule share a variable—for example, $Sells(x, y, z) \wedge Hostile(z)$ in the crime example—then the bindings from each literal are filtered through an equality node. A variable binding reaching a node for an n-ary literal such as $Sells(x, y, z)$ might have to wait for bindings for the other variables to be established before the process can continue. At any given point, the state of a rete network captures all the partial matches of the rules, avoiding a great deal of recomputation.

PRODUCTION SYSTEMS

Rete networks, and various improvements thereon, have been a key component of so-called **production systems**, which were among the earliest forward chaining systems in widespread use.[4] The XCON system (originally called R1, McDermott, 1982) was built using a production system architecture. XCON contained several thousand rules for designing configurations of computer components for customers of the Digital Equipment Corporation. It was one of the first clear commercial successes in the emerging field of expert systems. Many other similar systems have been built using the same underlying technology, which has been implemented in the general-purpose language OPS-5.

COGNITIVE ARCHITECTURES

Production systems are also popular in **cognitive architectures**—that is, models of human reasoning—such as ACT (Anderson, 1983) and SOAR (Laird *et al.*, 1987). In such systems, the "working memory" of the system models human short-term memory, and the productions are part of long-term memory. On each cycle of operation, productions are matched against the working memory of facts. A production whose conditions are satisfied can add or delete facts in working memory. In contrast to the typical situation in databases, production systems often have many rules and relatively few facts. With suitably optimized matching technology, some modern systems can operate in real time with over a million rules.

Irrelevant facts

The final source of inefficiency in forward chaining appears to be intrinsic to the approach and also arises in the propositional context. (See Section 7.5.) Forward chaining makes all allowable inferences based on the known facts, *even if they are irrelevant to the goal at hand.* In our crime example, there were no rules capable of drawing irrelevant conclusions,

[3] Rete is Latin for net. The English pronunciation rhymes with treaty.

[4] The word **production** in **production systems** denotes a condition–action rule.

so the lack of directedness was not a problem. In other cases (e.g., if we have several rules describing the eating habits of Americans and the prices of missiles), FOL-FC-ASK will generate many irrelevant conclusions.

One way to avoid drawing irrelevant conclusions is to use backward chaining, as described in Section 9.4. Another solution is to restrict forward chaining to a selected subset of rules; this approach was discussed in the propositional context. A third approach has emerged in the deductive database community, where forward chaining is the standard tool. The idea is to rewrite the rule set, using information from the goal, so that only relevant variable bindings—those belonging to a so-called **magic set**—are considered during forward inference. For example, if the goal is $Criminal(West)$, the rule that concludes $Criminal(x)$ will be rewritten to include an extra conjunct that constrains the value of x:

MAGIC SET

$$Magic(x) \wedge American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x) .$$

The fact $Magic(West)$ is also added to the KB. In this way, even if the knowledge base contains facts about millions of Americans, only Colonel West will be considered during the forward inference process. The complete process for defining magic sets and rewriting the knowledge base is too complex to go into here, but the basic idea is to perform a sort of "generic" backward inference from the goal in order to work out which variable bindings need to be constrained. The magic sets approach can therefore be thought of as a kind of hybrid between forward inference and backward preprocessing.

9.4 BACKWARD CHAINING

The second major family of logical inference algorithms uses the **backward chaining** approach introduced in Section 7.5. These algorithms work backward from the goal, chaining through rules to find known facts that support the proof. We describe the basic algorithm, and then we describe how it is used in **logic programming**, which is the most widely used form of automated reasoning. We will also see that backward chaining has some disadvantages compared with forward chaining, and we look at ways to overcome them. Finally, we will look at the close connection between logic programming and constraint satisfaction problems.

A backward chaining algorithm

Figure 9.6 shows a simple backward-chaining algorithm, FOL-BC-ASK. It is called with a list of goals containing a single element, the original query, and returns the set of all substitutions satisfying the query. The list of goals can be thought of as a "stack" waiting to be worked on; if *all* of them can be satisfied, then the current branch of the proof succeeds. The algorithm takes the first goal in the list and finds every clause in the knowledge base whose positive literal, or **head**, unifies with the goal. Each such clause creates a new recursive call in which the premise, or **body**, of the clause is added to the goal stack. Remember that facts are clauses with a head but no body, so when a goal unifies with a known fact, no new subgoals are added to the stack and the goal is solved. Figure 9.7 is the proof tree for deriving $Criminal(West)$ from sentences (9.3) through (9.10).

function FOL-BC-ASK(KB, $goals$, θ) **returns** a set of substitutions
  inputs: KB, a knowledge base
    $goals$, a list of conjuncts forming a query (θ already applied)
    θ, the current substitution, initially the empty substitution $\{\}$
  local variables: $answers$, a set of substitutions, initially empty

  if $goals$ is empty **then return** $\{\theta\}$
  $q' \leftarrow$ SUBST(θ, FIRST($goals$))
  for each sentence r **in** KB where STANDARDIZE-APART(r) = $(p_1 \wedge \ldots \wedge p_n \Rightarrow q)$
      and $\theta' \leftarrow$ UNIFY(q, q') succeeds
    $new\_goals \leftarrow [p_1, \ldots, p_n | \text{REST}(goals)]$
    $answers \leftarrow$ FOL-BC-ASK(KB, new_goals, COMPOSE(θ', θ)) $\cup$ $answers$
  return $answers$

Figure 9.6 A simple backward-chaining algorithm.

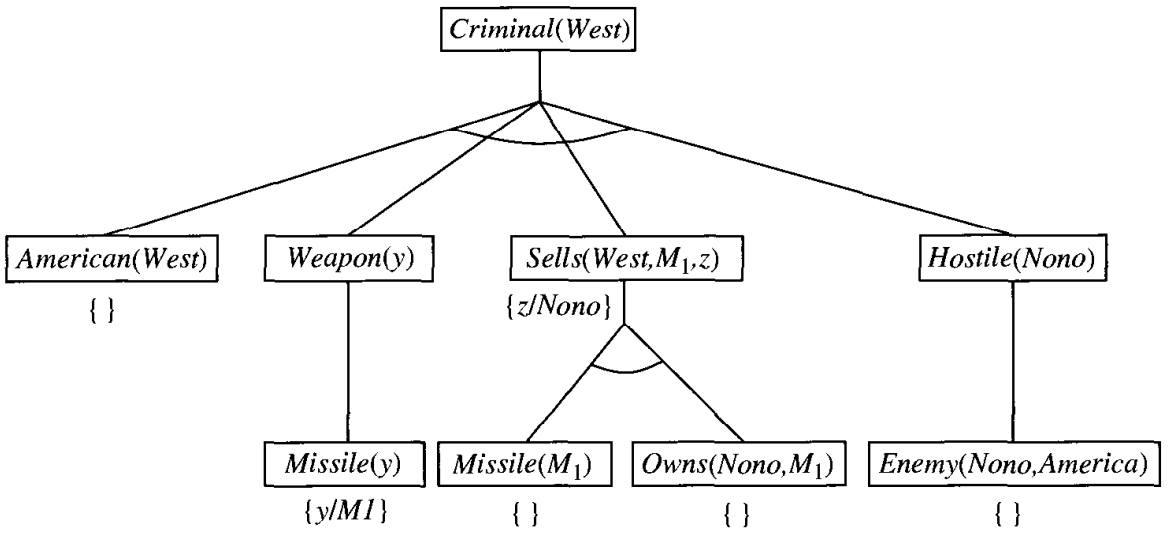


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove $Criminal(West)$, we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally $Hostile(z)$, z is already bound to $Nono$.

COMPOSITION

The algorithm uses **composition** of substitutions. COMPOSE(θ_1, θ_2) is the substitution whose effect is identical to the effect of applying each substitution in turn. That is,

$$\text{SUBST}(\text{COMPOSE}(\theta_1, \theta_2), p) = \text{SUBST}(\theta_2, \text{SUBST}(\theta_1, p)) .$$

In the algorithm, the current variable bindings, which are stored in θ, are composed with the bindings resulting from unifying the goal with the clause head, giving a new set of current bindings for the recursive call.

Backward chaining, as we have written it, is clearly a depth-first search algorithm. This means that its space requirements are linear in the size of the proof (neglecting, for now, the space required to accumulate the solutions). It also means that backward chaining (unlike forward chaining) suffers from problems with repeated states and incompleteness. We will discuss these problems and some potential solutions, but first we will see how backward chaining is used in logic programming systems.

Logic programming

Logic programming is a technology that comes fairly close to embodying the declarative ideal described in Chapter 7: that systems should be constructed by expressing knowledge in a formal language and that problems should be solved by running inference processes on that knowledge. The ideal is summed up in Robert Kowalski's equation,

$$Algorithm = Logic + Control\ .$$

PROLOG

Prolog is by far the most widely used logic programming language. Its users number in the hundreds of thousands. It is used primarily as a rapid-prototyping language and for symbol-manipulation tasks such as writing compilers (Van Roy, 1990) and parsing natural language (Pereira and Warren, 1980). Many expert systems have been written in Prolog for legal, medical, financial, and other domains.

Prolog programs are sets of definite clauses written in a notation somewhat different from standard first-order logic. Prolog uses uppercase letters for variables and lowercase for constants. Clauses are written with the head preceding the body; "`:-`" is used for left-implication, commas separate literals in the body, and a period marks the end of a sentence:

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
```

Prolog includes "syntactic sugar" for list notation and arithmetic. As an example, here is a Prolog program for `append(X,Y,Z)`, which succeeds if list `Z` is the result of appending lists `X` and `Y`:

```
append([],Y,Y).
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

In English, we can read these clauses as (1) appending an empty list with a list `Y` produces the same list `Y` and (2) `[A|Z]` is the result of appending `[A|X]` onto `Y`, provided that `Z` is the result of appending `X` onto `Y`. This definition of `append` appears fairly similar to the corresponding definition in Lisp, but is actually much more powerful. For example, we can ask the query `append(A,B,[1,2])`: what two lists can be appended to give `[1,2]`? We get back the solutions

```
A=[]     B=[1,2]
A=[1]    B=[2]
A=[1,2]  B=[]
```

The execution of Prolog programs is done via depth-first backward chaining, where clauses are tried in the order in which they are written in the knowledge base. Some aspects of Prolog fall outside standard logical inference:

- There is a set of built-in functions for arithmetic. Literals using these function symbols are "proved" by executing code rather than doing further inference. For example, the goal "`X is 4+3`" succeeds with `X` bound to 7. On the other hand, the goal "`5 is X+Y`" fails, because the built-in functions do not do arbitrary equation solving.[5]
- There are built-in predicates that have side effects when executed. These include input–output predicates and the `assert/retract` predicates for modifying the knowledge base. Such predicates have no counterpart in logic and can produce some confusing effects—for example, if facts are asserted in a branch of the proof tree that eventually fails.
- Prolog allows a form of negation called **negation as failure**. A negated goal `not P` is considered proved if the system fails to prove `P`. Thus, the sentence

  ```
  alive(X) :- not dead(X).
  ```

  can be read as "Everyone is alive if not provably dead."
- Prolog has an equality operator, `=`, but it lacks the full power of logical equality. An equality goal succeeds if the two terms are *unifiable* and fails otherwise. So `X+Y=2+3` succeeds with `X` bound to `2` and `Y` bound to `3`, but `morningstar=eveningstar` fails. (In classical logic, the latter equality might or might not be true.) No facts or rules about equality can be asserted.
- The **occur check** is omitted from Prolog's unification algorithm. This means that some unsound inferences can be made; these are seldom a problem except when using Prolog for mathematical theorem proving.

The decisions made in the design of Prolog represent a compromise between declarativeness and execution efficiency—inasmuch as efficiency was understood at the time Prolog was designed. We will return to this subject after looking at how Prolog is implemented.

Efficient implementation of logic programs

The execution of a Prolog program can happen in two modes: interpreted and compiled. Interpretation essentially amounts to running the FOL-BC-ASK algorithm from Figure 9.6, with the program as the knowledge base. We say "essentially," because Prolog interpreters contain a variety of improvements designed to maximize speed. Here we consider only two.

First, instead of constructing the list of all possible answers for each subgoal before continuing to the next, Prolog interpreters generate one answer and a "promise" to generate the rest when the current answer has been fully explored. This promise is called a **choice point**. When the depth-first search completes its exploration of the possible solutions arising from the current answer and backs up to the choice point, the choice point is expanded to yield a new answer for the subgoal and a new choice point. This approach saves both time and space. It also provides a very simple interface for debugging because at all times there is only a single solution path under consideration.

CHOICE POINT

Second, our simple implementation of FOL-BC-ASK spends a good deal of time generating and composing substitutions. Prolog implements substitutions using logic variables

[5] Note that if the Peano axioms are provided, such goals can be solved by inference within a Prolog program.

```
procedure APPEND(ax, y, az, continuation)

  trail ← GLOBAL-TRAIL-POINTER()
  if ax = [] and UNIFY(y, az) then CALL(continuation)
  RESET-TRAIL(trail)
  a ← NEW-VARIABLE(); x ← NEW-VARIABLE(); z ← NEW-VARIABLE()
  if UNIFY(ax, [a | x]) and UNIFY(az, [a | z]) then APPEND(x, y, z, continuation)
```

Figure 9.8 Pseudocode representing the result of compiling the Append predicate. The function NEW-VARIABLE returns a new variable, distinct from all other variables so far used. The procedure CALL(*continuation*) continues execution with the specified continuation.

that can remember their current binding. At any point in time, every variable in the program either is unbound or is bound to some value. Together, these variables and values implicitly define the substitution for the current branch of the proof. Extending the path can only add new variable bindings, because an attempt to add a different binding for an already bound variable results in a failure of unification. When a path in the search fails, Prolog will back up to a previous choice point, and then it might have to unbind some variables. This is done by keeping track of all the variables that have been bound in a stack called the **trail**. As each new variable is bound by UNIFY-VAR, the variable is pushed onto the trail. When a goal fails and it is time to back up to a previous choice point, each of the variables is unbound as it is removed from the trail.

TRAIL

Even the most efficient Prolog interpreters require several thousand machine instructions per inference step because of the cost of index lookup, unification, and building the recursive call stack. In effect, the interpreter always behaves as if it has never seen the program before; for example, it has to *find* clauses that match the goal. A compiled Prolog program, on the other hand, is an inference procedure for a specific set of clauses, so it *knows* what clauses match the goal. Prolog basically generates a miniature theorem prover for each different predicate, thereby eliminating much of the overhead of interpretation. It is also possible to **open-code** the unification routine for each different call, thereby avoiding explicit analysis of term structure. (For details of open-coded unification, see Warren *et al.* (1977).)

OPEN-CODE

The instruction sets of today's computers give a poor match with Prolog's semantics, so most Prolog compilers compile into an intermediate language rather than directly into machine language. The most popular intermediate language is the Warren Abstract Machine, or WAM, named after David H. D. Warren, one of the implementors of the first Prolog compiler. The WAM is an abstract instruction set that is suitable for Prolog and can be either interpreted or translated into machine language. Other compilers translate Prolog into a high-level language such as Lisp or C and then use that language's compiler to translate to machine language. For example, the definition of the Append predicate can be compiled into the code shown in Figure 9.8. There are several points worth mentioning:

- Rather than having to search the knowledge base for Append clauses, the clauses become a procedure and the inferences are carried out simply by calling the procedure.

- As described earlier, the current variable bindings are kept on a trail. The first step of the procedure saves the current state of the trail, so that it can be restored by RESET-TRAIL if the first clause fails. This will undo any bindings generated by the first call to UNIFY.

CONTINUATIONS

- The trickiest part is the use of **continuations** to implement choice points. You can think of a continuation as packaging up a procedure and a list of arguments that together define what should be done next whenever the current goal succeeds. It would not do just to return from a procedure like APPEND when the goal succeeds, because it could succeed in several ways, and each of them has to be explored. The continuation argument solves this problem because it can be called each time the goal succeeds. In the APPEND code, if the first argument is empty, then the APPEND predicate has succeeded. We then CALL the continuation, with the appropriate bindings on the trail, to do whatever should be done next. For example, if the call to APPEND were at the top level, the continuation would print the bindings of the variables.

Before Warren's work on the compilation of inference in Prolog, logic programming was too slow for general use. Compilers by Warren and others allowed Prolog code to achieve speeds that are competitive with C on a variety of standard benchmarks (Van Roy, 1990). Of course, the fact that one can write a planner or natural language parser in a few dozen lines of Prolog makes it somewhat more desirable than C for prototyping most small-scale AI research projects.

Parallelization can also provide substantial speedup. There are two principal sources of

OR-PARALLELISM

parallelism. The first, called **OR-parallelism**, comes from the possibility of a goal unifying with many different clauses in the knowledge base. Each gives rise to an independent branch in the search space that can lead to a potential solution, and all such branches can be solved

AND-PARALLELISM

in parallel. The second, called **AND-parallelism**, comes from the possibility of solving each conjunct in the body of an implication in parallel. AND-parallelism is more difficult to achieve, because solutions for the whole conjunction require consistent bindings for all the variables. Each conjunctive branch must communicate with the other branches to ensure a global solution.

Redundant inference and infinite loops

We now turn to the Achilles heel of Prolog: the mismatch between depth-first search and search trees that include repeated states and infinite paths. Consider the following logic program that decides if a path exists between two points on a directed graph:

```
path(X,Z) :- link(X,Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```

A simple three-node graph, described by the facts `link(a,b)` and `link(b,c)`, is shown in Figure 9.9(a). With this program, the query `path(a,c)` generates the proof tree shown in Figure 9.10(a). On the other hand, if we put the two clauses in the order

```
path(X,Z) :- path(X,Y), link(Y,Z).
path(X,Z) :- link(X,Z).
```

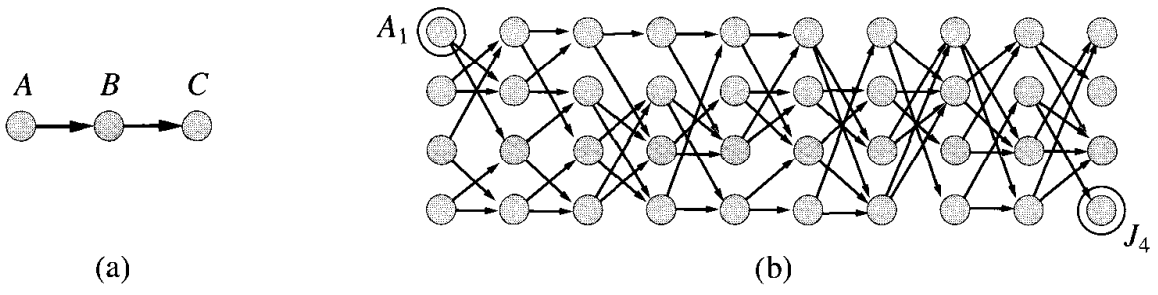


Figure 9.9 (a) Finding a path from A to C can lead Prolog into an infinite loop. (b) A graph in which each node is connected to two random successors in the next layer. Finding a path from A_1 to J_4 requires 877 inferences.

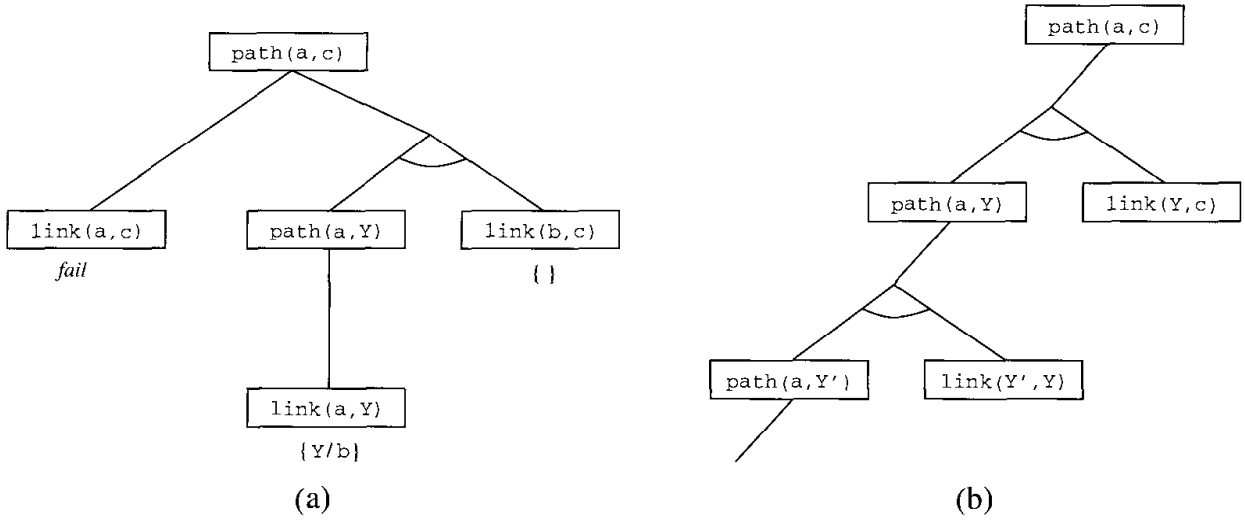


Figure 9.10 (a) Proof that a path exists from A to C. (b) Infinite proof tree generated when the clauses are in the "wrong" order.

then Prolog follows the infinite path shown in Figure 9.10(b). Prolog is therefore **incomplete** as a theorem prover for definite clauses—even for Datalog programs, as this example shows—because, for some knowledge bases, it fails to prove sentences that are entailed. Notice that forward chaining does not suffer from this problem: once `path(a,b)`, `path(b,c)`, and `path(a,c)` are inferred, forward chaining halts.

Depth-first backward chaining also has problems with redundant computations. For example, when finding a path from A_1 to J_4 in Figure 9.9(b), Prolog performs 877 inferences, most of which involve finding all possible paths to nodes from which the goal is unreachable. This is similar to the repeated-state problem discussed in Chapter 3. The total amount of inference can be exponential in the number of ground facts that are generated. If we apply forward chaining instead, at most n^2 `path(X,Y)` facts can be generated linking n nodes. For the problem in Figure 9.9(b), only 62 inferences are needed.

DYNAMIC PROGRAMMING

Forward chaining on graph search problems is an example of **dynamic programming**, in which the solutions to subproblems are constructed incrementally from those of smaller subproblems and are cached to avoid recomputation. We can obtain a similar effect in a backward chaining system using **memoization**—that is, caching solutions to subgoals as they are

found and then reusing those solutions when the subgoal recurs, rather than repeating the previous computation. This is the approach taken by **tabled logic programming** systems, which use efficient storage and retrieval mechanisms to perform memoization. Tabled logic programming combines the goal-directedness of backward chaining with the dynamic programming efficiency of forward chaining. It is also complete for Datalog programs, which means that the programmer need worry less about infinite loops.

TABLED LOGIC PROGRAMMING

Constraint logic programming

In our discussion of forward chaining (Section 9.3), we showed how constraint satisfaction problems (CSPs) can be encoded as definite clauses. Standard Prolog solves such problems in exactly the same way as the backtracking algorithm given in Figure 5.3.

Because backtracking enumerates the domains of the variables, it works only for **finite domain** CSPs. In Prolog terms, there must be a finite number of solutions for any goal with unbound variables. (For example, the goal `diff(q,sa)`, which says that Queensland and South Australia must be different colors, has six solutions if three colors are allowed.) Infinite-domain CSPs—for example with integer or real-valued variables—require quite different algorithms, such as bounds propagation or linear programming.

The following clause succeeds if three numbers satisfy the triangle inequality:

```
triangle(X,Y,Z) :-
    X>=0, Y>=0, Z>=0, X+Y>=Z, Y+Z>=X, X+Z>=Y.
```

If we ask Prolog the query `triangle(3,4,5)`, this works fine. On the other hand, if we ask `triangle(3,4,Z)`, no solution will be found, because the subgoal `Z>=0` cannot be handled by Prolog. The difficulty is that variables in Prolog must be in one of two states: unbound or bound to a particular term.

Binding a variable to a particular term can be viewed as an extreme form of constraint, namely an equality constraint. **Constraint logic programming** (CLP) allows variables to be *constrained* rather than *bound*. A solution to a constraint logic program is the most specific set of constraints on the query variables that can be derived from the knowledge base. For example, the solution to the `triangle(3,4,Z)` query is the constraint `7 >= Z >= 1`. Standard logic programs are just a special case of CLP in which the solution constraints must be equality constraints—that is bindings.

CONSTRAINT LOGIC PROGRAMMING

CLP systems incorporate various constraint-solving algorithms for the constraints allowed in the language. For example, a system that allows linear inequalities on real-valued variables might include a linear programming algorithm for solving those constraints. CLP systems also adopt a much more flexible approach to solving standard logic programming queries. For example, instead of depth-first, left-to-right backtracking, they might use any of the more efficient algorithms discussed in Chapter 5, including heuristic conjunct ordering, backjumping, cutset conditioning, and so on. CLP systems therefore combine elements of constraint satisfaction algorithms, logic programming, and deductive databases.

CLP systems can also take advantage of the variety of CSP search optimizations described in Chapter 5, such as variable and value ordering, forward checking, and intelligent backtracking. Several systems have been defined that allow the programmer more control

over the search order for inference. For example, the MRS Language (Genesereth and Smith, 1981; Russell, 1985) allows the programmer to write **metarules** to determine which conjuncts are tried first. The user could write a rule saying that the goal with the fewest variables should be tried first or could write domain-specific rules for particular predicates.

METARULES

9.5 RESOLUTION

The last of our three families of logical systems is based on **resolution**. We saw in Chapter 7 that propositional resolution is a refutation complete inference procedure for propositional logic. In this section, we will see how to extend resolution to first-order logic.

The question of the existence of complete proof procedures is of direct concern to mathematicians. If a complete proof procedure can be found for mathematical statements, two things follow: first, all conjectures can be established mechanically; second, all of mathematics can be established as the logical consequence of a set of fundamental axioms. The question of completeness has therefore generated some of the most important mathematical work of the 20th century. In 1930, the German mathematician Kurt Gödel proved the first **completeness theorem** for first-order logic, showing that any entailed sentence has a finite proof. (No really *practical* proof procedure was found until J. A. Robinson published the resolution algorithm in 1965.) In 1931, Gödel proved an even more famous **incompleteness theorem**. The theorem states that a logical system that includes the principle of induction—without which very little of discrete mathematics can be constructed—is necessarily incomplete. Hence, there are sentences that are entailed, but have no finite proof within the system. The needle may be in the metaphorical haystack, but no procedure can guarantee that it will be found.

COMPLETENESS THEOREM

INCOMPLETENESS THEOREM

Despite Gödel's theorem, resolution-based theorem provers have been applied widely to derive mathematical theorems, including several for which no proof was known previously. Theorem provers have also been used to verify hardware designs and to generate logically correct programs, among other applications.

Conjunctive normal form for first-order logic

As in the propositional case, first-order resolution requires that sentences be in **conjunctive normal form** (CNF)—that is, a conjunction of clauses, where each clause is a disjunction of literals.[6] Literals can contain variables, which are assumed to be universally quantified. For example, the sentence

$$\forall x \;\; American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$$

becomes, in CNF,

$$\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x, y, z) \vee \neg Hostile(z) \vee Criminal(x) \; .$$

[6] A clause can also be represented as an implication with a conjunction of atoms on the left and a disjunction of atoms on the right, as shown in Exercise 7.12. This form, sometimes called **Kowalski form** when written with a right-to-left implication symbol (Kowalski, 1979b), is often much easier to read.

Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence. In particular, the CNF sentence will be unsatisfiable just when the original sentence is unsatisfiable, so we have a basis for doing proofs by contradiction on the CNF sentences.

The procedure for conversion to CNF is very similar to the propositional case, which we saw on page 215. The principal difference arises from the need to eliminate existential quantifiers. We will illustrate the procedure by translating the sentence "Everyone who loves all animals is loved by someone," or

$$\forall x \ [\forall y \ \mathit{Animal}(y) \Rightarrow \mathit{Loves}(x, y)] \Rightarrow [\exists y \ \mathit{Loves}(y, x)] \ .$$

The steps are as follows:

◇ **Eliminate implications**:

$$\forall x \ [\neg \forall y \ \neg \mathit{Animal}(y) \vee \mathit{Loves}(x, y)] \vee [\exists y \ \mathit{Loves}(y, x)] \ .$$

◇ **Move $\neg$ inwards**: In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have

$$\begin{array}{lll} \neg \forall x \ p & \text{becomes} & \exists x \ \neg p \\ \neg \exists x \ p & \text{becomes} & \forall x \ \neg p \ . \end{array}$$

Our sentence goes through the following transformations:

$$\begin{array}{l} \forall x \ [\exists y \ \neg(\neg \mathit{Animal}(y) \vee \mathit{Loves}(x, y))] \vee [\exists y \ \mathit{Loves}(y, x)] \ . \\ \forall x \ [\exists y \ \neg\neg \mathit{Animal}(y) \wedge \neg \mathit{Loves}(x, y)] \vee [\exists y \ \mathit{Loves}(y, x)] \ . \\ \forall x \ [\exists y \ \mathit{Animal}(y) \wedge \neg \mathit{Loves}(x, y)] \vee [\exists y \ \mathit{Loves}(y, x)] \ . \end{array}$$

Notice how a universal quantifier ($\forall y$) in the premise of the implication has become an existential quantifier. The sentence now reads "Either there is some animal that x doesn't love, or (if this is not the case) someone loves x." Clearly, the meaning of the original sentence has been preserved.

◇ **Standardize variables**: For sentences like $(\forall x \ P(x)) \vee (\exists x \ Q(x))$ which use the same variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers. Thus, we have

$$\forall x \ [\exists y \ \mathit{Animal}(y) \wedge \neg \mathit{Loves}(x, y)] \vee [\exists z \ \mathit{Loves}(z, x)] \ .$$

SKOLEMIZATION

◇ **Skolemize**: **Skolemization** is the process of removing existential quantifiers by elimination. In the simple case, it is just like the Existential Instantiation rule of Section 9.1: translate $\exists x \ P(x)$ into $P(A)$, where A is a new constant. If we apply this rule to our sample sentence, however, we obtain

$$\forall x \ [\mathit{Animal}(A) \wedge \neg \mathit{Loves}(x, A)] \vee \mathit{Loves}(B, x)$$

which has the wrong meaning entirely: it says that everyone either fails to love a particular animal A or is loved by some particular entity B. In fact, our original sentence allows each person to fail to love a different animal or to be loved by a different person. Thus, we want the Skolem entities to depend on x:

$$\forall x \ [\mathit{Animal}(F(x)) \wedge \neg \mathit{Loves}(x, F(x))] \vee \mathit{Loves}(G(x), x) \ .$$

SKOLEM FUNCTION

Here F and G are **Skolem functions**. The general rule is that the arguments of the

Skolem function are all the universally quantified variables in whose scope the existential quantifier appears. As with Existential Instantiation, the Skolemized sentence is satisfiable exactly when the original sentence is satisfiable.

◇ **Drop universal quantifiers**: At this point, all remaining variables must be universally quantified. Moreover, the sentence is equivalent to one in which all the universal quantifiers have been moved to the left. We can therefore drop the universal quantifiers:

$$[Animal(F(x)) \wedge \neg Loves(x, F(x))] \vee Loves(G(x), x) .$$

◇ **Distribute $\vee$ over $\wedge$**:

$$[Animal(F(x)) \vee Loves(G(x), x)] \wedge [\neg Loves(x, F(x)) \vee Loves(G(x), x)] .$$

This step may also require flattening out nested conjunctions and disjunctions.

The sentence is now in CNF and consists of two clauses. It is quite unreadable. (It may help to explain that the Skolem function $F(x)$ refers to the animal potentially unloved by x, whereas $G(x)$ refers to someone who might love x.) Fortunately, humans seldom need look at CNF sentences—the translation process is easily automated.

The resolution inference rule

The resolution rule for first-order clauses is simply a lifted version of the propositional resolution rule given on page 214. Two clauses, which are assumed to be standardized apart so that they share no variables, can be resolved if they contain complementary literals. Propositional literals are complementary if one is the negation of the other; first-order literals are complementary if one *unifies with* the negation of the other. Thus we have

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\text{SUBST}(\theta, \ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

where $\text{UNIFY}(\ell_i, \neg m_j) = \theta$. For example, we can resolve the two clauses

$$[Animal(F(x)) \vee Loves(G(x), x)] \quad \text{and} \quad [\neg Loves(u, v) \vee \neg Kills(u, v)]$$

by eliminating the complementary literals $Loves(G(x), x)$ and $\neg Loves(u, v)$, with unifier $\theta = \{u/G(x), v/x\}$, to produce the **resolvent** clause

$$[Animal(F(x)) \vee \neg Kills(G(x), x)] .$$

BINARY RESOLUTION

The rule we have just given is the **binary resolution** rule, because it resolves exactly two literals. The binary resolution rule by itself does not yield a complete inference procedure. The full resolution rule resolves subsets of literals in each clause that are unifiable. An alternative approach is to extend **factoring**—the removal of redundant literals—to the first-order case. Propositional factoring reduces two literals to one if they are *identical*; first-order factoring reduces two literals to one if they are *unifiable*. The unifier must be applied to the entire clause. The combination of binary resolution and factoring is complete.

Example proofs

Resolution proves that $KB \models \alpha$ by proving $KB \wedge \neg\alpha$ unsatisfiable, i.e., by deriving the empty clause. The algorithmic approach is identical to the propositional case, described in

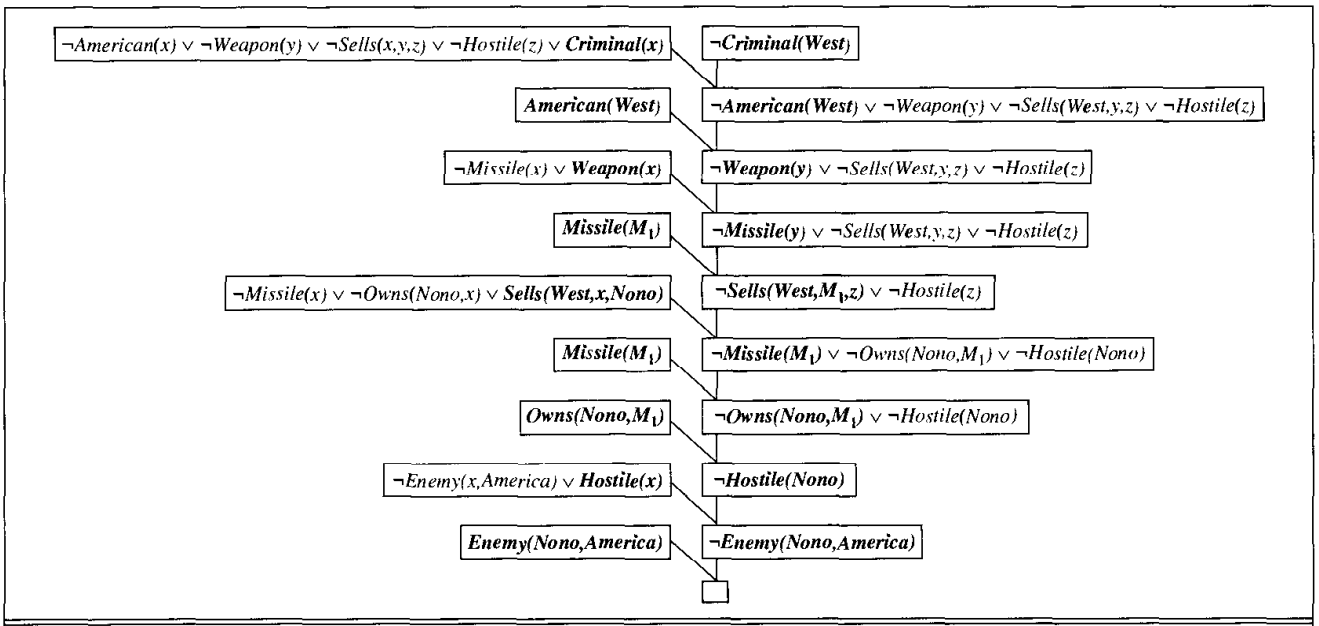


Figure 9.11 A resolution proof that West is a criminal.

Figure 7.12, so we will not repeat it here. Instead, we will give two example proofs. The first is the crime example from Section 9.3. The sentences in CNF are

$$\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x, y, z) \vee \neg Hostile(z) \vee Criminal(x) .$$
$$\neg Missile(x) \vee \neg Owns(Nono, x) \vee Sells(West, x, Nono) .$$
$$\neg Enemy(x, America) \vee Hostile(x) .$$
$$\neg Missile(x) \vee Weapon(x) .$$
$$Owns(Nono, M_1) . \qquad Missile(M_1) .$$
$$American(West) . \qquad Enemy(Nono, America) .$$

We also include the negated goal $\neg Criminal(West)$. The resolution proof is shown in Figure 9.11. Notice the structure: single "spine" beginning with the goal clause, resolving against clauses from the knowledge base until the empty clause is generated. This is characteristic of resolution on Horn clause knowledge bases. In fact, the clauses along the main spine correspond *exactly* to the consecutive values of the *goals* variable in the backward chaining algorithm of Figure 9.6. This is because we always chose to resolve with a clause whose positive literal unified with the leftmost literal of the "current" clause on the spine; this is exactly what happens in backward chaining. Thus, backward chaining is really just a special case of resolution with a particular control strategy to decide which resolution to perform next.

Our second example makes use of Skolemization and involves clauses that are not definite clauses. This results in a somewhat more complex proof structure. In English, the problem is as follows:

> Everyone who loves all animals is loved by someone.
> Anyone who kills an animal is loved by no one.
> Jack loves all animals.
> Either Jack or Curiosity killed the cat, who is named Tuna.
> Did Curiosity kill the cat?

First, we express the original sentences, some background knowledge, and the negated goal G in first-order logic:

A. $\forall x\ [\forall y\ \ Animal(y) \Rightarrow Loves(x, y)] \Rightarrow [\exists y\ \ Loves(y, x)]$

B. $\forall x\ [\exists y\ \ Animal(y) \wedge Kills(x, y)] \Rightarrow [\forall z\ \neg Loves(z, x)]$

C. $\forall x\ \ Animal(x) \Rightarrow Loves(Jack, x)$

D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$

E. $Cat(Tuna)$

F. $\forall x\ \ Cat(x) \Rightarrow Animal(x)$

$\neg$G. $\neg Kills(Curiosity, Tuna)$

Now we apply the conversion procedure to convert each sentence to CNF:

A1. $Animal(F(x)) \vee Loves(G(x), x)$

A2. $\neg Loves(x, F(x)) \vee Loves(G(x), x)$

B. $\neg Animal(y) \vee \neg Kills(x, y) \vee \neg Loves(z, x)$

C. $\neg Animal(x) \vee Loves(Jack, x)$

D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$

E. $Cat(Tuna)$

F. $\neg Cat(x) \vee Animal(x)$

$\neg$G. $\neg Kills(Curiosity, Tuna)$

The resolution proof that Curiosity killed the cat is given in Figure 9.12. In English, the proof could be paraphrased as follows:

> Suppose Curiosity did not kill Tuna. We know that either Jack or Curiosity did; thus Jack must have. Now, Tuna is a cat and cats are animals, so Tuna is an animal. Because anyone who kills an animal is loved by no one, we know that no one loves Jack. On the other hand, Jack loves all animals, so someone loves him; so we have a contradiction. Therefore, Curiosity killed the cat.

The proof answers the question "Did Curiosity kill the cat?" but often we want to pose more general questions, such as "Who killed the cat?" Resolution can do this, but it takes a little

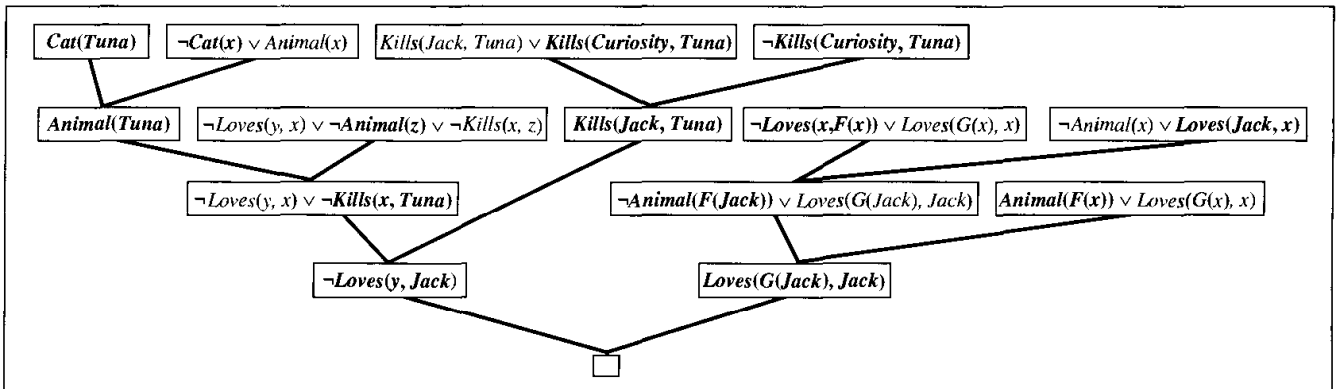


Figure 9.12 A resolution proof that Curiosity killed the cat. Notice the use of factoring in the derivation of the clause $Loves(G(Jack), Jack)$.

more work to obtain the answer. The goal is $\exists w\ Kills(w, Tuna)$, which, when negated, becomes $\neg Kills(w, Tuna)$ in CNF. Repeating the proof in Figure 9.12 with the new negated goal, we obtain a similar proof tree, but with the substitution $\{w/Curiosity\}$ in one of the steps. So, in this case, finding out who killed the cat is just a matter of keeping track of the bindings for the query variables in the proof.

NONCONSTRUCTIVE PROOF

Unfortunately, resolution can produce **nonconstructive proofs** for existential goals. For example, $\neg Kills(w, Tuna)$ resolves with $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$ to give $Kills(Jack, Tuna)$, which resolves again with $\neg Kills(w, Tuna)$ to yield the empty clause. Notice that w has two different bindings in this proof; resolution is telling us that, yes, someone killed Tuna—either Jack or Curiosity. This is no great surprise! One solution is to restrict the allowed resolution steps so that the query variables can be bound only once in a given proof; then we need to be able to backtrack over the possible bindings. Another solution is to add a special **answer literal** to the negated goal, which becomes $\neg Kills(w, Tuna) \vee Answer(w)$. Now, the resolution process generates an answer whenever a clause is generated containing just a *single* answer literal. For the proof in Figure 9.12, this is $Answer(Curiosity)$. The nonconstructive proof would generate the clause $Answer(Curiosity) \vee Answer(Jack)$, which does not constitute an answer.

ANSWER LITERAL

Completeness of resolution

This section gives a completeness proof of resolution. It can be safely skipped by those who are willing to take it on faith.

REFUTATION COMPLETENESS

We will show that resolution is **refutation-complete**, which means that *if* a set of sentences is unsatisfiable, then resolution will always be able to derive a contradiction. Resolution cannot be used to generate all logical consequences of a set of sentences, but it can be used to establish that a given sentence is entailed by the set of sentences. Hence, it can be used to find all answers to a given question, using the negated-goal method that we described earlier in the Chapter.

We will take it as given that any sentence in first-order logic (without equality) can be rewritten as a set of clauses in CNF. This can be proved by induction on the form of the sentence, using atomic sentences as the base case (Davis and Putnam, 1960). Our goal therefore is to prove the following: *if S is an unsatisfiable set of clauses, then the application of a finite number of resolution steps to S will yield a contradiction.*

Our proof sketch follows the original proof due to Robinson, with some simplifications from Genesereth and Nilsson (1987). The basic structure of the proof is shown in Figure 9.13; it proceeds as follows:

1. First, we observe that if S is unsatisfiable, then there exists a particular set of *ground instances* of the clauses of S such that this set is also unsatisfiable (Herbrand's theorem).
2. We then appeal to the **ground resolution theorem** given in Chapter 7, which states that propositional resolution is complete for ground sentences.
3. We then use a **lifting lemma** to show that, for any propositional resolution proof using the set of ground sentences, there is a corresponding first-order resolution proof using the first-order sentences from which the ground sentences were obtained.

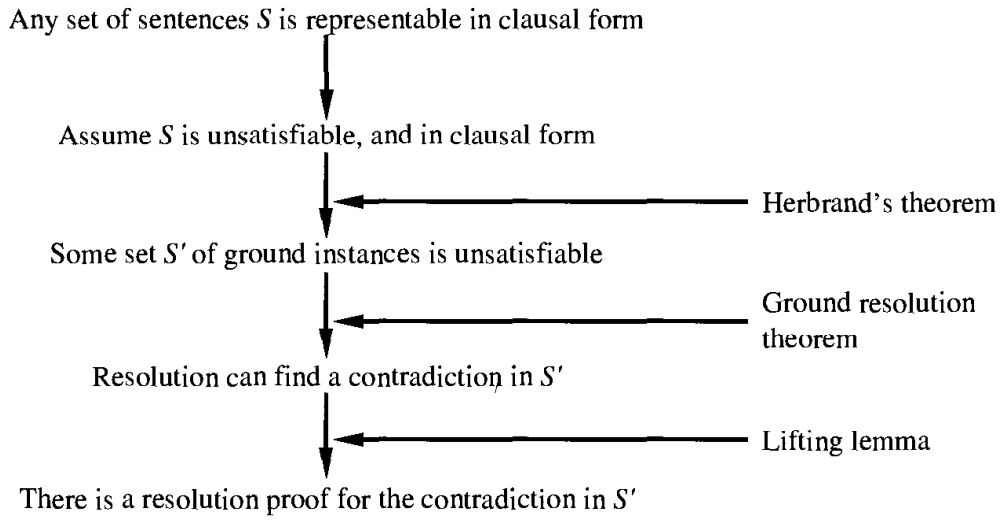


Figure 9.13 Structure of a completeness proof for resolution.

To carry out the first step, we will need three new concepts:

HERBRAND UNIVERSE

◇ **Herbrand universe**: If S is a set of clauses, then H_S, the Herbrand universe of S, is the set of all ground terms constructible from the following:

a. The function symbols in S, if any.
b. The constant symbols in S, if any; if none, then the constant symbol A.

For example, if S contains just the clause $\neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B)$, then H_S is the following infinite set of ground terms:

$$\{A, B, F(A, A), F(A, B), F(B, A), F(B, B), F(A, F(A, A)), \ldots\} .$$

SATURATION

◇ **Saturation**: If S is a set of clauses and P is a set of ground terms, then $P(S)$, the saturation of S with respect to P, is the set of all ground clauses obtained by applying all possible consistent substitutions of ground terms in P with variables in S.

HERBRAND BASE

◇ **Herbrand base**: The saturation of a set S of clauses with respect to its Herbrand universe is called the Herbrand base of S, written as $H_S(S)$. For example, if S contains solely the clause just given, then $H_S(S)$ is the infinite set of clauses

$$\{\neg P(A, F(A, A)) \vee \neg Q(A, A) \vee R(A, B),$$
$$\neg P(B, F(B, A)) \vee \neg Q(B, A) \vee R(B, B),$$
$$\neg P(F(A, A), F(F(A, A), A)) \vee \neg Q(F(A, A), A) \vee R(F(A, A), B),$$
$$\neg P(F(A, B), F(F(A, B), A)) \vee \neg Q(F(A, B), A) \vee R(F(A, B), B), \ldots\}$$

HERBRAND'S THEOREM

These definitions allow us to state a form of **Herbrand's theorem** (Herbrand, 1930):

> If a set S of clauses is unsatisfiable, then there exists a finite subset of $H_S(S)$ that is also unsatisfiable.

Let S' be this finite subset of ground sentences. Now, we can appeal to the ground resolution theorem (page 217) to show that the **resolution closure** $RC(S')$ contains the empty clause. That is, running propositional resolution to completion on S' will derive a contradiction.

Now that we have established that there is always a resolution proof involving some finite subset of the Herbrand base of S, the next step is to show that there is a resolution

GÖDEL'S INCOMPLETENESS THEOREM

By slightly extending the language of first-order logic to allow for the **mathematical induction schema** in arithmetic, Gödel was able to show, in his **incompleteness theorem**, that there are true arithmetic sentences that cannot be proved.

The proof of the incompleteness theorem is somewhat beyond the scope of this book, occupying, as it does, at least 30 pages, but we can give a hint here. We begin with the logical theory of numbers. In this theory, there is a single constant, 0, and a single function, S (the successor function). In the intended model, $S(0)$ denotes 1, $S(S(0))$ denotes 2, and so on; the language therefore has names for all the natural numbers. The vocabulary also includes the function symbols $+$, $\times$, and *Expt* (exponentiation) and the usual set of logical connectives and quantifiers. The first step is to notice that the set of sentences that we can write in this language can be enumerated. (Imagine defining an alphabetical order on the symbols and then arranging, in alphabetical order, each of the sets of sentences of length 1, 2, and so on.) We can then number each sentence α with a unique natural number $\#\alpha$ (the **Gödel number**). This is crucial: number theory contains a name for each of its own sentences. Similarly, we can number each possible proof P with a Gödel number $G(P)$, because a proof is simply a finite sequence of sentences.

Now suppose we have a recursively enumerable set A of sentences that are true statements about the natural numbers. Recalling that A can be named by a given set of integers, we can imagine writing in our language a sentence $\alpha(j, A)$ of the following sort:

> $\forall i$ i is not the Gödel number of a proof of the sentence whose Gödel number is j, where the proof uses only premises in A.

Then let σ be the sentence $\alpha(\#\sigma, A)$, that is, a sentence that states its own unprovability from A. (That this sentence always exists is true but not entirely obvious.)

Now we make the following ingenious argument: Suppose that σ *is* provable from A; then σ is false (because σ says it cannot be proved). But then we have a false sentence that is provable from A, so A cannot consist of only true sentences—a violation of our premise. Therefore σ is *not* provable from A. But this is exactly what σ itself claims; hence σ is a true sentence.

So, we have shown (barring $29\frac{1}{2}$ pages) that for any set of true sentences of number theory, and in particular any set of basic axioms, there are other true sentences that *cannot* be proved from those axioms. This establishes, among other things, that we can never prove all the theorems of mathematics *within any given system of axioms*. Clearly, this was an important discovery for mathematics. Its significance for AI has been widely debated, beginning with speculations by Gödel himself. We take up the debate in Chapter 26.

proof using the clauses of S itself, which are not necessarily ground clauses. We start by considering a single application of the resolution rule. Robinson's basic lemma implies the following fact:

> Let C_1 and C_2 be two clauses with no shared variables, and let C_1' and C_2' be ground instances of C_1 and C_2. If C' is a resolvent of C_1' and C_2', then there exists a clause C such that (1) C is a resolvent of C_1 and C_2 and (2) C' is a ground instance of C.

LIFTING LEMMA

This is called a **lifting lemma**, because it lifts a proof step from ground clauses up to general first-order clauses. In order to prove his basic lifting lemma, Robinson had to invent unification and derive all of the properties of most general unifiers. Rather than repeat the proof here, we simply illustrate the lemma:

$$\begin{aligned}
C_1 &= \neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B) \\
C_2 &= \neg N(G(y), z) \vee P(H(y), z) \\
C_1' &= \neg P(H(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\
C_2' &= \neg N(G(B), F(H(B), A)) \vee P(H(B), F(H(B), A)) \\
C' &= \neg N(G(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\
C &= \neg N(G(y), F(H(y), A)) \vee \neg Q(H(y), A) \vee R(H(y), B) \, .
\end{aligned}$$

We see that indeed C' is a ground instance of C. In general, for C_1' and C_2' to have any resolvents, they must be constructed by first applying to C_1 and C_2 the most general unifier of a pair of complementary literals in C_1 and C_2. From the lifting lemma, it is easy to derive a similar statement about any sequence of applications of the resolution rule:

> For any clause C' in the resolution closure of S' there is a clause C in the resolution closure of S, such that C' is a ground instance of C and the derivation of C is the same length as the derivation of C'.

From this fact, it follows that if the empty clause appears in the resolution closure of S', it must also appear in the resolution closure of S. This is because the empty clause cannot be a ground instance of any other clause. To recap: we have shown that if S is unsatisfiable, then there is a finite derivation of the empty clause using the resolution rule.

The lifting of theorem proving from ground clauses to first-order clauses provides a vast increase in power. This increase comes from the fact that the first-order proof need instantiate variables only as far as necessary for the proof, whereas the ground-clause methods were required to examine a huge number of arbitrary instantiations.

Dealing with equality

None of the inference methods described so far in this chapter handle equality. There are three distinct approaches that can be taken. The first approach is to axiomatize equality—to write down sentences about the equality relation in the knowledge base. We need to say that equality is reflexive, symmetric, and transitive, and we also have to say that we can substitute

equals for equals in any predicate or function. So we need three basic axioms, and then one for each predicate and function:

$$\begin{aligned}
&\forall x \;\; x = x \\
&\forall x, y \;\; x = y \Rightarrow y = x \\
&\forall x, y, z \;\; x = y \wedge y = z \Rightarrow x = z \\
&\forall x, y \;\; x = y \Rightarrow (P_1(x) \Leftrightarrow P_1(y)) \\
&\forall x, y \;\; x = y \Rightarrow (P_2(x) \Leftrightarrow P_2(y)) \\
&\quad \vdots \\
&\forall w, x, y, z \;\; w = y \wedge x = z \Rightarrow (F_1(w, x) = F_1(y, z)) \\
&\forall w, x, y, z \;\; w = y \wedge x = z \Rightarrow (F_2(w, x) = F_2(y, z)) \\
&\quad \vdots
\end{aligned}$$

Given these sentences, a standard inference procedure such as resolution can perform tasks requiring equality reasoning, such as solving mathematical equations.

Another way to deal with equality is with an additional inference rule. The simplest rule, **demodulation**, takes a unit clause $x = y$ and substitutes y for any term that unifies with x in some other clause. More formally, we have

DEMODULATION

◇ **Demodulation**: For any terms x, y, and z, where $\text{UNIFY}(x, z) = \theta$ and $m_n[z]$ is a literal containing z:

$$\frac{x = y, \qquad m_1 \vee \cdots \vee m_n[z]}{m_1 \vee \cdots \vee m_n[\text{SUBST}(\theta, y)]} .$$

Demodulation is typically used for simplifying expressions using collections of assertions such as $x + 0 = x$, $x^1 = x$, and so on. The rule can also be extended to handle non-unit clauses in which an equality literal appears:

PARAMODULATION

◇ **Paramodulation**: For any terms x, y, and z, where $\text{UNIFY}(x, z) = \theta$,

$$\frac{\ell_1 \vee \cdots \vee \ell_k \vee x = y, \qquad m_1 \vee \cdots \vee m_n[z]}{\text{SUBST}(\theta, \ell_1 \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_n[y])} .$$

Unlike demodulation, paramodulation yields a complete inference procedure for first-order logic with equality.

A third approach handles equality reasoning entirely within an extended unification algorithm. That is, terms are unifiable if they are *provably* equal under some substitution, where "provably" allows for some amount of equality reasoning. For example, the terms $1 + 2$ and $2 + 1$ normally are not unifiable, but a unification algorithm that knows that $x + y = y + x$ could unify them with the empty substitution. **Equational unification** of this kind can be done with efficient algorithms designed for the particular axioms used (commutativity, associativity, and so on), rather than through explicit inference with those axioms. Theorem provers using this technique are closely related to the constraint logic programming systems described in Section 9.4.

EQUATIONAL UNIFICATION

Resolution strategies

We know that repeated applications of the resolution inference rule will eventually find a proof if one exists. In this subsection, we examine strategies that help find proofs *efficiently*.

Unit preference

This strategy prefers to do resolutions where one of the sentences is a single literal (also known as a **unit clause**). The idea behind the strategy is that we are trying to produce an empty clause, so it might be a good idea to prefer inferences that produce shorter clauses. Resolving a unit sentence (such as P) with any other sentence (such as $\neg P \vee \neg Q \vee R$) always yields a clause (in this case, $\neg Q \vee R$) that is shorter than the other clause. When the unit preference strategy was first tried for propositional inference in 1964, it led to a dramatic speedup, making it feasible to prove theorems that could not be handled without the preference. Unit preference by itself does not, however, reduce the branching factor in medium-sized problems enough to make them solvable by resolution. It is, nonetheless, a useful heuristic that can be combined with other strategies.

UNIT RESOLUTION

Unit resolution is a restricted form of resolution in which every resolution step must involve a unit clause. Unit resolution is incomplete in general, but complete for Horn knowledge bases. Unit resolution proofs on Horn knowledge bases resemble forward chaining.

Set of support

Preferences that try certain resolutions first are helpful, but in general it is more effective to try to eliminate some potential resolutions altogether. The set-of-support strategy does

SET OF SUPPORT

just that. It starts by identifying a subset of the sentences called the **set of support**. Every resolution combines a sentence from the set of support with another sentence and adds the resolvent into the set of support. If the set of support is small relative to the whole knowledge base, the search space will be reduced dramatically.

We have to be careful with this approach, because a bad choice for the set of support will make the algorithm incomplete. However, if we choose the set of support S so that the remainder of the sentences are jointly satisfiable, then set-of-support resolution will be complete. A common approach is to use the negated query as the set of support, on the assumption that the original knowledge base is consistent. (After all, if it is not consistent, then the fact that the query follows from it is vacuous.) The set-of-support strategy has the additional advantage of generating proof trees that are often easy for humans to understand, because they are goal-directed.

Input resolution

INPUT RESOLUTION

In the **input resolution** strategy, every resolution combines one of the input sentences (from the KB or the query) with some other sentence. The proof in Figure 9.11 uses only input resolutions and has the characteristic shape of a single "spine" with single sentences combining onto the spine. Clearly, the space of proof trees of this shape is smaller than the space of all proof graphs. In Horn knowledge bases, Modus Ponens is a kind of input resolution strategy, because it combines an implication from the original KB with some other sentences. Thus, it is no surprise that input resolution is complete for knowledge bases that are in Horn form, but

LINEAR RESOLUTION

incomplete in the general case. The **linear resolution** strategy is a slight generalization that allows P and Q to be resolved together either if P is in the original KB or if P is an ancestor of Q in the proof tree. Linear resolution is complete.

Subsumption

SUBSUMPTION

The **subsumption** method eliminates all sentences that are subsumed by (i.e., more specific than) an existing sentence in the KB. For example, if $P(x)$ is in the KB, then there is no sense in adding $P(A)$ and even less sense in adding $P(A) \vee Q(B)$. Subsumption helps keep the KB small, and thus helps keep the search space small.

Theorem provers

Theorem provers (also known as automated reasoners) differ from logic programming languages in two ways. First, most logic programming languages handle only Horn clauses, whereas theorem provers accept full first-order logic. Second, Prolog programs intertwine logic and control. The programmer's choice `A :- B, C` instead of `A :- C, B` affects the execution of the program. In most theorem provers, the syntactic form chosen for sentences does not affect the results. Theorem provers still need control information to operate efficiently, but that information is usually kept distinct from the knowledge base, rather than being part of the knowledge representation itself. Most of the research in theorem provers involves finding control strategies that are generally useful, as well as increasing the speed.

Design of a theorem prover

In this section, we describe the theorem prover OTTER (Organized Techniques for Theorem-proving and Effective Research) (McCune, 1992), with particular attention to its control strategy. In preparing a problem for OTTER, the user must divide the knowledge into four parts:

- A set of clauses known as the **set of support** (or *sos*), which defines the important facts about the problem. Every resolution step resolves a member of the set of support against another axiom, so the search is focused on the set of support.
- A set of **usable axioms** that are outside the set of support. These provide background knowledge about the problem area. The boundary between what is part of the problem (and thus in *sos*) and what is background (and thus in the usable axioms) is up to the user's judgment.
- A set of equations known as **rewrites** or **demodulators**. Although demodulators are equations, they are always applied in the left to right direction. Thus, they define a canonical form into which all terms will be simplified. For example, the demodulator $x + 0 = x$ says that every term of the form $x + 0$ should be replaced by the term x.
- A set of parameters and clauses that defines the control strategy. In particular, the user specifies a heuristic function to control the search and a filtering function to eliminate some subgoals as uninteresting.

OTTER works by continually resolving an element of the set of support against one of the usable axioms. Unlike Prolog, it uses a form of best-first search. Its heuristic function measures the "weight" of each clause, where lighter clauses are preferred. The exact choice of heuristic is up to the user, but generally, the weight of a clause should be correlated with its size or difficulty. Unit clauses are treated as light; the search can thus be seen as a generalization of the unit preference strategy. At each step, OTTER moves the "lightest" clause in the

of support to the usable list and adds to the set of support some immediate consequences of resolving the lightest clause with elements of the usable list. OTTER halts when it has found a refutation or when there are no more clauses in the set of support. The algorithm is shown in more detail in Figure 9.14.

```
procedure OTTER(sos, usable)
  inputs: sos, a set of support—clauses defining the problem (a global variable)
          usable, background knowledge potentially relevant to the problem

  repeat
     clause ← the lightest member of sos
     move clause from sos to usable
     PROCESS(INFER(clause, usable), sos)
  until sos = [] or a refutation has been found

function INFER(clause, usable) returns clauses

  resolve clause with each member of usable
  return the resulting clauses after applying FILTER

procedure PROCESS(clauses, sos)

  for each clause in clauses do
     clause ← SIMPLIFY(clause)
     merge identical literals
     discard clause if it is a tautology
     sos ← [clause | sos]
     if clause has no literals then a refutation has been found
     if clause has one literal then look for unit refutation
```

Figure 9.14 Sketch of the OTTER theorem prover. Heuristic control is applied in the selection of the "lightest" clause and in the FILTER function that eliminates uninteresting clauses from consideration.

Extending Prolog

An alternative way to build a theorem prover is to start with a Prolog compiler and extend it to get a sound and complete reasoner for full first-order logic. This was the approach taken in the Prolog Technology Theorem Prover, or PTTP (Stickel, 1988). PTTP includes five significant changes to Prolog to restore completeness and expressiveness:

- The occurs check is put back into the unification routine to make it sound.
- The depth-first search is replaced by an iterative deepening search. This makes the search strategy complete and takes only a constant factor more time.
- Negated literals (such as $\neg P(x)$) are allowed. In the implementation, there are two separate routines, one trying to prove P and one trying to prove $\neg P$.

- A clause with n atoms is stored as n different rules. For example, $A \Leftarrow B \wedge C$ would also be stored as $\neg B \Leftarrow C \wedge \neg A$ and as $\neg C \Leftarrow B \wedge \neg A$. This technique, known as **locking**, means that the current goal need be unified with only the head of each clause, yet it still allows for proper handling of negation.

LOCKING

- Inference is made complete (even for non-Horn clauses) by the addition of the linear input resolution rule: If the current goal unifies with the negation of one of the goals on the stack, then that goal can be considered solved. This is a way of reasoning by contradiction. Suppose the original goal is P and this is reduced by a series of inferences to the goal $\neg P$. This establishes that $\neg P \Rightarrow P$, which is logically equivalent to P.

Despite these changes, PTTP retains the features that make Prolog fast. Unifications are still done by modifying variables directly, with unbinding done by unwinding the trail during backtracking. The search strategy is still based on input resolution, meaning that every resolution is against one of the clauses given in the original statement of the problem (rather than a derived clause). This makes it feasible to compile all the clauses in the original statement of the problem.

The main drawback of PTTP is that the user has to relinquish all control over the search for solutions. Each inference rule is used by the system both in its original form and in the contrapositive form. This can lead to unintuitive searches. For example, consider the rule

$$(f(x,y) = f(a,b)) \Leftarrow (x = a) \wedge (y = b) \ .$$

As a Prolog rule, this is a reasonable way to prove that two f terms are equal. But PTTP would also generate the contrapositive:

$$(x \neq a) \Leftarrow (f(x,y) \neq f(a,b)) \wedge (y = b) \ .$$

It seems that this is a wasteful way to prove that any two terms x and a are different.

Theorem provers as assistants

So far, we have thought of a reasoning system as an independent agent that has to make decisions and act on its own. Another use of theorem provers is as an assistant, providing advice to, say, a mathematician. In this mode the mathematician acts as a supervisor, mapping out the strategy for determining what to do next and asking the theorem prover to fill in the details. This alleviates the problem of semi-decidability to some extent, because the supervisor can cancel a query and try another approach if the query is taking too much time.

PROOF-CHECKER

A theorem prover can also act as a **proof-checker**, where the proof is given by a human as a series of fairly large steps; the individual inferences required to show that each step is sound are filled in by the system.

SOCRATIC REASONER

A **Socratic reasoner** is a theorem prover whose ASK function is incomplete, but which can always arrive at a solution if asked the right series of questions. Thus, Socratic reasoners make good assistants, provided that there is a supervisor to make the right series of calls to ASK. ONTIC (McAllester, 1989) is a Socratic reasoning system for mathematics.

Practical uses of theorem provers

Theorem provers have come up with novel mathematical results. The SAM (Semi-Automated Mathematics) program was the first, proving a lemma in lattice theory (Guard *et al.*, 1969). The AURA program has also answered open questions in several areas of mathematics (Wos and Winker, 1983). The Boyer–Moore theorem prover (Boyer and Moore, 1979) has been used and extended over many years and was used by Natarajan Shankar to give the first fully rigorous formal proof of Gödel's Incompleteness Theorem (Shankar, 1986). The OTTER program is one of the strongest theorem provers; it has been used to solve several open questions in combinatorial logic. The most famous of these concerns **Robbins algebra**. In 1933, Herbert Robbins proposed a simple set of axioms that appeared to define Boolean algebra, but no proof of this could be found (despite serious work by several mathematicians including Alfred Tarski himself). On October 10, 1996, after eight days of computation, EQP (a version of OTTER) found a proof (McCune, 1997).

ROBBINS ALGEBRA

VERIFICATION

SYNTHESIS

Theorem provers can be applied to the problems involved in the **verification** and **synthesis** of both hardware and software, because both domains can be given correct axiomatizations. Thus, theorem proving research is carried out in the fields of hardware design, programming languages, and software engineering—not just in AI. In the case of software, the axioms state the properties of each syntactic element of the programming language. (Reasoning about programs is quite similar to reasoning about actions in the situation calculus.) An algorithm is verified by showing that its outputs meet the specifications for all inputs. The RSA public key encryption algorithm and the Boyer–Moore string-matching algorithm have been verified this way (Boyer and Moore, 1984). In the case of hardware, the axioms describe the interactions between signals and circuit elements. (See Chapter 8 for an example.) The design of a 16-bit adder has been verified by AURA (Wojcik, 1983). Logical reasoners designed specially for verification have been able to verify entire CPUs, including their timing properties (Srivas and Bickford, 1990).

The formal synthesis of algorithms was one of the first uses of theorem provers, as outlined by Cordell Green (1969a), who built on earlier ideas by Simon (1963). The idea is to prove a theorem to the effect that "there exists a program p satisfying a certain specification." If the proof is constrained to be constructive, the program can be extracted. Although fully automated **deductive synthesis**, as it is called, has not yet become feasible for general-purpose programming, hand-guided deductive synthesis has been successful in designing several novel and sophisticated algorithms. Synthesis of special-purpose programs is also an active area of research. In the area of hardware synthesis, the AURA theorem prover has been applied to design circuits that are more compact than any previous design (Wojciechowski and Wojcik, 1983). For many circuit designs, propositional logic is sufficient because the set of interesting propositions is fixed by the set of circuit elements. The application of propositional inference in hardware synthesis is now a standard technique having many large-scale deployments (see, e.g., Nowick *et al.* (1993)).

DEDUCTIVE SYNTHESIS

These same techniques are now starting to be applied to software verification as well, by systems such as the SPIN model checker (Holzmann, 1997). For example, the Remote Agent spacecraft control program was verified before and after flight (Havelund *et al.*, 2000).

9.6 SUMMARY

We have presented an analysis of logical inference in first-order logic and a number of algorithms for doing it.

- A first approach uses inference rules for instantiating quantifiers in order to propositionalize the inference problem. Typically, this approach is very slow.
- The use of **unification** to identify appropriate substitutions for variables eliminates the instantiation step in first-order proofs, making the process much more efficient.
- A lifted version of **Modus Ponens** uses unification to provide a natural and powerful inference rule, **generalized Modus Ponens**. The **forward chaining** and **backward chaining** algorithms apply this rule to sets of definite clauses.
- Generalized Modus Ponens is complete for definite clauses, although the entailment problem is **semidecidable**. For **Datalog** programs consisting of function-free definite clauses, entailment is decidable.
- Forward chaining is used in **deductive databases**, where it can be combined with relational database operations. It is also used in **production systems**, which perform efficient updates with very large rule sets.
- Forward chaining is complete for Datalog programs and runs in polynomial time.
- Backward chaining is used in **logic programming systems** such as **Prolog**, which employ sophisticated compiler technology to provide very fast inference.
- Backward chaining suffers from redundant inferences and infinite loops; these can be alleviated by **memoization**.
- The generalized **resolution** inference rule provides a complete proof system for first-order logic, using knowledge bases in conjunctive normal form.
- Several strategies exist for reducing the search space of a resolution system without compromising completeness. Efficient resolution-based theorem provers have been used to prove interesting mathematical theorems and to verify and synthesize software and hardware.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

SYLLOGISM

Logical inference was studied extensively in Greek mathematics. The type of inference most carefully studied by Aristotle was the **syllogism**, which is a kind of inference rule. Aristotle's syllogisms did include elements of first-order logic, such as quantification, but were restricted to unary predicates. Syllogisms were categorized by "figures" and "moods," depending on the order of the terms (which we would call predicates) in the sentences, the degree of generality (which we would today interpret through quantifiers) applied to each term, and whether each term is negated. The most fundamental syllogism is that of the first mood of the first figure:

All S are M.
All M are P.
Therefore, all S are P.

Aristotle tried to prove the validity of other syllogisms by "reducing" them to those of the first figure. He was much less precise in describing what this "reduction" should involve than he was in characterizing the syllogistic figures and moods themselves.

Gottlob Frege, who developed full first-order logic in 1879, based his system of inference on a large collection of logically valid schemas plus a single inference rule, Modus Ponens. Frege took advantage of the fact that the effect of an inference rule of the form "From P, infer Q" can be simulated by applying Modus Ponens to P along with a logically valid schema $P \Rightarrow Q$. This "axiomatic" style of exposition, using Modus Ponens plus a number of logically valid schemas, was employed by a number of logicians after Frege; most notably, it was used in *Principia Mathematica* (Whitehead and Russell, 1910).

Inference rules, as distinct from axiom schemas, were the focus of the **natural deduction** approach, introduced by Gerhard Gentzen (1934) and by Stanisław Jaśkowski (1934). Natural deduction is called "natural" because it does not require conversion to (unreadable) normal form and because its inference rules are intended to appear natural to humans. Prawitz (1965) offers a book-length treatment of natural deduction. Gallier (1986) uses Gentzen's approach to expound the theoretical underpinnings of automated deduction.

The invention of clausal form was a crucial step in the development of a deep mathematical analysis of first-order logic. Whitehead and Russell (1910) expounded the so-called *rules of passage* (the actual term is from Herbrand (1930)) that are used to move quantifiers to the front of formulas. Skolem constants and Skolem functions were introduced, appropriately enough, by Thoralf Skolem (1920). The general procedure for skolemization is given by Skolem (1928), along with the important notion of the Herbrand universe.

Herbrand's theorem, named after the French logician Jacques Herbrand (1930), has played a vital role in the development of automated reasoning methods, both before and after Robinson's introduction of resolution. This is reflected in our reference to the "Herbrand universe" rather than the "Skolem universe," even though Skolem really invented the concept. Herbrand can also be regarded as the inventor of unification. Gödel (1930) built on the ideas of Skolem and Herbrand to show that first-order logic has a complete proof procedure. Alan Turing (1936) and Alonzo Church (1936) simultaneously showed, using very different proofs, that validity in first-order logic was not decidable. The excellent text by Enderton (1972) explains all of these results in a rigorous yet moderately understandable fashion.

Although McCarthy (1958) had suggested the use of first-order logic for representation and reasoning in AI, the first such systems were developed by logicians interested in mathematical theorem proving. It was Abraham Robinson who proposed the use of propositionalization and Herbrand's theorem, and Gilmore (1960) who wrote the first program based on this approach. Davis and Putnam (1960) used clausal form and produced a program that attempted to find refutations by substituting members of the Herbrand universe for variables to produce ground clauses and then looking for propositional inconsistencies among the ground clauses. Prawitz (1960) developed the key idea of letting the quest for propositional

inconsistency drive the search process, and generating terms from the Herbrand universe only when it was necessary to do so in order to establish propositional inconsistency. After further development by other researchers, this idea led J. A. Robinson (no relation) to develop the resolution method (Robinson, 1965). The so-called inverse method developed at about the same time by the Soviet researcher S. Maslov (1964, 1967), based on somewhat different principles, offers similar computational advantages over propositionalization. Wolfgang Bibel's (1981) **connection method** can be viewed as an extension of this approach.

After the development of resolution, work on first-order inference proceeded in several different directions. In AI, resolution was adopted for question-answering systems by Cordell Green and Bertram Raphael (1968). A somewhat less formal approach was taken by Carl Hewitt (1969). His PLANNER language, although never fully implemented, was a precursor to logic programming and included directives for forward and backward chaining and for negation as failure. A subset known as MICRO-PLANNER (Sussman and Winograd, 1970) was implemented and used in the SHRDLU natural language understanding system (Winograd, 1972). Early AI implementations put a good deal of effort into data structures that would allow efficient retrieval of facts; this work is covered in AI programming texts (Charniak *et al.*, 1987; Norvig, 1992; Forbus and de Kleer, 1993).

By the early 1970s, **forward chaining** was well established in AI as an easily understandable alternative to resolution. It was used in a wide variety of systems, ranging from Nevins's geometry theorem prover (Nevins, 1975) to the R1 expert system for VAX configuration (McDermott, 1982). AI applications typically involved large numbers of rules, so it was important to develop efficient rule-matching technology, particularly for incremental updates. The technology for **production systems** was developed to support such applications. The production system language OPS-5 (Forgy, 1981; Brownston *et al.*, 1985) was used for R1 and for the SOAR cognitive architecture (Laird *et al.*, 1987). OPS-5 incorporated the rete match process (Forgy, 1982). SOAR, which generates new rules to cache the results of previous computations, can create very large rule sets—over 1,000,000 rules in the case of the TACAIR-SOAR system for controlling simulated fighter aircraft (Jones *et al.*, 1998). CLIPS (Wygant, 1989) was a C-based production system language developed at NASA that allowed better integration with other software, hardware, and sensor systems and was used for spacecraft automation and several military applications.

The area of research known as **deductive databases** has also contributed a great deal to our understanding of forward inference. It began with a workshop in Toulouse in 1977, organized by Jack Minker, that brought together experts in logical inference and database systems (Gallaire and Minker, 1978). A recent historical survey (Ramakrishnan and Ullman, 1995) says, "Deductive [database] systems are an attempt to adapt Prolog, which has a 'small data' view of the world, to a 'large data' world." Thus, it aims to meld relational database technology, which is designed for retrieving large *sets* of facts, with Prolog-based inference technology, which typically retrieves one fact at a time. Texts on deductive databases include Ullman (1989) and Ceri *et al.* (1990).

Influential work by Chandra and Harel (1980) and Ullman (1985) led to the adoption of Datalog as a standard language for deductive databases. "Bottom-up" inference, or forward chaining, also became the standard—partly because it avoids the problems with nontermi-

nation and redundant computation that occur with backward chaining and partly because it has a more natural implementation in terms of the basic relational database operations. The development of the **magic sets** technique for rule rewriting by Bancilhon *et al.* (1986) allowed forward chaining to borrow the advantage of goal-directedness from backward chaining. Equalizing the arms race, tabled logic programming methods (see page 313) borrow the advantage of dynamic programming from forward chaining.

Much of our understanding of the complexity of logical inference has come from the deductive database community. Chandra and Merlin (1977) first showed that matching a single nonrecursive rule (a **conjunctive query** in database terminology) can be NP-hard. Kuper and Vardi (1993) proposed **data complexity**—that is, complexity as a function of database size, viewing rule size as constant—as a suitable measure for query answering. Gottlob *et al.* (1999b) discuss the connection between conjunctive queries and constraint satisfaction, showing how hypertree decomposition can optimize the matching process.

CONJUNCTIVE QUERY

DATA COMPLEXITY

As mentioned earlier, **backward chaining** for logical inference appeared in Hewitt's PLANNER language (1969). Logic programming *per se* evolved independently of this effort. A restricted form of linear resolution called **SL-resolution** was developed by Kowalski and Kuehner (1971), building on Loveland's **model elimination** technique (1968); when applied to definite clauses, it becomes **SLD-resolution**, which lends itself to the interpretation of definite clauses as programs (Kowalski, 1974, 1979a, 1979b). Meanwhile, in 1972, the French researcher Alain Colmerauer had developed and implemented **Prolog** for the purpose of parsing natural language—Prolog's clauses were intended initially as context-free grammar rules (Roussel, 1975; Colmerauer *et al.*, 1973). Much of the theoretical background for logic programming was developed by Kowalski, working with Colmerauer. The semantic definition using least fixed points is due to Van Emden and Kowalski (1976). Kowalski (1988) and Cohen (1988) provide good historical overviews of the origins of Prolog. *Foundations of Logic Programming* (Lloyd, 1987) is a theoretical analysis of the underpinnings of Prolog and other logic programming languages.

SL-RESOLUTION

SLD-RESOLUTION

Efficient Prolog compilers are generally based on the Warren Abstract Machine (WAM) model of computation developed by David H. D. Warren (1983). Van Roy (1990) showed that the application of additional compiler techniques, such as type inference, made Prolog programs competitive with C programs in terms of speed. The Japanese Fifth Generation project, a 10-year research effort beginning in 1982, was based completely on Prolog as the means to develop intelligent systems.

Methods for avoiding unnecessary looping in recursive logic programs were developed independently by Smith *et al.* (1986) and Tamaki and Sato (1986). The latter paper also included memoization for logic programs, a method developed extensively as **tabled logic programming** by David S. Warren. Swift and Warren (1994) show how to extend the WAM to handle tabling, enabling Datalog programs to execute an order of magnitude faster than forward-chaining deductive database systems.

Early theoretical work on constraint logic programming was done by Jaffar and Lassez (1987). Jaffar *et al.* (1992a) developed the CLP(R) system for handling real-valued constraints. Jaffar *et al.* (1992b) generalized the WAM to produce the CLAM (Constraint Logic Abstract Machine) for specifying implementations of CLP. Ait-Kaci and Podelski (1993)

describe a sophisticated language called LIFE, which combines CLP with functional programming and with inheritance reasoning. Kohn (1991) describes an ambitious project to use constraint logic programming as the foundation for a real-time control architecture, with applications to fully automatic pilots.

There are a number of textbooks on logic programming and Prolog. *Logic for Problem Solving* (Kowalski, 1979b) is an early text on logic programming in general. Prolog texts include Clocksin and Mellish (1994), Shoham (1994), and Bratko (2001). Marriott and Stuckey (1998) provide excellent coverage of CLP. Until its demise in 2000, the *Journal of Logic Programming* was the journal of record; it has now been replaced by *Theory and Practice of Logic Programming*. Logic programming conferences include the International Conference on Logic Programming (ICLP) and the International Logic Programming Symposium (ILPS).

Research into **mathematical theorem proving** began even before the first complete first-order systems were developed. Herbert Gelernter's Geometry Theorem Prover (Gelernter, 1959) used heuristic search methods combined with diagrams for pruning false subgoals and was able to prove some quite intricate results in Euclidean geometry. Since that time, however, there has not been very much interaction between theorem proving and AI.

Early work concentrated on completeness. Following Robinson's seminal paper, the demodulation and paramodulation rules for equality reasoning were introduced by Wos *et al.* (1967) and Wos and Robinson (1968), respectively. These rules were also developed independently in the context of term rewriting systems (Knuth and Bendix, 1970). The incorporation of equality reasoning into the unification algorithm is due to Gordon Plotkin (1972); it was also a feature of QLISP (Sacerdoti *et al.*, 1976). Jouannaud and Kirchner (1991) survey equational unification from a term rewriting perspective. Efficient algorithms for standard unification were developed by Martelli and Montanari (1976) and Paterson and Wegman (1978).

In addition to equality reasoning, theorem provers have incorporated a variety of special-purpose decision procedures. Nelson and Oppen (1979) proposed an influential scheme for integrating such procedures into a general reasoning system; other methods include Stickel's (1985) "theory resolution" and Manna and Waldinger's (1986) "special relations."

A number of control strategies have been proposed for resolution, beginning with the unit preference strategy (Wos *et al.*, 1964). The set of support strategy was proposed by Wos *et al.* (1965), to provide a degree of goal-directedness in resolution. Linear resolution first appeared in Loveland (1970). Genesereth and Nilsson (1987, Chapter 5) provide a short but thorough analysis of a wide variety of control strategies.

Guard *et al.* (1969) describe the early SAM theorem prover, which helped to solve an open problem in lattice theory. Wos and Winker (1983) give an overview of the contributions of the AURA theorem prover toward solving open problems in various areas of mathematics and logic. McCune (1992) follows up on this, recounting the accomplishments of AURA's successor, OTTER, in solving open problems. Weidenbach (2001) describes SPASS, one of the strongest current theorem provers. *A Computational Logic* (Boyer and Moore, 1979) is the basic reference on the Boyer-Moore theorem prover. Stickel (1988) covers the Prolog Technology Theorem Prover (PTTP), which combines the advantages of Prolog compilation with the completeness of model elimination (Loveland, 1968). SETHEO (Letz *et al.*, 1992) is another widely used theorem prover based on this approach; it can perform several million

inferences per second on 2000-model workstations. LEANTAP (Beckert and Posegga, 1995) is an efficient theorem prover implemented in only 25 lines of Prolog.

Early work in automated program synthesis was done by Simon (1963), Green (1969a), and Manna and Waldinger (1971). The transformational system of Burstall and Darlington (1977) used equational reasoning for recursive program synthesis. KIDS (Smith, 1990, 1996) is one of the strongest modern systems; it operates as an expert assistant. Manna and Waldinger (1992) give a tutorial introduction to the current state of the art, with emphasis on their own deductive approach. *Automating Software Design* (Lowry and McCartney, 1991) collects a number of papers in the area. The use of logic in hardware design is surveyed by Kern and Greenstreet (1999); Clarke *et al.* (1999) cover model checking for hardware verification.

Computability and Logic (Boolos and Jeffrey, 1989) is a good reference on completeness and undecidability. Many early papers in mathematical logic are to be found in *From Frege to Gödel: A Source Book in Mathematical Logic* (van Heijenoort, 1967). The journal of record for the field of pure mathematical logic (as opposed to automated deduction) is *The Journal of Symbolic Logic*. Textbooks geared toward automated deduction include the classic *Symbolic Logic and Mechanical Theorem Proving* (Chang and Lee, 1973), as well as more recent works by Wos *et al.* (1992), Bibel (1993), and Kaufmann *et al.* (2000). The anthology *Automation of Reasoning* (Siekmann and Wrightson, 1983) includes many important early papers on automated deduction. Other historical surveys have been written by Loveland (1984) and Bundy (1999). The principal journal for the field of theorem proving is the *Journal of Automated Reasoning*; the main conference is the annual Conference on Automated Deduction (CADE). Research in theorem proving is also strongly related to the use of logic in analyzing programs and programming languages, for which the principal conference is Logic in Computer Science.

EXERCISES

9.1 Prove from first principles that Universal Instantiation is sound and that Existential Instantiation produces an inferentially equivalent knowledge base.

EXISTENTIAL INTRODUCTION

9.2 From $Likes(Jerry, IceCream)$ it seems reasonable to infer $\exists x \;\; Likes(x, IceCream)$. Write down a general inference rule, **Existential Introduction**, that sanctions this inference. State carefully the conditions that must be satisfied by the variables and terms involved.

9.3 Suppose a knowledge base contains just one sentence, $\exists x \; AsHighAs(x, Everest)$. Which of the following are legitimate results of applying Existential Instantiation?

a. $AsHighAs(Everest, Everest)$.

b. $AsHighAs(Kilimanjaro, Everest)$.

c. $AsHighAs(Kilimanjaro, Everest) \wedge AsHighAs(BenNevis, Everest)$ (after two applications).

9.4 For each pair of atomic sentences, give the most general unifier if it exists:

a. $P(A, B, B)$, $P(x, y, z)$.

b. $Q(y, G(A, B))$, $Q(G(x, x), y)$.

c. $Older(Father(y), y)$, $Older(Father(x), John)$.

d. $Knows(Father(y), y)$, $Knows(x, x)$.

9.5 Consider the subsumption lattices shown in Figure 9.2.

a. Construct the lattice for the sentence $Employs(Mother(John), Father(Richard))$.

b. Construct the lattice for the sentence $Employs(IBM, y)$ ("Everyone works for IBM"). Remember to include every kind of query that unifies with the sentence.

c. Assume that STORE indexes each sentence under every node in its subsumption lattice. Explain how FETCH should work when some of these sentences contain variables; use as examples the sentences in (a) and (b) and the query $Employs(x, Father(x))$.

9.6 Suppose we put into a logical database a segment of the U.S. census data listing the age, city of residence, date of birth, and mother of every person, using social security numbers as identifying constants for each person. Thus, George's age is given by Age(443-65-1282, 56). Which of the indexing schemes S1–S5 following enable an efficient solution for which of the queries Q1–Q4 (assuming normal backward chaining)?

- ◇ **S1**: an index for each atom in each position.
- ◇ **S2**: an index for each first argument.
- ◇ **S3**: an index for each predicate atom.
- ◇ **S4**: an index for each *combination* of predicate and first argument.
- ◇ **S5**: an index for each *combination* of predicate and second argument and an index for each first argument (nonstandard).
- ◇ **Q1**: Age(443-44-4321, x)
- ◇ **Q2**: $ResidesIn(x, Houston)$
- ◇ **Q3**: $Mother(x, y)$
- ◇ **Q4**: $Age(x, 34) \wedge ResidesIn(x, TinyTownUSA)$

9.7 One might suppose that we can avoid the problem of variable conflict in unification during backward chaining by standardizing apart all of the sentences in the knowledge base once and for all. Show that, for some sentences, this approach cannot work. (*Hint*: Consider a sentence, one part of which unifies with another.)

9.8 Explain how to write any given 3-SAT problem of arbitrary size using a single first-order definite clause and no more than 30 ground facts.

9.9 Write down logical representations for the following sentences, suitable for use with Generalized Modus Ponens:

a. Horses, cows, and pigs are mammals.
b. An offspring of a horse is a horse.
c. Bluebeard is a horse.
d. Bluebeard is Charlie's parent.
e. Offspring and parent are inverse relations.
f. Every mammal has a parent.

9.10 In this question we will use the sentences you wrote in Exercise 9.9 to answer a question using a backward-chaining algorithm.

a. Draw the proof tree generated by an exhaustive backward-chaining algorithm for the query $\exists h \ \ Horse(h)$, where clauses are matched in the order given.
b. What do you notice about this domain?
c. How many solutions for h actually follow from your sentences?
d. Can you think of a way to find all of them? (*Hint*: You might want to consult Smith *et al.* (1986).)

9.11 A popular children's riddle is "Brothers and sisters have I none, but that man's father is my father's son." Use the rules of the family domain (Chapter 8) to show who that man is. You may apply any of the inference methods described in this chapter. Why do you think that this riddle is difficult?

9.12 Trace the execution of the backward chaining algorithm in Figure 9.6 when it is applied to solve the crime problem. Show the sequence of values taken on by the *goals* variable, and arrange them into a tree.

9.13 The following Prolog code defines a predicate `P`:

```
P(X, [X|Y]).
P(X, [Y|Z]) :- P(X,Z).
```

a. Show proof trees and solutions for the queries `P(A, [1,2,3])` and `P(2, [1,A,3])`.
b. What standard list operation does `P` represent?

9.14 In this exercise, we will look at sorting in Prolog.

a. Write Prolog clauses that define the predicate `sorted(L)`, which is true if and only if list `L` is sorted in ascending order.
b. Write a Prolog definition for the predicate `perm(L,M)`, which is true if and only if `L` is a permutation of `M`.
c. Define `sort(L,M)` (`M` is a sorted version of `L`) using `perm` and `sorted`.
d. Run `sort` on longer and longer lists until you lose patience. What is the time complexity of your program?
e. Write a faster sorting algorithm, such as insertion sort or quicksort, in Prolog.

9.15 In this exercise, we will look at the recursive application of rewrite rules, using logic programming. A rewrite rule (or **demodulator** in OTTER terminology) is an equation with a specified direction. For example, the rewrite rule $x+0 \rightarrow x$ suggests replacing any expression that matches $x + 0$ with the expression x. The application of rewrite rules is a central part of equational reasoning systems. We will use the predicate `rewrite(X,Y)` to represent rewrite rules. For example, the earlier rewrite rule is written as `rewrite(X+0,X)`. Some terms are *primitive* and cannot be further simplified; thus, we will write `primitive(0)` to say that 0 is a primitive term.

a. Write a definition of a predicate `simplify(X,Y)`, that is true when `Y` is a simplified version of `X`—that is, when no further rewrite rules are applicable to any subexpression of `Y`.

b. Write a collection of rules for the simplification of expressions involving arithmetic operators, and apply your simplification algorithm to some sample expressions.

c. Write a collection of rewrite rules for symbolic differentiation, and use them along with your simplification rules to differentiate and simplify expressions involving arithmetic expressions, including exponentiation.

9.16 In this exercise, we will consider the implementation of search algorithms in Prolog. Suppose that `successor(X,Y)` is true when state `Y` is a successor of state `X`; and that `goal(X)` is true when `X` is a goal state. Write a definition for `solve(X,P)`, which means that `P` is a path (list of states) beginning with `X`, ending in a goal state, and consisting of a sequence of legal steps as defined by `successor`. You will find that depth-first search is the easiest way to do this. How easy would it be to add heuristic search control?

9.17 How can resolution be used to show that a sentence is valid? Unsatisfiable?

9.18 From "Horses are animals," it follows that "The head of a horse is the head of an animal." Demonstrate that this inference is valid by carrying out the following steps:

a. Translate the premise and the conclusion into the language of first-order logic. Use three predicates: $HeadOf(h, x)$ (meaning "h is the head of x"), $Horse(x)$, and $Animal(x)$.

b. Negate the conclusion, and convert the premise and the negated conclusion into conjunctive normal form.

c. Use resolution to show that the conclusion follows from the premise.

9.19 Here are two sentences in the language of first-order logic:

(A): $\forall x \ \exists y \ (x \geq y)$

(B): $\exists y \ \forall x \ (x \geq y)$

a. Assume that the variables range over all the natural numbers $0, 1, 2, \ldots, \infty$ and that the "$\geq$" predicate means "is greater than or equal to." Under this interpretation, translate (A) and (B) into English.

b. Is (A) true under this interpretation?

c. Is (B) true under this interpretation?

d. Does (A) logically entail (B)?

e. Does (B) logically entail (A)?

f. Using resolution, try to prove that (A) follows from (B). Do this even if you think that (B) does not logically entail (A); continue until the proof breaks down and you cannot proceed (if it does break down). Show the unifying substitution for each resolution step. If the proof fails, explain exactly where, how, and why it breaks down.

g. Now try to prove that (B) follows from (A).

9.20 Resolution can produce nonconstructive proofs for queries with variables, so we had to introduce special mechanisms to extract definite answers. Explain why this issue does not arise with knowledge bases containing only definite clauses.

9.21 We said in this chapter that resolution cannot be used to generate all logical consequences of a set of sentences. Can any algorithm do this?

10 KNOWLEDGE REPRESENTATION

In which we show how to use first-order logic to represent the most important aspects of the real world, such as action, space, time, mental events, and shopping.

The last three chapters described the technology for knowledge-based agents: the syntax, semantics, and proof theory of propositional and first-order logic, and the implementation of agents that use these logics. In this chapter we address the question of what *content* to put into such an agent's knowledge base—how to represent facts about the world.

Section 10.1 introduces the idea of a general ontology, which organizes everything in the world into a hierarchy of categories. Section 10.2 covers the basic categories of objects, substances, and measures. Section 10.3 discusses representations for actions, which are central to the construction of knowledge-based agents, and also explains the more general notion of **events**, or space–time chunks. Section 10.4 discusses knowledge about beliefs, and Section 10.5 brings all the knowledge together in the context of an Internet shopping environment. Sections 10.6 and 10.7 cover specialized reasoning systems for representing uncertain and changing knowledge.

10.1 ONTOLOGICAL ENGINEERING

In "toy" domains, the choice of representation is not that important; it is easy to come up with a consistent vocabulary. On the other hand, complex domains such as shopping on the Internet or controlling a robot in a changing physical environment require more general and flexible representations. This chapter shows how to create these representations, concentrating on general concepts—such as *Actions, Time, Physical Objects*, and *Beliefs*—that occur in many different domains. Representing these abstract concepts is sometimes called **ontological engineering**—it is related to the **knowledge engineering** process described in Section 8.4, but operates on a grander scale.

ONTOLOGICAL ENGINEERING

The prospect of representing *everything* in the world is daunting. Of course, we won't actually write a complete description of everything—that would be far too much for even a 1000–page textbook—but we will leave placeholders where new knowledge for any domain can fit in. For example, we will define what it means to be a physical object, and the details

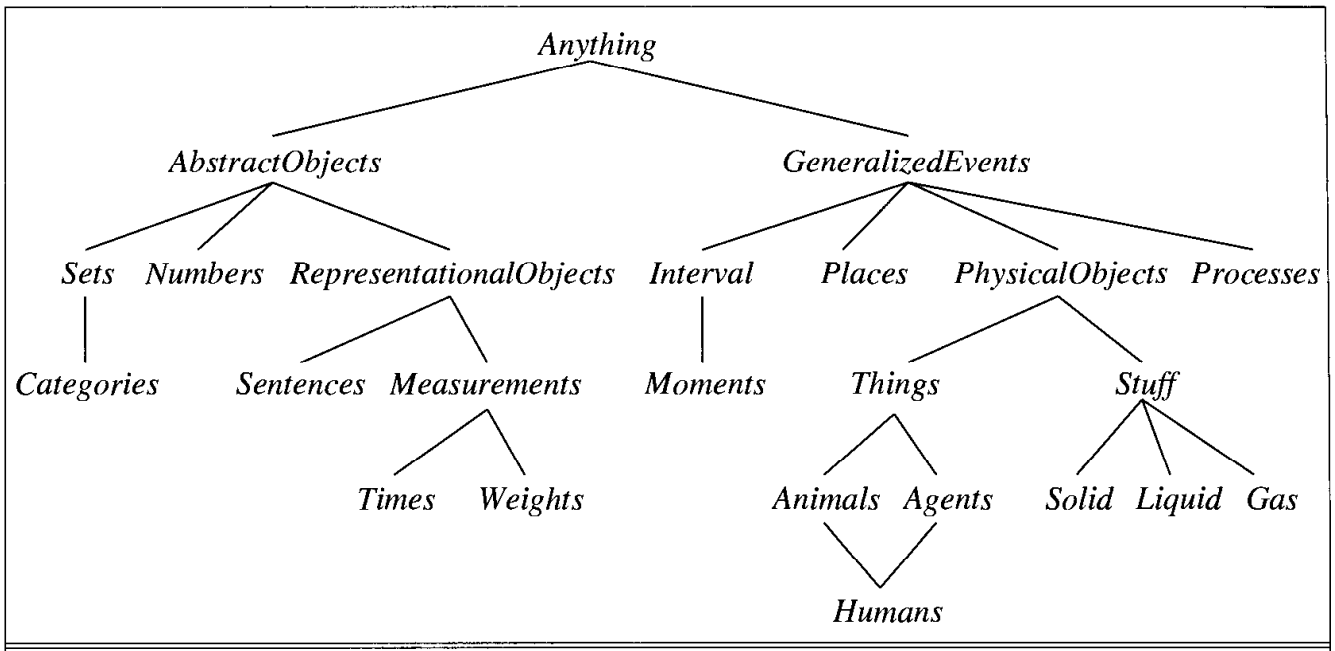


Figure 10.1 The upper ontology of the world, showing the topics to be covered later in the chapter. Each arc indicates that the lower concept is a specialization of the upper one.

of different types of objects—robots, televisions, books, or whatever—can be filled in later.

UPPER ONTOLOGY

The general framework of concepts is called an **upper ontology**, because of the convention of drawing graphs with the general concepts at the top and the more specific concepts below them, as in Figure 10.1.

Before considering the ontology further, we should state one important caveat. We have elected to use first-order logic to discuss the content and organization of knowledge. Certain aspects of the real world are hard to capture in FOL. The principal difficulty is that almost all generalizations have exceptions, or hold only to a degree. For example, although "tomatoes are red" is a useful rule, some tomatoes are green, yellow, or orange. Similar exceptions can be found to almost all the general statements in this chapter. The ability to handle exceptions and uncertainty is extremely important, but is orthogonal to the task of understanding the general ontology. For this reason, we will delay the discussion of exceptions until Section 10.6, and the more general topic of uncertain information until Chapter 13.

Of what use is an upper ontology? Consider again the ontology for circuits in Section 8.4. It makes a large number of simplifying assumptions. For example, time is omitted completely. Signals are fixed, and do not propagate. The structure of the circuit remains constant. If we wanted to make this more general, consider signals at particular times, and include the wire lengths and propagation delays. This would allow us to simulate the timing properties of the circuit, and indeed such simulations are often carried out by circuit designers. We could also introduce more interesting classes of gates, for example by describing the technology (TTL, MOS, CMOS, and so on) as well as the input/output specification. If we wanted to discuss reliability or diagnosis, we would include the possibility that the structure of the circuit or the properties of the gates might change spontaneously. To account for stray capacitances, we would need to move from a purely topological representation of connectivity to a more realistic description of geometric properties.

If we look at the wumpus world, similar considerations apply. Although we do include time, it has a very simple structure: Nothing happens except when the agent acts, and all changes are instantaneous. A more general ontology, better suited for the real world, would allow for simultaneous changes extended over time. We also used a Pit predicate to say which squares have pits. We could have allowed for different kinds of pits by having several individuals belonging to the class of pits, each having different properties. Similarly, we might want to allow for other animals besides wumpuses. It might not be possible to pin down the exact species from the available percepts, so we would need to build up a wumpus-world biological taxonomy to help the agent predict behavior from scanty clues.

For any special-purpose ontology, it is possible to make changes like these to move toward greater generality. An obvious question then arises: do all these ontologies converge on a general-purpose ontology? After centuries of philosophical and computational investigation, the answer is "Possibly." In this section, we will present one version, representing a synthesis of ideas from those centuries. There are two major characteristics of general-purpose ontologies that distinguish them from collections of special-purpose ontologies:

- A general-purpose ontology should be applicable in more or less any special-purpose domain (with the addition of domain-specific axioms). This means that, as far as possible, no representational issue can be finessed or brushed under the carpet.
- In any sufficiently demanding domain, different areas of knowledge must be *unified*, because reasoning and problem solving could involve several areas simultaneously. A robot circuit-repair system, for instance, needs to reason about circuits in terms of electrical connectivity and physical layout, and about time, both for circuit timing analysis and estimating labor costs. The sentences describing time therefore must be capable of being combined with those describing spatial layout and must work equally well for nanoseconds and minutes and for angstroms and meters.

After we present the general ontology we use it to describe the Internet shopping domain. This domain is more than adequate to exercise our ontology, and leaves plenty of scope for the reader to do some creative knowledge representation of his or her own. Consider for example that the Internet shopping agent must know about myriad subjects and authors to buy books at Amazon.com, about all sorts of foods to buy groceries at Peapod.com, and about everything one might find at a garage sale to hunt for bargains at Ebay.com.[1]

10.2 Categories and Objects

CATEGORIES

The organization of objects into **categories** is a vital part of knowledge representation. Although interaction with the world takes place at the level of individual objects, *much reasoning takes place at the level of categories.* For example, a shopper might have the goal of buying a basketball, rather than a *particular* basketball such as BB_9. Categories also serve to make predictions about objects once they are classified. One infers the presence of certain

[1] We apologize if, due to circumstances beyond our control, some of these online stores are no longer functioning by the time you read this.

objects from perceptual input, infers category membership from the perceived properties of the objects, and then uses category information to make predictions about the objects. For example, from its green, mottled skin, large size, and ovoid shape, one can infer that an object is a watermelon; from this, one infers that it would be useful for fruit salad.

There are two choices for representing categories in first-order logic: predicates and objects. That is, we can use the predicate $Basketball(b)$, or we can **reify** the category as an object, $Basketballs$. We could then say $Member(b, Basketballs)$ (which we will abbreviate as $b \in Basketballs$) to say that b is a member of the category of basketballs. We say $Subset(Basketballs, Balls)$ (abbreviated as $Basketballs \subset Balls$) to say that $Basketballs$ is a subcategory, or subset, of $Balls$. So you can think of a category as being the set of its members, or you can think of it as a more complex object that just happens to have the $Member$ and $Subset$ relations defined for it.

INHERITANCE

Categories serve to organize and simplify the knowledge base through **inheritance**. If we say that all instances of the category $Food$ are edible, and if we assert that $Fruit$ is a subclass of $Food$ and $Apples$ is a subclass of $Fruit$, then we know that every apple is edible. We say that the individual apples **inherit** the property of edibility, in this case from their membership in the $Food$ category.

TAXONOMY

Subclass relations organize categories into a **taxonomy**, or **taxonomic hierarchy**. Taxonomies have been used explicitly for centuries in technical fields. For example, systematic biology aims to provide a taxonomy of all living and extinct species; library science has developed a taxonomy of all fields of knowledge, encoded as the Dewey Decimal system; and tax authorities and other government departments have developed extensive taxonomies of occupations and commercial products. Taxonomies are also an important aspect of general commonsense knowledge.

First-order logic makes it easy to state facts about categories, either by relating objects to categories or by quantifying over their members:

- An object is a member of a category. For example:
  $BB_9 \in Basketballs$
- A category is a subclass of another category. For example:
  $Basketballs \subset Balls$
- All members of a category have some properties. For example:
  $x \in Basketballs \Rightarrow Round(x)$
- Members of a category can be recognized by some properties. For example:
  $Orange(x) \wedge Round(x) \wedge Diameter(x) = 9.5'' \wedge x \in Balls \Rightarrow x \in Basketballs$
- A category as a whole has some properties. For example:
  $Dogs \in DomesticatedSpecies$

Notice that because $Dogs$ is a category and is a member of $DomesticatedSpecies$, the latter must be a category of categories. One can even have categories of categories of categories, but they are not much use.

Although subclass and member relations are the most important ones for categories, we also want to be able to state relations between categories that are not subclasses of each other. For example, if we just say that $Males$ and $Females$ are subclasses of $Animals$, then

DISJOINT

EXHAUSTIVE DECOMPOSITION

PARTITION

we have not said that a male cannot be a female. We say that two or more categories are **disjoint** if they have no members in common. And even if we know that males and females are disjoint, we will not know that an animal that is not a male must be a female, unless we say that males and females constitute an **exhaustive decomposition** of the animals. A disjoint exhaustive decomposition is known as a **partition**. The following examples illustrate these three concepts:

$$Disjoint(\{Animals, Vegetables\})$$
$$ExhaustiveDecomposition(\{Americans, Canadians, Mexicans\}, NorthAmericans)$$
$$Partition(\{Males, Females\}, Animals) .$$

(Note that the *ExhaustiveDecomposition* of *NorthAmericans* is not a *Partition*, because some people have dual citizenship.) The three predicates are defined as follows:

$$Disjoint(s) \Leftrightarrow (\forall c_1, c_2 \;\; c_1 \in s \wedge c_2 \in s \wedge c_1 \neq c_2 \Rightarrow Intersection(c_1, c_2) = \{\})$$
$$ExhaustiveDecomposition(s, c) \Leftrightarrow (\forall i \;\; i \in c \Leftrightarrow \exists c_2 \;\; c_2 \in s \wedge i \in c_2)$$
$$Partition(s, c) \Leftrightarrow Disjoint(s) \wedge ExhaustiveDecomposition(s, c) .$$

Categories can also be *defined* by providing necessary and sufficient conditions for membership. For example, a bachelor is an unmarried adult male:

$$x \in Bachelors \Leftrightarrow Unmarried(x) \wedge x \in Adults \wedge x \in Males .$$

As we discuss in the sidebar on natural kinds, strict logical definitions for categories are neither always possible nor always necessary.

Physical composition

The idea that one object can be part of another is a familiar one. One's nose is part of one's head, Romania is part of Europe, and this chapter is part of this book. We use the general *PartOf* relation to say that one thing is part of another. Objects can be grouped into *PartOf* hierarchies, reminiscent of the *Subset* hierarchy:

$$PartOf(Bucharest, Romania)$$
$$PartOf(Romania, EasternEurope)$$
$$PartOf(EasternEurope, Europe)$$
$$PartOf(Europe, Earth) .$$

The *PartOf* relation is transitive and reflexive; that is,

$$PartOf(x, y) \wedge PartOf(y, z) \Rightarrow PartOf(x, z) .$$
$$PartOf(x, x) .$$

Therefore, we can conclude $PartOf(Bucharest, Earth)$.

COMPOSITE OBJECT

Categories of **composite objects** are often characterized by structural relations among parts. For example, a biped has two legs attached to a body:

$$\begin{aligned} Biped(a) \Rightarrow \exists l_1, l_2, b \;\; & Leg(l_1) \wedge Leg(l_2) \wedge Body(b) \wedge \\ & PartOf(l_1, a) \wedge PartOf(l_2, a) \wedge PartOf(b, a) \wedge \\ & Attached(l_1, b) \wedge Attached(l_2, b) \wedge \\ & l_1 \neq l_2 \wedge [\forall l_3 \;\; Leg(l_3) \wedge PartOf(l_3, a) \Rightarrow (l_3 = l_1 \vee l_3 = l_2)] . \end{aligned}$$

The notation for "exactly two" is a little awkward; we are forced to say that there are two legs, that they are not the same, and that if anyone proposes a third leg, it must be the same as one of the other two. In Section 10.6, we will see how a formalism called description logic makes it easier to represent constraints like "exactly two."

We can define a *PartPartition* relation analogous to the *Partition* relation for categories. (See Exercise 10.6.) An object is composed of the parts in its *PartPartition* and can be viewed as deriving some properties from those parts. For example, the mass of a composite object is the sum of the masses of the parts. Notice that this is not the case with categories, which have no mass, even though their elements might.

It is also useful to define composite objects with definite parts but no particular structure. For example, we might want to say, "The apples in this bag weigh two pounds." The temptation would be to ascribe this weight to the *set* of apples in the bag, but this would be a mistake because the set is an abstract mathematical concept that has elements but does not have weight. Instead, we need a new concept, which we will call a **bunch**. For example, if the apples are $Apple_1$, $Apple_2$, and $Apple_3$, then

BUNCH

$$BunchOf(\{Apple_1, Apple_2, Apple_3\})$$

denotes the composite object with the three apples as parts (not elements). We can then use the bunch as a normal, albeit unstructured, object. Notice that $BunchOf(\{x\}) = x$. Furthermore, $BunchOf(Apples)$ is the composite object consisting of all apples—not to be confused with $Apples$, the category or set of all apples.

We can define $BunchOf$ in terms of the $PartOf$ relation. Obviously, each element of s is part of $BunchOf(s)$:

$$\forall x \;\; x \in s \;\Rightarrow\; PartOf(x, BunchOf(s)) \; .$$

Furthermore, $BunchOf(s)$ *is the smallest object satisfying this condition.* In other words, $BunchOf(s)$ must be part of any object that has all the elements of s as parts:

$$\forall y \;\; [\forall x \;\; x \in s \;\Rightarrow\; PartOf(x, y)] \;\Rightarrow\; PartOf(BunchOf(s), y) \; .$$

LOGICAL MINIMIZATION

These axioms are an example of a general technique called **logical minimization**, which means defining an object as the smallest one satisfying certain conditions.

Measurements

In both scientific and commonsense theories of the world, objects have height, mass, cost, and so on. The values that we assign for these properties are called **measures**. Ordinary quantitative measures are quite easy to represent. We imagine that the universe includes abstract "measure objects," such as the *length* that is the length of this line segment: ⊢——————————⊣. We can call this length 1.5 inches or 3.81 centimeters. Thus, the same length has different names in our language. Logically, this can be done by combining a **units function** with a number. (An alternative scheme is explored in Exercise 10.8.) If the line segment is called L_1, we can write

MEASURES

UNITS FUNCTION

$$Length(L_1) = Inches(1.5) = Centimeters(3.81) \; .$$

Conversion between units is done by equating multiples of one unit to another:

$$Centimeters(2.54 \times d) = Inches(d) \; .$$

NATURAL KINDS

Some categories have strict definitions: an object is a triangle if and only if it is a polygon with three sides. On the other hand, most categories in the real world have no clear-cut definition; these are called **natural kind** categories. For example, tomatoes tend to be a dull scarlet; roughly spherical; with an indentation at the top where the stem was; about two to four inches in diameter; with a thin but tough skin; and with flesh, seeds, and juice inside. There is, however, variation: some tomatoes are orange, unripe tomatoes are green, some are smaller or larger than average, and cherry tomatoes are uniformly small. Rather than having a complete definition of tomatoes, we have a set of features that serves to identify objects that are clearly typical tomatoes, but might not be able to decide for other objects. (Could there be a tomato that is furry, like a peach?)

This poses a problem for a logical agent. The agent cannot be sure that an object it has perceived is a tomato, and even if it were sure, it could not be certain which of the properties of typical tomatoes this one has. This problem is an inevitable consequence of operating in partially observable environments.

One useful approach is to separate what is true of all instances of a category from what is true only of typical instances. So in addition to the category $Tomatoes$, we will also have the category $Typical(Tomatoes)$. Here, the $Typical$ function maps a category to the subclass that contains only typical instances:

$$Typical(c) \subseteq c\ .$$

Most knowledge about natural kinds will actually be about their typical instances:

$$x \in Typical(Tomatoes) \Rightarrow Red(x) \wedge Round(x)\ .$$

Thus, we can write down useful facts about categories without exact definitions.

The difficulty of providing exact definitions for most natural categories was explained in depth by Wittgenstein (1953), in his book *Philosophical Investigations*. He used the example of *games* to show that members of a category shared "family resemblances" rather than necessary and sufficient characteristics.

The utility of the notion of strict definition was also challenged by Quine (1953). He pointed out that even the definition of "bachelor" as an unmarried adult male is suspect; one might, for example, question a statement such as "the Pope is a bachelor." While not strictly *false*, this usage is certainly *infelicitous* because it induces unintended inferences on the part of the listener. The tension could perhaps be resolved by distinguishing between logical definitions suitable for internal knowledge representation and the more nuanced criteria for felicitous linguistic usage. The latter may be achieved by "filtering" the assertions derived from the former. It is also possible that failures of linguistic usage serve as feedback for modifying internal definitions, so that filtering becomes unnecessary.

Similar axioms can be written for pounds and kilograms, seconds and days, and dollars and cents. Measures can be used to describe objects as follows:

$$Diameter(Basketball_{12}) = Inches(9.5) .$$
$$ListPrice(Basketball_{12}) = \$(19) .$$
$$d \in Days \Rightarrow Duration(d) = Hours(24) .$$

Note that $\$(1)$ is *not* a dollar bill! One can have two dollar bills, but there is only one object named $\$(1)$. Note also that, while $Inches(0)$ and $Centimeters(0)$ refer to the same zero length, they are not identical to other zero measures, such as $Seconds(0)$.

Simple, quantitative measures are easy to represent. Other measures present more of a problem, because they have no agreed scale of values. Exercises have difficulty, desserts have deliciousness, and poems have beauty, yet numbers cannot be assigned to these qualities. One might, in a moment of pure accountancy, dismiss such properties as useless for the purpose of logical reasoning; or, still worse, attempt to impose a numerical scale on beauty. This would be a grave mistake, because it is unnecessary. The most important aspect of measures is not the particular numerical values, but the fact that measures can be *ordered*.

Although measures are not numbers, we can still compare them using an ordering symbol such as $>$. For example, we might well believe that Norvig's exercises are tougher than Russell's, and that one scores less on tougher exercises:

$$e_1 \in Exercises \wedge e_2 \in Exercises \wedge Wrote(Norvig, e_1) \wedge Wrote(Russell, e_2) \Rightarrow Difficulty(e_1) > Difficulty(e_2) .$$
$$e_1 \in Exercises \wedge e_2 \in Exercises \wedge Difficulty(e_1) > Difficulty(e_2) \Rightarrow ExpectedScore(e_1) < ExpectedScore(e_2) .$$

This is enough to allow one to decide which exercises to do, even though no numerical values for difficulty were ever used. (One does, however, have to discover who wrote which exercises.) These sorts of monotonic relationships among measures form the basis for the field of **qualitative physics**, a subfield of AI that investigates how to reason about physical systems without plunging into detailed equations and numerical simulations. Qualitative physics is discussed in the historical notes section.

Substances and objects

The real world can be seen as consisting of primitive objects (particles) and composite objects built from them. By reasoning at the level of large objects such as apples and cars, we can overcome the complexity involved in dealing with vast numbers of primitive objects individually. There is, however, a significant portion of reality that seems to defy any obvious **individuation**—division into distinct objects. We give this portion the generic name **stuff**. For example, suppose I have some butter and an aardvark in front of me. I can say there is one aardvark, but there is no obvious number of "butter-objects," because any part of a butter-object is also a butter-object, at least until we get to very small parts indeed. This is the major distinction between stuff and things. If we cut an aardvark in half, we do not get two aardvarks (unfortunately).

INDIVIDUATION

STUFF

The English language distinguishes clearly between stuff and things. We say "an aardvark," but, except in pretentious California restaurants, one cannot say "a butter." Linguists

COUNT NOUNS
MASS NOUNS

distinguish between **count nouns**, such as aardvarks, holes, and theorems, and **mass nouns**, such as butter, water, and energy. Several competing ontologies claim to handle this distinction. We will describe just one; the others are covered in the historical notes section.

To represent stuff properly, we begin with the obvious. We will need to have as objects in our ontology at least the gross "lumps" of stuff we interact with. For example, we might recognize a lump of butter as the same butter that was left on the table the night before; we might pick it up, weigh it, sell it, or whatever. In these senses, it is an object just like the aardvark. Let us call it $Butter_3$. We will also define the category $Butter$. Informally, its elements will be all those things of which one might say "It's butter," including $Butter_3$. With some caveats about very small parts that we will omit for now, any part of a butter-object is also a butter-object:

$$x \in Butter \wedge PartOf(y, x) \Rightarrow y \in Butter\ .$$

We can now say that butter melts at around 30 degrees centigrade:

$$x \in Butter \Rightarrow MeltingPoint(x, Centigrade(30))\ .$$

We could go on to say that butter is yellow, is less dense than water, is soft at room temperature, has a high fat content, and so on. On the other hand, butter has no particular size, shape, or weight. We can define more specialized categories of butter such as $UnsaltedButter$, which is also a kind of stuff. On the other hand, the category $PoundOfButter$, which includes as members all butter-objects weighing one pound, is not a substance! If we cut a pound of butter in half, we do not, alas, get two pounds of butter.

INTRINSIC
EXTRINSIC

What is actually going on is this: there are some properties that are **intrinsic**: they belong to the very substance of the object, rather than to the object as a whole. When you cut a substance in half, the two pieces retain the same set of intrinsic properties—things like density, boiling point, flavor, color, ownership, and so on. On the other hand, **extrinsic** properties are the opposite: properties such as weight, length, shape, function, and so on are not retained under subdivision.

A class of objects that includes in its definition only *intrinsic* properties is then a substance, or mass noun; a class that includes *any* extrinsic properties in its definition is a count noun. The category *Stuff* is the most general substance category, specifying no intrinsic properties. The category *Thing* is the most general discrete object category, specifying no extrinsic properties. All physical objects belong to both categories, so the categories are coextensive—they refer to the same entities.

10.3 Actions, Situations, and Events

Reasoning about the results of actions is central to the operation of a knowledge-based agent. Chapter 7 gave examples of propositional sentences describing how actions affect the wumpus world—for example, Equation (7.3) on page 227 states how the agent's location is changed by forward motion. One drawback of propositional logic is the need to have a different copy of the action description for each time at which the action might be executed. This section describes a representation method that uses first-order logic to avoid that problem.

The ontology of situation calculus

One obvious way to avoid multiple copies of axioms is simply to quantify over time—to say, "$\forall t$, such-and-such is the result at $t+1$ of doing the action at t." Instead of dealing with explicit times like $t+1$, we will concentrate in this section on *situations*, which denote the states resulting from executing actions. This approach is called **situation calculus** and involves the following ontology:

SITUATION CALCULUS

- As in Chapter 8, actions are logical terms such as $Forward$ and $Turn(Right)$. For now, we will assume that the environment contains only one agent. (If there is more than one, an additional argument can be inserted to say which agent is doing the action.)

SITUATIONS

- **Situations** are logical terms consisting of the initial situation (usually called S_0) and all situations that are generated by applying an action to a situation. The function $Result(a, s)$ (sometimes called Do) names the situation that results when action a is executed in situation s. Figure 10.2 illustrates this idea.

FLUENTS

- **Fluents** are functions and predicates that vary from one situation to the next, such as the location of the agent or the aliveness of the wumpus. The dictionary says a fluent is something that flows, like a liquid. In this use, it means flowing or changing across situations. By convention, the situation is always the last argument of a fluent. For example, $\neg Holding(G_1, S_0)$ says that the agent is not holding the gold G_1 in the initial situation S_0. $Age(Wumpus, S_0)$ refers to the wumpus's age in S_0.
- **Atemporal** or **eternal** predicates and functions are also allowed. Examples include the predicate $Gold(G_1)$ and the function $LeftLegOf(Wumpus)$.

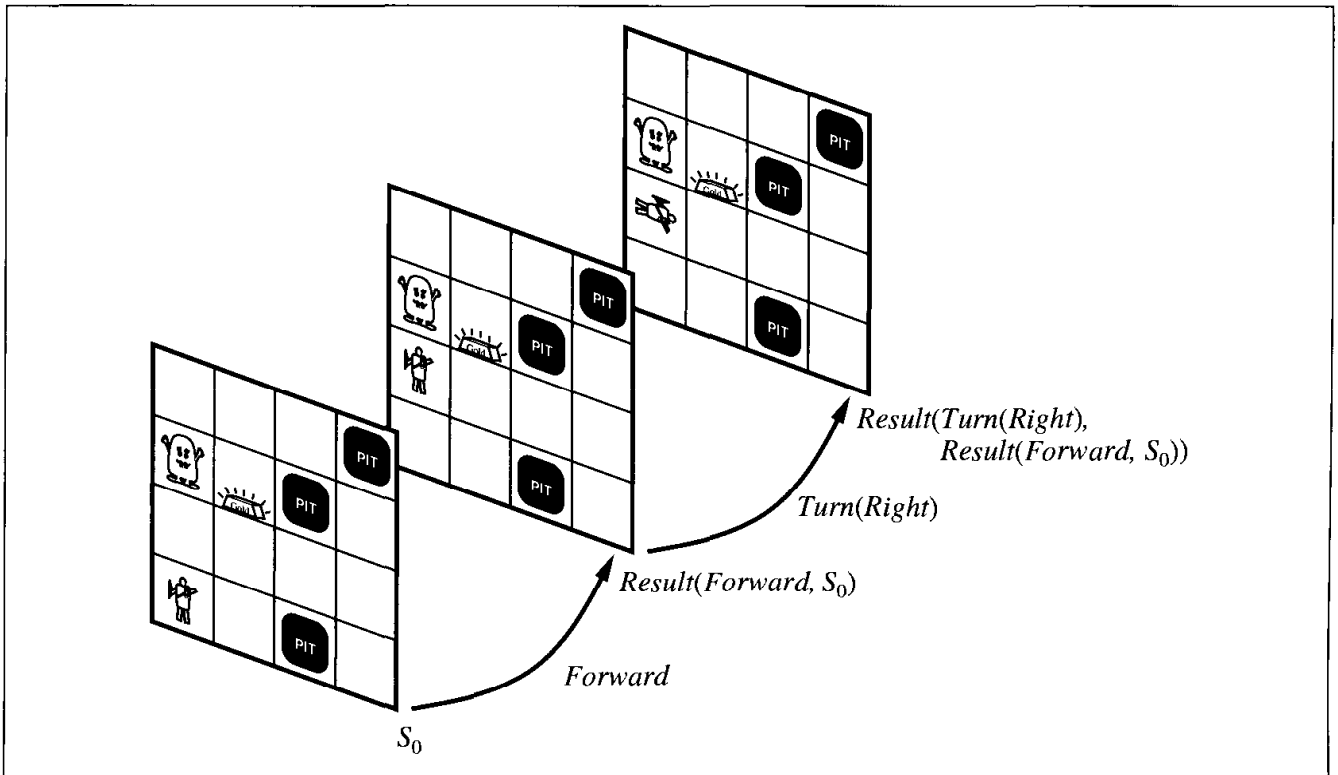


Figure 10.2 In situation calculus, each situation (except S_0) is the result of an action.

In addition to single actions, it is also helpful to reason about action sequences. We can define the results of sequences in terms of the results of individual actions. First, we say that executing an empty sequence leaves the situation unchanged:

$$Result([\,], s) = s\ .$$

Executing a nonempty sequence is the same as executing the first action and then executing the rest in the resulting situation:

$$Result([a|seq], s) = Result(seq, Result(a, s))\ .$$

A situation calculus agent should be able to deduce the outcome of a given sequence of actions; this is the **projection** task. With a suitable constructive inference algorithm, it should also be able to *find* a sequence that achieves a desired effect; this is the **planning** task.

PROJECTION
PLANNING

We will use an example from a modified version of the wumpus world where we do not worry about the agent's orientation and where the agent can Go from one location to an adjacent one. Suppose the agent is at $[1,1]$ and the gold is at $[1,2]$. The aim is to have the gold in $[1,1]$. The fluent predicates are $At(o,x,s)$ and $Holding(o,s)$. Then the initial knowledge base might include the following description:

$$At(Agent, [1,1], S_0) \wedge At(G_1, [1,2], S_0)\ .$$

This is not quite enough, however, because it doesn't say what what *isn't* true in S_0. (See page 355 for further discussion of this point.) The complete description is as follows:

$$At(o,x,S_0) \Leftrightarrow [(o = Agent \wedge x = [1,1]) \vee (o = G_1 \wedge x = [1,2])]\ .$$
$$\neg Holding(o, S_0)\ .$$

We also need to state that G_1 is gold and that $[1,1]$ and $[1,2]$ are adjacent:

$$Gold(G_1) \wedge Adjacent([1,1],[1,2]) \wedge Adjacent([1,2],[1,1])\ .$$

One would like to be able to prove that the agent achieves its aim by going to $[1,2]$, grabbing the gold, and returning to $[1,1]$. That is,

$$At(G_1, [1,1], Result([Go([1,1],[1,2]), Grab(G_1), Go([1,2],[1,1])], S_0))\ .$$

More interesting is the possibility of constructing a plan to get the gold, which is achieved by answering the query "what sequence of actions results in the gold being at [1,1]?"

$$\exists\, seq\ \ At(G_1, [1,1], Result(seq, S_0))\ .$$

Let us see what has to go into the knowledge base for these queries to be answered.

Describing actions in situation calculus

In the simplest version of situation calculus, each action is described by two axioms: a **possibility axiom** that says when it is possible to execute the action, and an **effect axiom** that says what happens when a possible action is executed. We will use $Poss(a,s)$ to mean that it is possible to execute action a in situation s. The axioms have the following form:

POSSIBILITY AXIOM
EFFECT AXIOM

POSSIBILITY AXIOM: $Preconditions \Rightarrow Poss(a,s)$.
EFFECT AXIOM: $Poss(a,s) \Rightarrow$ *Changes that result from taking action.*

We will present these axioms for the modified wumpus world. To shorten our sentences, we will omit universal quantifiers whose scope is the entire sentence. We assume that the variable s ranges over situations, a ranges over actions, o over objects (including agents), g over gold, and x and y over locations.

The possibility axioms for this world state that an agent can go between adjacent locations, grab a piece of gold in the current location, and release some gold that it is holding:

$$\begin{aligned} At(Agent, x, s) \wedge Adjacent(x, y) &\Rightarrow Poss(Go(x, y), s) . \\ Gold(g) \wedge At(Agent, x, s) \wedge At(g, x, s) &\Rightarrow Poss(Grab(g), s) . \\ Holding(g, s) &\Rightarrow Poss(Release(g), s) . \end{aligned}$$

The effect axioms state that, if an action is possible, then certain properties (fluents) will hold in the situation that results from executing the action. Going from x to y results in being at y, grabbing the gold results in holding the gold, and releasing the gold results in not holding it:

$$\begin{aligned} Poss(Go(x, y), s) &\Rightarrow At(Agent, y, Result(Go(x, y), s)) . \\ Poss(Grab(g), s) &\Rightarrow Holding(g, Result(Grab(g), s)) . \\ Poss(Release(g), s) &\Rightarrow \neg Holding(g, Result(Release(g), s)) . \end{aligned}$$

Having stated these axioms, can we prove that our little plan achieves the goal? Unfortunately not! At first, everything works fine; $Go([1,1],[1,2])$ is indeed possible in S_0 and the effect axiom for Go allows us to conclude that the agent reaches [1,2]:

$$At(Agent, [1,2], Result(Go([1,1],[1,2]), S_0)) .$$

Now we consider the $Grab(G_1)$ action. We have to show that it is possible in the new situation—that is,

$$At(G_1, [1,2], Result(Go([1,1],[1,2]), S_0)) .$$

Alas, nothing in the knowledge base justifies such a conclusion. Intuitively, we understand that the agent's Go action should have no effect on the gold's location, so it should still be at $[1,2]$, where it was in S_0. *The problem is that the effect axioms say what changes, but don't say what stays the same.*

FRAME PROBLEM

Representing all the things that stay the same is called the **frame problem**. We must find an efficient solution to the frame problem because, in the real world, almost everything stays the same almost all the time. Each action affects only a tiny fraction of all fluents.

FRAME AXIOM

One approach is to write explicit **frame axioms** that *do* say what stays the same. For example, the agent's movements leave other objects stationary unless they are held:

$$At(o, x, s) \wedge (o \neq Agent) \wedge \neg Holding(o, s) \Rightarrow At(o, x, Result(Go(y, z), s)) .$$

If there are F fluent predicates and A actions, then we will need $O(AF)$ frame axioms. On the other hand, if each action has at most E effects, where E is typically much less than F, then we should be able to represent what happens with a much smaller knowledge base of size $O(AE)$. This is the **representational frame problem**. The closely related **inferential frame problem** is to project the results of a t-step sequence of actions in time $O(Et)$, rather than time $O(Ft)$ or $O(AEt)$. We will address each problem in turn. Even then, another problem remains—that of ensuring that *all* necessary conditions for an action's success have been specified. For example, Go fails if the agent dies *en route*. This is the **qualification problem**, for which there is no complete solution.

REPRESENTATIONAL FRAME PROBLEM

INFERENTIAL FRAME PROBLEM

QUALIFICATION PROBLEM

Solving the representational frame problem

The solution to the representational frame problem involves just a slight change in viewpoint on how to write the axioms. Instead of writing out the effects of each action, we consider how each fluent predicate evolves over time.[3] The axioms we use are called **successor-state axioms**. They have the following form:

SUCCESSOR-STATE AXIOM

SUCCESSOR-STATE AXIOM:
Action is possible $\Rightarrow$
(*Fluent is true in result state* $\Leftrightarrow$ *Action's effect made it true*
$\vee$ *It was true before and action left it alone*) .

After the qualification that we are not considering impossible actions, notice that this definition uses $\Leftrightarrow$, not $\Rightarrow$. This means that the axiom says that the fluent will be true if *and only if* the right-hand side holds. Put another way, we are specifying the truth value of each fluent in the next state as a function of the action and the truth value in the current state. This means that the next state is completely specified from the current state and hence that there are no additional frame axioms needed.

The successor-state axiom for the agent's location says that the agent is at y after executing an action either if the action is possible and consists of moving to y or if the agent was already at y and the action is not a move to somewhere else:

$$Poss(a, s) \Rightarrow$$
$$(At(Agent, y, Result(a, s)) \Leftrightarrow a = Go(x, y) \vee (At(Agent, y, s) \wedge a \neq Go(y, z))) .$$

The axiom for *Holding* says that the agent is holding g after executing an action if the action was a grab of g and the grab is possible or if the agent was already holding g, and the action is not releasing it:

$$Poss(a, s) \Rightarrow$$
$$(Holding(g, Result(a, s)) \Leftrightarrow a = Grab(g) \vee (Holding(g, s) \wedge a \neq Release(g))) .$$

Successor-state axioms solve the representational frame problem because the total size of the axioms is $O(AE)$ literals: each of the E effects of each of the A actions is mentioned exactly once. The literals are spread over F different axioms, so the axioms have average size AE/F.

The astute reader will have noticed that these axioms handle the At fluent for the agent, but not for the gold; thus, we still cannot prove that the three-step plan achieves the goal of having the gold in $[1, 1]$. We need to say that an **implicit effect** of an agent moving from x to y is that any gold it is carrying will move too (as will any ants on the gold, any bacteria on the ants, etc.). Dealing with implicit effects is called the **ramification problem**. We will discuss the problem in general later, but for this specific domain, it can be solved by writing a more general successor-state axiom for At. The new axiom, which subsumes the previous version, says that an object o is at y if the agent went to y and o is the agent or something the

IMPLICIT EFFECT

RAMIFICATION PROBLEM

[3] This is essentially the approach we took in building the Boolean circuit-based agent in Chapter 7. Indeed, axioms such as Equation (7.4) and Equation (7.5) can be viewed as successor-state axioms.

agent was holding; or if o was already at y and the agent didn't go elsewhere, with o being the agent or something the agent was holding.

$$\begin{aligned} &Poss(a, s) \Rightarrow \\ &At(o, y, Result(a, s)) \quad \Leftrightarrow \quad (a = Go(x, y) \wedge (o = Agent \vee Holding(o, s))) \\ &\qquad\qquad \vee (At(o, y, s) \wedge \neg(\exists z \;\; y \neq z \wedge a = Go(y, z) \wedge \\ &\qquad\qquad\quad (o = Agent \vee Holding(o, s)))) \; . \end{aligned}$$

There is one more technicality: an inference process that uses these axioms must be able to prove nonidentities. The simplest kind of nonidentity is between constants—for example, $Agent \neq G_1$. The general semantics of first-order logic allows distinct constants to refer to the same object, so the knowledge base must include an axiom to prevent this. The **unique names axiom** states a disequality for every pair of constants in the knowledge base. When this is assumed by the theorem prover, rather than written down in the knowledge base, it is called a **unique names assumption**. We also need to state disequalities between action terms: $Go([1, 1], [1, 2])$ is a different action from $Go([1, 2], [1, 1])$ or $Grab(G_1)$. First, we say that each type of action is distinct—that no Go action is a $Grab$ action. For each pair of action names A and B, we would have

UNIQUE NAMES AXIOM

$$A(x_1, \ldots, x_m) \neq B(y_1, \ldots, y_n) \; .$$

Next, we say that two action terms with the same action name refer to the same action only if they involve all the same objects:

$$A(x_1, \ldots, x_m) = A(y_1, \ldots, y_m) \;\Leftrightarrow\; x_1 = y_1 \wedge \ldots \wedge x_m = y_m \; .$$

These are called, collectively, the **unique action axioms**. The combination of initial state description, successor-state axioms, unique name axiom, and unique action axioms suffices to prove that the proposed plan achieves the goal.

UNIQUE ACTION AXIOMS

Solving the inferential frame problem

Successor-state axioms solve the representational frame problem, but not the inferential frame problem. Consider a t-step plan p such that $S_t = Result(p, S_0)$. To decide which fluents are true in S_t, we need to consider each of the F frame axioms on each of the t time steps. Because the axioms have average size AE/F, this gives us $O(AEt)$ inferential work. Most of the work involves copying fluents unchanged from one situation to the next.

To solve the inferential frame problem, we have two possibilities. First, we could discard situation calculus and invent a new formalism for writing axioms. This has been done with formalisms such as the **fluent calculus**. Second, we could alter the inference mechanism to handle frame axioms more efficiently. A hint that this should be possible is that the simple approach is $O(AEt)$; why should it depend on the number of actions, A, when we know exactly which one action is executed at each time step? To see how to improve matters, we first look at the format of the frame axioms:

$$\begin{aligned} &Poss(a, s) \Rightarrow \\ &F_i(Result(a, s)) \quad \Leftrightarrow \quad (a = A_1 \vee a = A_2 \ldots) \\ &\qquad\qquad\qquad\qquad \vee F_i(s) \wedge (a \neq A_3) \wedge (a \neq A_4) \ldots \end{aligned}$$

That is, each axiom mentions several actions that can make the fluent true and several actions that can make it false. We can formalize this by introducing the predicate $PosEffect(a, F_i)$, meaning that action a causes F_i to become true, and $NegEffect(a, F_i)$ meaning that a causes F_i to become false. Then we can rewrite the foregoing axiom schema as:

$$Poss(a, s) \Rightarrow$$
$$F_i(Result(a, s)) \Leftrightarrow PosEffect(a, F_i) \vee [F_i(s) \wedge \neg NegEffect(a, F_i)]$$
$$PosEffect(A_1, F_i)$$
$$PosEffect(A_2, F_i)$$
$$NegEffect(A_3, F_i)$$
$$NegEffect(A_4, F_i) .$$

Whether this can be done automatically depends on the exact format of the frame axioms. To make an efficient inference procedure using axioms like this, we need to do three things:

1. Index the $PosEffect$ and $NegEffect$ predicates by their first argument so that when we are given an action that occurs at time t, we can find its effects in $O(1)$ time.
2. Index the axioms so that once you know that F_i is an effect of an action, you can find the axiom for F_i in $O(1)$ time. Then you need not even consider the axioms for fluents that are not an effect of the action.
3. Represent each situation as a previous situation plus a delta. Thus, if nothing changes from one step to the next, we need do no work at all. In the old approach, we would need to do $O(F)$ work in generating an assertion for each fluent $F_i(Result(a, s))$ from the preceding $F_i(s)$ assertions.

Thus at each time step, we look at the current action, fetch its effects, and update the set of true fluents. Each time step will have an average of E of these updates, for a total complexity of $O(Et)$. This constitutes a solution to the inferential frame problem.

Time and event calculus

Situation calculus works well when there is a single agent performing instantaneous, discrete actions. When actions have duration and can overlap with each other, situation calculus becomes somewhat awkward. Therefore, we will cover those topics with an alternative formalism known as **event calculus**, which is based on points in time rather than on situations. (The terms "event" and "action" may be used interchangeably. Informally, "event" connotes a wider class of actions, including ones with no explicit agent. These are easier to handle in event calculus than in situation calculus.)

EVENT CALCULUS

In event calculus, fluents hold at points in time rather than at situations, and the calculus is designed to allow reasoning over intervals of time. The event calculus axiom says that a fluent is true at a point in time if the fluent was initiated by an event at some time in the past and was not terminated by an intervening event. The $Initiates$ and $Terminates$ relations play a role similar to the $Result$ relation in situation calculus; $Initiates(e, f, t)$ means that the occurrence of event e at time t causes fluent f to become true, while $Terminates(w, f, t)$ means that f ceases to be true. We use $Happens(e, t)$ to mean that event e happens at time t,

and we use $Clipped(f, t, t_2)$ to mean that f is terminated by some event sometime between t and t_2. Formally, the axiom is:

EVENT CALCULUS AXIOM:

$$T(f, t_2) \Leftrightarrow \exists e, t \;\; Happens(e, t) \wedge Initiates(e, f, t) \wedge (t < t_2) \wedge \neg Clipped(f, t, t_2)$$

$$Clipped(f, t, t_2) \Leftrightarrow \exists e, t_1 \;\; Happens(e, t_1) \wedge Terminates(e, f, t_1) \wedge (t < t_1) \wedge (t_1 < t_2) \; .$$

This gives us functionality that is similar to situation calculus, but with the ability to talk about time points and intervals, so we can say $Happens(TurnOff(LightSwitch_1), 1{:}00)$ to say that a lightswitch was turned off at exactly 1:00.

Many extensions to event calculus have been made to address problems of indirect effects, events with duration, concurrent events, continuously changing events, nondeterministic effects, causal constraints, and other complications. We will revisit some of these issues in the next subsection. It is fair to say that, at present, completely satisfactory solutions are not yet available for most of them, but no insuperable obstacles have been encountered.

Generalized events

So far, we have looked at two main concepts: actions and objects. Now it is time to see how they fit into an encompassing ontology in which both actions and objects can be thought of as aspects of a physical universe. We think of a particular universe as having both a spatial and a temporal dimension. The wumpus world has its spatial component laid out in a two-dimensional grid and has discrete time; our world has three spatial dimensions and one temporal dimension,[3] all continuous. A **generalized event** is composed from aspects of some "space–time chunk"—a piece of this multidimensional space–time universe. This abstraction generalizes most of the concepts we have seen so far, including actions, locations, times, fluents, and physical objects. Figure 10.3 gives the general idea. From now on, we will use the simple term "event" to refer to generalized events.

GENERALIZED EVENT

For example, World War II is an event that took place at various points in space–time, as indicated by the irregularly shaped grey patch. We can break it down into **subevents**:[4]

SUBEVENTS

$$SubEvent(BattleOfBritain, WorldWarII) \; .$$

Similarly, World War II is a subevent of the 20th century:

$$SubEvent(WorldWarII, TwentiethCentury) \; .$$

The 20th century is an *interval* of time. Intervals are chunks of space–time that include all of space between two time points. The function $Period(e)$ denotes the smallest interval enclosing the event e. $Duration(i)$ is the length of time occupied by an interval, so we can say $Duration(Period(WorldWarII)) > Years(5)$.

[3] Some physicists studying string theory argue for 10 dimensions or more, and some argue for a discrete world, but 4-D continuous space–time is an adequate representation for commonsense reasoning purposes.

[4] Note that *SubEvent* is a special case of the *PartOf* relation and is also transitive and reflexive.

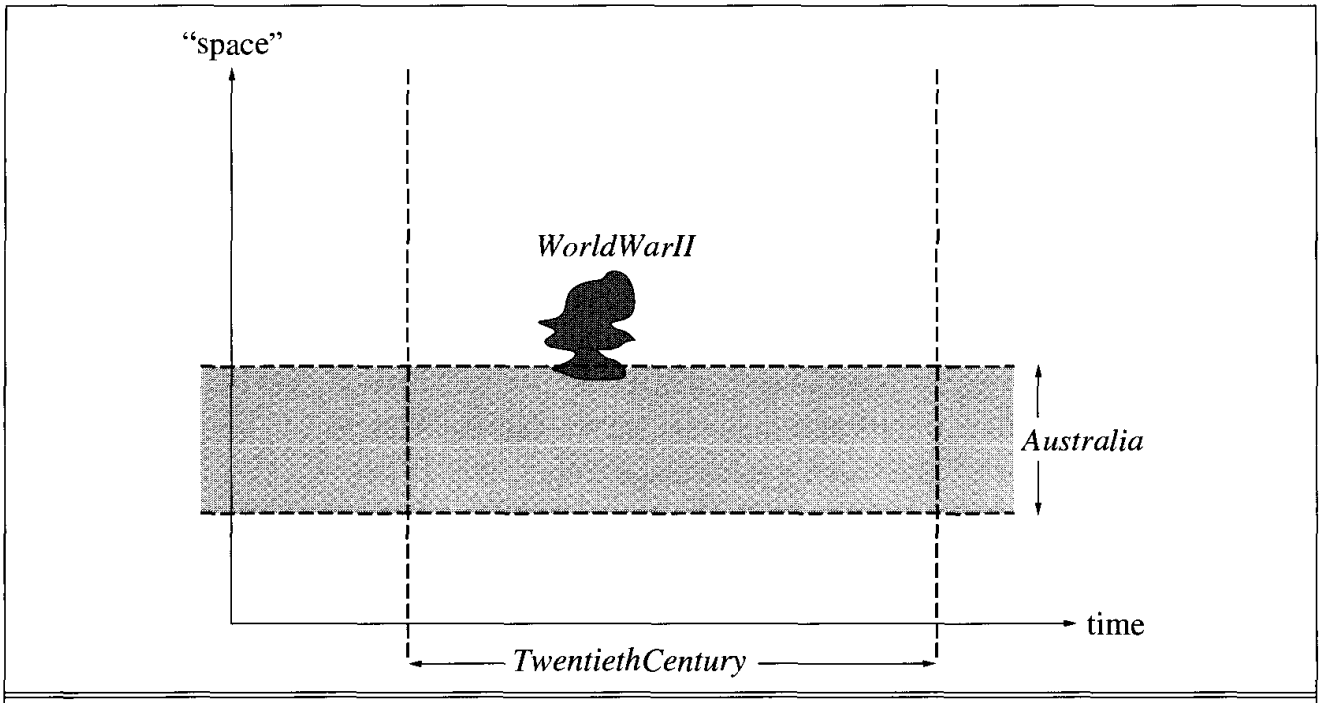


Figure 10.3 Generalized events. A universe has spatial and temporal dimensions; in this figure we show only a single spatial dimension. All events are *PartOf* the universe. An event, such as $WorldWarII$, occurs in a portion of space–time with somewhat arbitrary and time-varying borders. An $Interval$, such as the $TwentiethCentury$, has a fixed, limited temporal extent and maximal spatial extent, and a $Place$, such as Australia, has a roughly fixed spatial extent and maximal temporal extent.

Australia is a *place*; a chunk with some fixed spatial borders. The borders can vary over time, due to geological or political changes. We use the predicate In to denote the subevent relation that holds when one event's spatial projection is $PartOf$ of another's:

$$In(Sydney, Australia) .$$

The function $Location(e)$ denotes the smallest place that encloses the event e.

Like any other sort of object, events can be grouped into categories. For example, $WorldWarII$ belongs to the category $Wars$. To say that a civil war occurred in England in the 1640s, we would say

$$\exists w \;\; w \in CivilWars \wedge SubEvent(w, 1640s) \wedge In(Location(w), England) .$$

The notion of a category of events answers a question that we avoided when we described the effects of actions in Section 10.3: what exactly do logical terms such as $Go([1,1],[1,2])$ refer to? Are they events? The answer, perhaps surprisingly, is *no*. We can see this by considering a plan with two "identical" actions, such as

$$[Go([1,1],[1,2]), Go([1,2],[1,1]), Go([1,1],[1,2])] .$$

In this plan, $Go([1,1],[1,2])$ cannot be the name of an event, because there are *two different events* occurring at different times. Instead, $Go([1,1],[1,2])$ is the name of a *category* of events—all those events where the agent goes from $[1,1]$ to $[1,2]$. The three-step plan says that instances of these three event categories will occur.

Notice that this is the first time we have seen categories named by complex terms rather than just constant symbols. This presents no new difficulties; in fact, we can use the argument structure to our advantage. Eliminating arguments creates a more general category:

$$Go(x, y) \subseteq GoTo(y) \qquad Go(x, y) \subseteq GoFrom(x) .$$

Similarly, we can add arguments to create more specific categories. For example, to describe actions by other agents, we can add an agent argument. Thus, to say that Shankar flew from New York to New Delhi yesterday, we would write:

$$\exists e \;\; e \in Fly(Shankar, NewYork, NewDelhi) \wedge SubEvent(e, Yesterday) .$$

The form of this formula is so common that we will create an abbreviation for it: $E(c, i)$ will mean that an element of the category of events c is a subevent of the event or interval i:

$$E(c, i) \;\Leftrightarrow\; \exists e \;\; e \in c \wedge SubEvent(e, i) .$$

Thus, we have:

$$E(Fly(Shankar, NewYork, NewDelhi), Yesterday) .$$

Processes

DISCRETE EVENTS

The events we have seen so far are what we call **discrete events**—they have a definite structure. Shankar's trip has a beginning, middle, and end. If interrupted halfway, the event would be different—it would not be a trip from New York to New Delhi, but instead a trip from New York to somewhere over Europe. On the other hand, the category of events denoted by $Flying(Shankar)$ has a different quality. If we take a small interval of Shankar's flight, say, the third 20-minute segment (while he waits anxiously for a second bag of peanuts), that event is still a member of $Flying(Shankar)$. In fact, this is true for any subinterval.

PROCESS
LIQUID EVENT

Categories of events with this property are called **process** categories or **liquid event** categories. Any subinterval of a process is also a member of the same process category. We can employ the same notation used for discrete events to say that, for example, Shankar was flying at some time yesterday:

$$E(Flying(Shankar), Yesterday) .$$

We often want to say that some process was going on *throughout* some interval, rather than just in some subinterval of it. To do this, we use the predicate T:

$$T(Working(Stuart), TodayLunchHour) .$$

$T(c, i)$ means that some event of type c occurred over exactly the interval i—that is, the event begins and ends at the same time as the interval.

The distinction between liquid and nonliquid events is exactly analogous to the difference between substances, or *stuff*, and individual objects. In fact, some have called liquid event types **temporal substances**, whereas things like butter are **spatial substances**.

TEMPORAL SUBSTANCES
SPATIAL SUBSTANCES
STATES

As well as describing processes of continuous change, liquid events can describe processes of continuous non-change. These are often called **states**. For example, "Shankar being in New York" is a category of states that we denote by $In(Shankar, NewYork)$. To say he was in New York all day, we would write

$$T(In(Shankar, NewYork), Today) .$$

FLUENT CALCULUS

We can form more complex states and events by combining primitive ones. This approach is called **fluent calculus**. Fluent calculus reifies combinations of fluents, not just individual fluents. We have already seen a way of representing the event of two things happening at once, namely, the function $Both(e_1, e_2)$. In fluent calculus, this is usually abbreviated with the infix notation $e_1 \circ e_2$. For example, to say that someone walked and chewed gum at the same time, we can write

$$\exists p, i \;\; (p \in People) \wedge T(Walk(p) \circ ChewGum(p), i) \; .$$

The "$\circ$" function is commutative and associative, just like logical conjunction. We can also define analogs of disjunction and negation, but we have to be more careful—there are two reasonable ways of interpreting disjunction. When we say "the agent was either walking or chewing gum for the last two minutes" we might mean that the agent was doing one of the actions for the whole interval, or we might mean that the agent was alternating between the two actions. We will use *OneOf* and *Either* to indicate these two possibilities. Figure 10.4 diagrams the complex events.

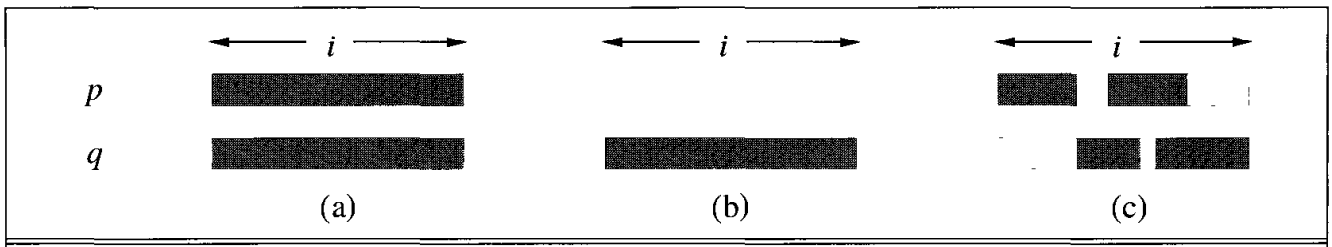


Figure 10.4 A depiction of complex events. (a) $T(Both(p.\, q).\, i)$, also denoted as $T(p \circ q, i)$. (b) $T(OneOf(p, q), i)$. (c) $T(Either(p.\, q).\, i)$.

Intervals

Time is important to any agent that takes action, and there has been much work on the representation of time intervals. We will consider two kinds: moments and extended intervals. The distinction is that only moments have zero duration:

$$Partition(\{Moments, ExtendedIntervals\}, Intervals)$$
$$i \in Moments \;\Leftrightarrow\; Duration(i) = Seconds(0) \; .$$

Next we invent a time scale and associate points on that scale with moments, giving us absolute times. The time scale is arbitrary; we will measure it in seconds and say that the moment at midnight (GMT) on January 1, 1900, has time 0. The functions *Start* and *End* pick out the earliest and latest moments in an interval, and the function *Time* delivers the point on the time scale for a moment. The function *Duration* gives the difference between the end time and the start time.

$$Interval(i) \;\Rightarrow\; Duration(i) = (Time(End(i)) - Time(Start(i))) \; .$$
$$Time(Start(AD1900)) = Seconds(0) \; .$$
$$Time(Start(AD2001)) = Seconds(3187324800) \; .$$
$$Time(End(AD2001)) = Seconds(3218860800) \; .$$
$$Duration(AD2001) = Seconds(31536000) \; .$$

To make these numbers easier to read, we also introduce a function *Date*, which takes six arguments (hours, minutes, seconds, day, month, and year) and returns a time point:

$$Time(Start(AD2001)) = Date(0, 0, 0, 1, Jan, 2001)$$
$$Date(0, 20, 21, 24, 1, 1995) = Seconds(3000000000) .$$

Two intervals *Meet* if the end time of the first equals the start time of the second. It is possible to define predicates such as *Before*, *After*, *During*, and *Overlap* solely in terms of *Meet*, but it is more intuitive to define them in terms of points on the time scale. (See Figure 10.5 for a graphical representation.)

$$\begin{aligned}
Meet(i, j) &\Leftrightarrow Time(End(i)) = Time(Start(j)) . \\
Before(i, j) &\Leftrightarrow Time(End(i)) < Time(Start(j)) . \\
After(j, i) &\Leftrightarrow Before(i, j) . \\
During(i, j) &\Leftrightarrow Time(Start(j)) \leq Time(Start(i)) \\
&\quad \wedge Time(End(i)) \leq Time(End(j)) . \\
Overlap(i, j) &\Leftrightarrow \exists k \;\; During(k, i) \wedge During(k, j) .
\end{aligned}$$

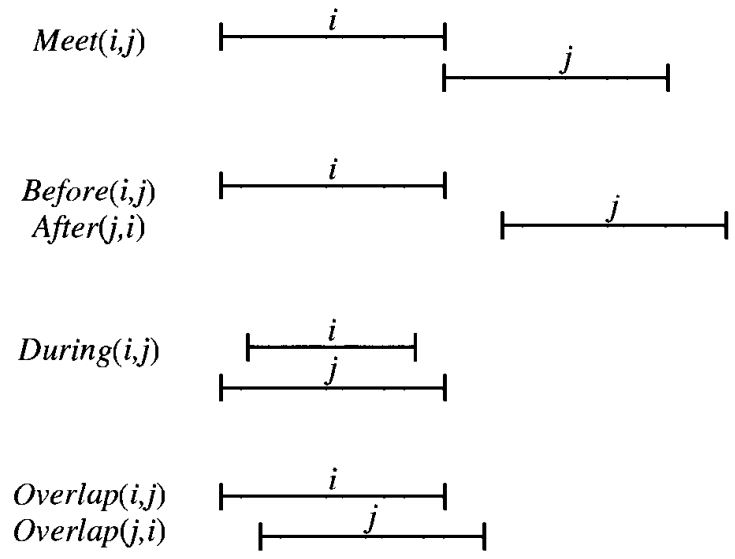


Figure 10.5 Predicates on time intervals.

For example, to say that the reign of Elizabeth II followed that of George VI, and the reign of Elvis overlapped with the 1950s, we can write the following:

$$After(ReignOf(ElizabethII), ReignOf(GeorgeVI)) .$$
$$Overlap(Fifties, ReignOf(Elvis)) .$$
$$Start(Fifties) = Start(AD1950) .$$
$$End(Fifties) = End(AD1959) .$$

Fluents and objects

We mentioned that physical objects can be viewed as generalized events, in the sense that a physical object is a chunk of space–time. For example, *USA* can be thought of as an event that began in, say, 1776 as a union of 13 states and is still in progress today as a union of 50.

We can describe the changing properties of USA using state fluents. For example, we can say that at some point in 1999 its population was 271 million:

$$E(Population(USA, 271000000), AD1999) .$$

Another property of the USA that changes every four or eight years, barring mishaps, is its president. One might propose that $President(USA)$ is a logical term that denotes a different object at different times. Unfortunately, this is not possible, because a term denotes exactly one object in a given model structure. (The term $President(USA, t)$ can denote different objects, depending on the value of t, but our ontology keeps time indices separate from fluents.) The only possibility is that $President(USA)$ denotes a single object that consists of different people at different times. It is the object that is George Washington from 1789 to 1796, John Adams from 1796 to 1800, and so on, as in Figure 10.6.

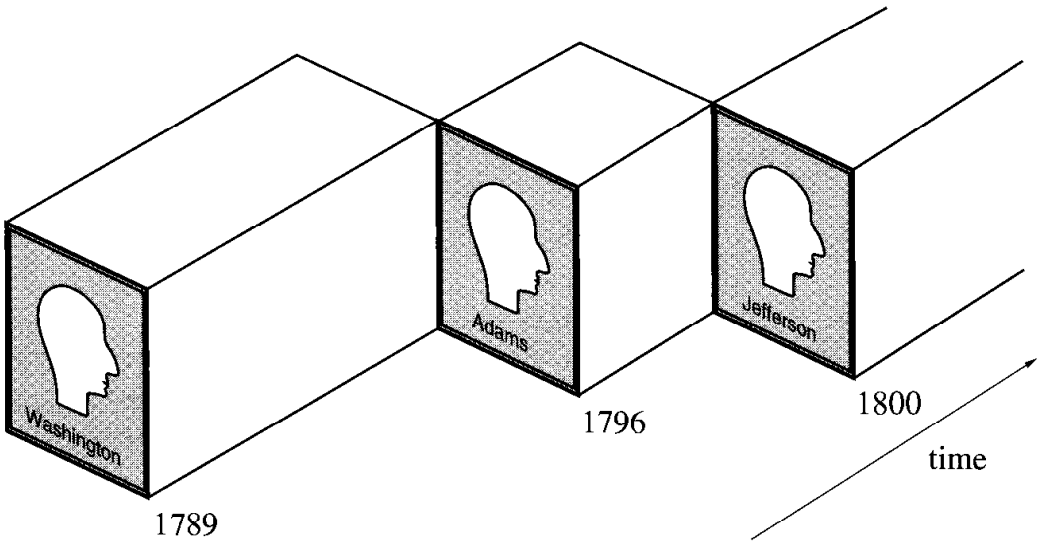


Figure 10.6 A schematic view of the object $President(USA)$ for the first 15 years of its existence.

To say that George Washington was president throughout 1790, we can write

$$T(President(USA) = GeorgeWashington, AD1790) .$$

We need to be careful, however. In this sentence, "=" must be a function symbol rather than the standard logical operator. The interpretation is *not* that $GeorgeWashington$ and $President(USA)$ are logically identical in 1790; logical identity is not something that can change over time. The logical identity exists between the subevents of each object that are defined by the period 1790.

Don't confuse the physical object $GeorgeWashington$ with a collections of atoms. George Washington is not logically identical to *any* specific collection of atoms, because the set of atoms of which he is constituted varies considerably over time. He has his short lifetime, and each atom has its own very long lifetime. They intersect for some period, during which the temporal slice of the atom is $PartOf$ George, and then they go their separate ways.

10.4 MENTAL EVENTS AND MENTAL OBJECTS

The agents we have constructed so far have beliefs and can deduce new beliefs. Yet none of them has any knowledge *about* beliefs or *about* deduction. For single-agent domains, knowledge about one's own knowledge and reasoning processes is useful for controlling inference. For example, if one knows that one does not know anything about Romanian geography, then one need not expend enormous computational effort trying to calculate the shortest path from Arad to Bucharest. One can also reason about one's own knowledge in order to construct plans that will change it—for example by buying a map of Romania. In multiagent domains, it becomes important for an agent to reason about the mental states of the other agents. For example, a Romanian police officer might well know the best way to get to Bucharest, so the agent might ask for help.

In essence, what we need is a model of the mental objects that are in someone's head (or something's knowledge base) and of the mental processes that manipulate those mental objects. The model should be faithful, but it does not have to be detailed. We do not have to be able to predict how many milliseconds it will take for a particular agent to make a deduction, nor do we have to predict what neurons will fire when an animal is faced with a particular visual stimulus. We will be happy to conclude that the Romanian police officer will tell us how to get to Bucharest if he or she knows the way and believes we are lost.

A formal theory of beliefs

We begin with the relationships between agents and "mental objects"—relationships such as *Believes*, *Knows*, and *Wants*. Relations of this kind are called **propositional attitudes**, because they describe an attitude that an agent can take toward a proposition. Suppose that Lois believes something—that is, $Believes(Lois, x)$. What kind of thing is x? Clearly, x cannot be a logical sentence. If $Flies(Superman)$ is a logical sentence, we can't say $Believes(Lois, Flies(Superman))$, because only terms (not sentences) can be arguments of predicates. But if *Flies* is a function, then $Flies(Superman)$ is a candidate for being a mental object, and *Believes* can be a relation between an agent and a propositional fluent. Turning a proposition into an object is called **reification**.[6]

PROPOSITIONAL ATTITUDE

REIFICATION

This appears to give us what we want: the ability for an agent to reason about the beliefs of agents. Unfortunately, there is a problem with that approach: If Clark and Superman are one and the same (i.e., $Clark = Superman$) then Clark's flying and Superman's flying are one and the same event category, i.e., $Flies(Clark) = Flies(Superman)$. Hence, we must conclude that if Lois believes that Superman can fly, she also believes that Clark can fly, *even if she doesn't believe that Clark is Superman.* That is,

$$(Superman = Clark) \models (Believes(Lois, Flies(Superman)) \Leftrightarrow Believes(Lois, Flies(Clark))) .$$

There is a sense in which this is right: Lois does believe of a certain person, who happens

[6] The term "reification" comes from the Latin word *res*, or thing. John McCarthy proposed the term "thingification," but it never caught on.

to be called Clark sometimes, that that person can fly. But there is another sense in which it is wrong: if you asked Lois "Can Clark fly?" she would certainly say no. Reified objects and events work fine for the first sense of *Believes*, but for the second sense we need to reify *descriptions* of those objects and events, so that *Clark* and *Superman* can be different descriptions (even though they refer to the same object).

REFERENTIAL TRANSPARENCY

Technically, the property of being able to substitute a term freely for an equal term is called **referential transparency**. In first-order logic, every relation is referentially transparent. We would like to define *Believes* (and the other propositional attitudes) as relations whose second argument is referentially **opaque**—that is, one cannot substitute an equal term for the second argument without changing the meaning.

OPAQUE

MODAL LOGIC

MODAL OPERATOR

There are two ways to achieve this. The first is to use a different form of logic called **modal logic**, in which propositional attitudes such as *Believes* and *Knows* become **modal operators** that are referentially opaque. This approach is covered in the historical notes section. The second approach, which we will pursue, is to achieve effective opacity within a referentially transparent language using a **syntactic theory** of mental objects. This means that mental objects are represented by **strings**. The result is a crude model of an agent's knowledge base as consisting of strings that represent sentences believed by the agent. A string is just a complex term denoting a list of symbols, so the event $Flies(Clark)$ can be represented by the list of characters $[F, l, i, e, s, (, C, l, a, r, k,)]$, which we will abbreviate as a quoted string, "$Flies(Clark)$". The syntactic theory includes a **unique string axiom** stating that strings are identical if and only if they consist of identical characters. In this way, even if $Clark = Superman$, we still have "$Clark$" $\neq$ "$Superman$".

SYNTACTIC THEORY

STRINGS

UNIQUE STRING AXIOM

Now all we have to do is provide a syntax, semantics, and proof theory for the string representation language, just as we did in Chapter 7. The difference is that we have to define them all in first-order logic. We start by defining *Den* as the function that maps a string to the object that it denotes and *Name* as a function that maps an object to a string that is the name of a constant that denotes the object. For example, the denotation of both "$Clark$" and "$Superman$" is the object referred to by the constant symbol *ManOfSteel*, and the name of that object within the knowledge base could be either "$Superman$", "$Clark$", or some other constant, such as "X_{11}":

$$Den(\text{"}Clark\text{"}) = ManOfSteel \wedge Den(\text{"}Superman\text{"}) = ManOfSteel\ .$$
$$Name(ManOfSteel) = \text{"}X_{11}\text{"}\ .$$

The next step is to define inference rules for logical agents. For example, we might want to say that a logical agent can do Modus Ponens: if it believes p and believes $p \Rightarrow q$, then it will also believe q. The first attempt at writing this axiom is

$$LogicalAgent(a) \wedge Believes(a, p) \wedge Believes(a, \text{"}p \Rightarrow q\text{"}) \Rightarrow Believes(a, q)\ .$$

But this is not right because the string "$p \Rightarrow q$" contains the letters 'p' and 'q' but has nothing to do with the strings that are the values of the variables p and q. The correct formulation is

$$LogicalAgent(a) \wedge Believes(a, p) \wedge Believes(a, Concat(p, \text{"}\Rightarrow\text{"}, q)) \Rightarrow Believes(a, q)\ .$$

where *Concat* is a function on strings that concatenates their elements. We will abbreviate $Concat(p, \text{"}\Rightarrow\text{"}, q)$ as "$\underline{p} \Rightarrow \underline{q}$". That is, an occurrence of $\underline{x}$ within a string is **unquoted**,

UNQUOTED

meaning that we are to substitute in the value of the variable x. Lisp programmers will recognize this as the comma/backquote operator, and Perl programmers will recognize it as $-variable interpolation.

Once we add in the other inference rules besides Modus Ponens, we will be able to answer questions of the form "given that a logical agent knows these premises, can it draw that conclusion?" Besides the normal inference rules, we need some rules that are specific to belief. For example, the following rule says that if a logical agent believes something, then it believes that it believes it.

$$LogicalAgent(a) \wedge Believes(a, p) \Rightarrow Believes(a, \text{“}Believes(\underline{Name(a)}, \underline{p})\text{”}) .$$

LOGICAL OMNISCIENCE

Now, according to our axioms, an agent can deduce any consequence of its beliefs infallibly. This is called **logical omniscience**. A variety of attempts have been made to define limited rational agents, which can make a limited number of deductions in a limited time. None is completely satisfactory, but these formulations do allow a highly restricted range of predictions about limited agents.

Knowledge and belief

The relation between believing and knowing has been studied extensively in philosophy. It is commonly said that knowledge is justified true belief. That is, if you believe something for an unassailably good reason, and if it is actually true, then you know it. The "unassailably good reason" is necessary to prevent you from saying "I know this coin flip will come up heads" and being right half the time.

Let $Knows(a, p)$ mean that agent a *knows that* proposition p is true. It is also possible to define other kinds of knowing. For example, here is a definition of "knowing whether":

$$KnowsWhether(a, p) \Leftrightarrow Knows(a, p) \vee Knows(a, \text{“}\neg \underline{p}\text{”}) .$$

Continuing our example, Lois knows whether Clark can fly if she either knows that Clark can fly or knows that he cannot.

The concept of "knowing what" is more complicated. One is tempted to say that an agent knows what Bob's phone number is if there is some x for which the agent knows $PhoneNumber(Bob) = x$. But that is not enough, because the agent might know that Alice and Bob have the same number (i.e., $PhoneNumber(Bob) = PhoneNumber(Alice)$), but if Alice's number is unknown, that isn't much help. A better definition of "knowing what" says that the agent has to be aware of some x that is a string of digits and that is Bob's number:

$$KnowsWhat(a, \text{“}PhoneNumber(\underline{b})\text{”}) \Leftrightarrow$$
$$\exists x \;\; Knows(a, \text{“}\underline{x} = PhoneNumber(\underline{b})\text{”}) \wedge x \in DigitStrings .$$

Of course, for other questions we have different criteria for what is an acceptable answer. For the question "what is the capital of New York," an acceptable answer is a proper name, "Albany," not something like "the city where the state house is." To handle this, we will make $KnowsWhat$ a three-place relation: it takes an agent, a string representing a term, and a category to which the answer must belong. For example, we might have the following:

$$KnowsWhat(Agent, \text{“}Capital(NewYork)\text{”}, ProperNames) .$$
$$KnowsWhat(Agent, \text{“}PhoneNumber(Bob)\text{”}, DigitStrings) .$$

Knowledge, time, and action

In most real situations, an agent will be dealing with beliefs—its own or those of other agents—that change over time. The agent will also have to make plans that involve changes to its own beliefs, such as buying a map to find out how to get to Bucharest. As with other predicates, we can reify *Believes* and talk about beliefs occurring over some period. For example, to say that Lois believes today that Superman can fly, we write

$$T(Believes(Lois, \text{“}Flies(Superman)\text{”}), Today) .$$

If the object of belief is a proposition that can change over time, then it too can be described using the T operator within the string. For example, Lois might believe today that that Superman could fly yesterday:

$$T(Believes(Lois, \text{“}T(Flies(Superman), Yesterday)\text{”}, Today) .$$

KNOWLEDGE PRECONDITIONS
KNOWLEDGE EFFECTS

Given a way to describe beliefs over time, we can use the machinery of event calculus to make plans involving beliefs. Actions can have **knowledge preconditions** and **knowledge effects**. For example, the action of dialing a person's number has the precondition of knowing the number, and the action of looking up the number has the effect of knowing the number. We can describe the latter action using the machinery of event calculus:

$$Initiates(Lookup(a, \text{“}PhoneNumber(\underline{b})\text{”}),$$
$$KnowsWhat(a, \text{“}PhoneNumber(\underline{b})\text{”}, DigitStrings), t) .$$

RUNTIME VARIABLES

Plans to gather and use information are often represented using a shorthand notation called **runtime variables**, which is closely related to the unquoted-variable convention described earlier. For example, the plan to look up Bob's number and then dial it can be written as

$$[Lookup(Agent, \text{“}PhoneNumber(Bob)\text{”}, \underline{n}), Dial(\underline{n})] .$$

Here, $\underline{n}$ is a runtime variable whose value will be bound by the *Lookup* action and can then be used by the *Dial* action. Plans of this kind occur frequently in partially observable domains. We will see examples in the next section and in Chapter 12.

10.5 The Internet Shopping World

In this section we will encode some knowledge related to shopping on the Internet. We will create a shopping research agent that helps a buyer find product offers on the Internet. The shopping agent is given a product description by the buyer and has the task of producing a list of Web pages that offer such a product for sale. In some cases the buyer's product description will be precise, as in *Coolpix 995 digital camera*, and the task is then to find the store(s) with the best offer. In other cases the description will be only partially specified, as in *digital camera for under $300*, and the agent will have to compare different products.

The shopping agent's environment is the entire World Wide Web—not a toy simulated environment, but the same complex, constantly evolving environment that is used by millions of people every day. The agent's percepts are Web pages, but whereas a human Web user would see pages displayed as an array of pixels on a screen, the shopping agent will perceive

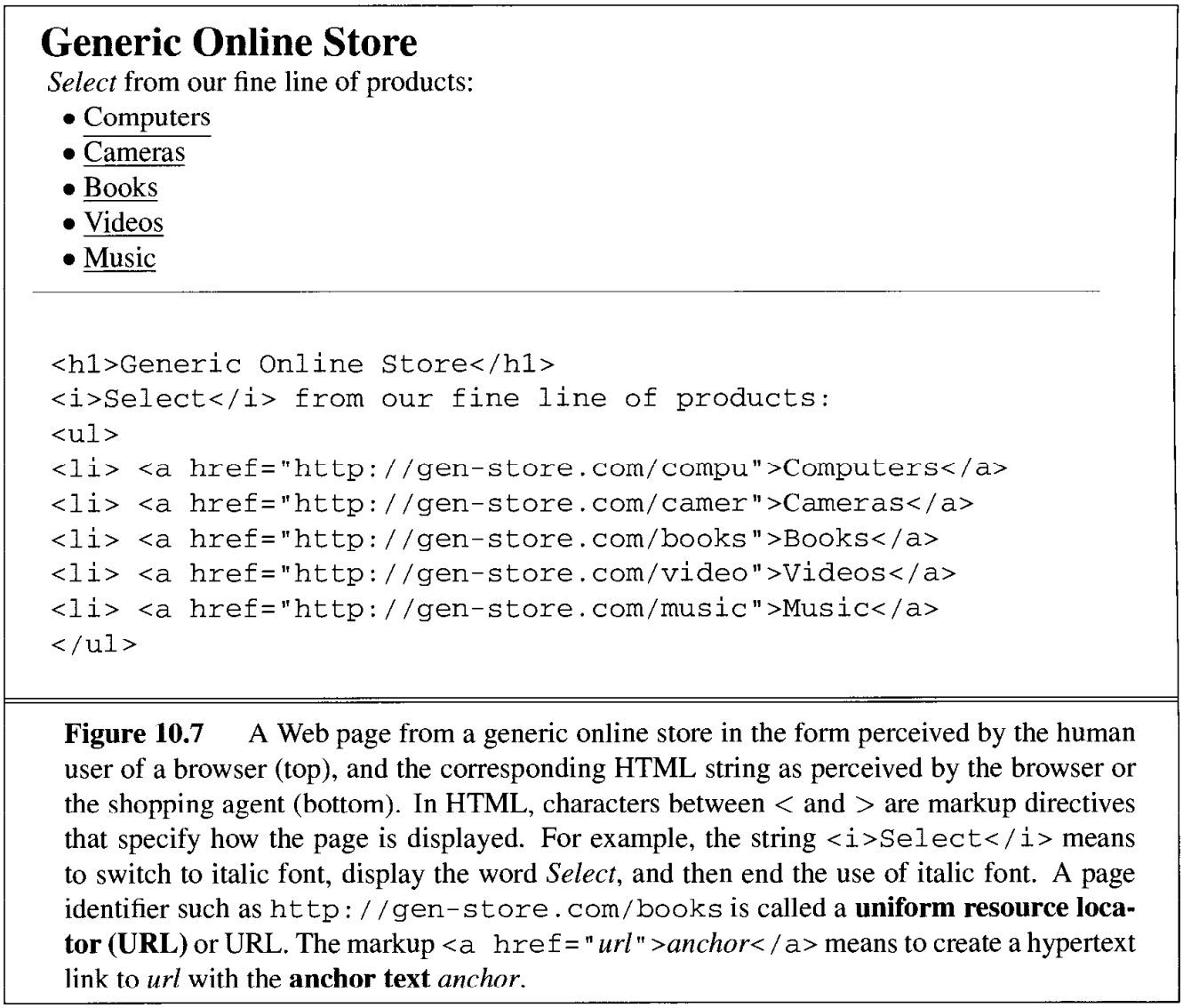


Figure 10.7 A Web page from a generic online store in the form perceived by the human user of a browser (top), and the corresponding HTML string as perceived by the browser or the shopping agent (bottom). In HTML, characters between < and > are markup directives that specify how the page is displayed. For example, the string `<i>Select</i>` means to switch to italic font, display the word *Select*, and then end the use of italic font. A page identifier such as `http://gen-store.com/books` is called a **uniform resource locator (URL)** or URL. The markup `<a href="`*url*`">`*anchor*`</a>` means to create a hypertext link to *url* with the **anchor text** *anchor*.

a page as a character string consisting of ordinary words interspersed with formatting commands in the HTML markup language. Figure 10.7 shows a Web page and a corresponding HTML character string. The perception problem for the shopping agent involves extracting useful information from percepts of this kind.

Clearly, perception on Web pages is easier than, say, perception while driving a taxi in Cairo. Nonetheless, there are complications to the Internet perception task. The web page in Figure 10.7 is very simple compared to real shopping sites, which include cookies, Java, Javascript, Flash, robot exclusion protocols, malformed HTML, sound files, movies, and text that appears only as part of a JPEG image. An agent that can deal with *all* of the Internet is almost as complex as a robot that can move in the real world. We will concentrate on a simple agent that ignores most of these complications.

The agent's first task is to find relevant product offers (we'll see later how to choose the best of the relevant offers). Let *query* be the product description that the user types in (e.g., "laptops"); then a page is a relevant offer for *query* if the page is relevant and the page is indeed an offer. We will also keep track of the URL associated with the page:

$$RelevantOffer(page, url, query) \Leftrightarrow Relevant(page, url, query) \wedge Offer(page) .$$

A page with a review of the latest high-end laptop would be relevant, but if it doesn't provide a way to buy, it isn't an offer. For now, we can say a page is an offer if it contains the word "buy" or "price" within an HTML link or form on the page. In other words, if the page contains a string of the form "`<a ...buy ...</a`" then it is an offer; it could also say "`price`" instead of "`buy`" or use "`form`" instead of "`a`". We can write axioms for this:

$$\begin{aligned}
Offer(page) &\Leftrightarrow (InTag(\text{“}a\text{”}, str, page) \vee InTag(\text{“}form\text{”}, str, page)) \\
&\quad \wedge (In(\text{“}buy\text{”}, str) \vee In(\text{“}price\text{”}, str)) . \\
InTag(tag, str, page) &\Leftrightarrow In(\text{“}<\text{”} + tag + str + \text{“}</\text{”} + tag, page) . \\
In(sub, str) &\Leftrightarrow \exists i \;\; str[i : i + Length(sub)] = sub .
\end{aligned}$$

Now we need to find relevant pages. The strategy is to start at the home page of an online store and consider all pages that can be reached by following relevant links.[7] The agent will have knowledge of a number of stores, for example:

$$\begin{aligned}
&Amazon \in OnlineStores \wedge Homepage(Amazon, \text{“}amazon.com\text{”}) . \\
&Ebay \in OnlineStores \wedge Homepage(Ebay, \text{“}ebay.com\text{”}) . \\
&GenStore \in OnlineStores \wedge Homepage(GenStore, \text{“}gen\text{-}store.com\text{”}) .
\end{aligned}$$

These stores classify their goods into product categories, and provide links to the major categories from their home page. Minor categories can be reached by following a chain of relevant links, and eventually we will reach offers. In other words, a page is relevant to the query if it can be reached by a chain of relevant category links from a store's home page, and then following one more link to the product offer:

$$\begin{aligned}
&Relevant(page, url, query) \Leftrightarrow \\
&\quad \exists\, store, home \;\; store \in OnlineStores \wedge Homepage(store, home) \\
&\quad \wedge \exists\, url_2 \;\; RelevantChain(home, url_2, query) \wedge Link(url_2, url) \\
&\qquad \wedge page = GetPage(url) .
\end{aligned}$$

Here the predicate $Link(from, to)$ means that there is a hyperlink from the *from* URL to the *to* URL. (See Exercise 10.13.) To define what counts as a *RelevantChain*, we need to follow not just any old hyperlinks, but only those links whose associated anchor text indicates that the link is relevant to the product query. For this, we will use $LinkText(from, to, text)$ to mean that there is a link between *from* and *to* with *text* as the anchor text. A chain of links between two URLs, *start* and *end*, is relevant to a description d if the anchor text of each link is a relevant category name for d. The existence of the chain itself is determined by a recursive definition, with the empty chain ($start = end$) as the base case:

$$\begin{aligned}
&RelevantChain(start, end, query) \Leftrightarrow (start = end) \\
&\quad \vee (\exists\, u, text \;\; LinkText(start, u, text) \wedge RelevantCategoryName(query, text) \\
&\qquad \wedge RelevantChain(u, end, query)) .
\end{aligned}$$

Now we must define what it means for *text* to be a *RelevantCategoryName* for *query*. First, we need to relate strings to the categories they name. This is done using the predicate $Name(s, c)$, which says that string s is a name for category c—for example, we might assert that $Name(\text{“}laptops\text{”}, LaptopComputers)$. Some more examples of the *Name* predicate

[7] An alternative to the link-following strategy is to use an Internet search engine; the technology behind Internet search, information retrieval, will be covered in Section 23.2.

$Books \subset Products$	$Name(\text{"}books\text{"}, Books)$
$MusicRecordings \subset Products$	$Name(\text{"}music\text{"}, MusicRecordings)$
$MusicCDs \subset MusicRecordings$	$Name(\text{"}CDs\text{"}, MusicCDs)$
$MusicTapes \subset MusicRecordings$	$Name(\text{"}tapes\text{"}, MusicTapes)$
$Electronics \subset Products$	$Name(\text{"}electronics\text{"}, Electronics)$
$DigitalCameras \subset Electronics$	$Name(\text{"}digital\ cameras\text{"}, DigitalCameras)$
$StereoEquipment \subset Electronics$	$Name(\text{"}stereos\text{"}, StereoEquipment)$
$Computers \subset Electronics$	$Name(\text{"}computers\text{"}, Computers)$
$LaptopComputers \subset Computers$	$Name(\text{"}laptops\text{"}, LaptopComputers)$
$DesktopComputers \subset Computers$	$Name(\text{"}desktops\text{"}, DesktopComputers)$
...	...
(a)	(b)

Figure 10.8 (a) Taxonomy of product categories. (b) Referring words for those categories.

appear in Figure 10.8(b). Next, we define relevance. Suppose that *query* is "laptops." Then $RelevantCategoryName(query, text)$ is true when one of the following holds:

- The *text* and *query* name the same category—e.g., "laptop computers" and "laptops."
- The *text* names a supercategory such as "computers."
- The *text* names a subcategory such as "ultralight notebooks."

The logical definition of $RelevantCategoryName$ is as follows:

$$RelevantCategoryName(query, text) \Leftrightarrow$$
$$\exists c_1, c_2 \;\; Name(query, c_1) \wedge Name(text, c_2) \wedge (c_1 \subseteq c_2 \vee c_2 \subseteq c_1) \; . \qquad (10.1)$$

Otherwise, the anchor text is irrelevant because it names a category outside this line, such as "mainframe computers" or "lawn & garden."

To follow relevant links, then, it is essential to have a rich hierarchy of product categories. The top part of this hierarchy might look like Figure 10.8(a). It will not be feasible to list *all* possible shopping categories, because a buyer could always come up with some new desire and manufacturers will always come out with new products to satisfy them (electric kneecap warmers?). Nonetheless, an ontology of about a thousand categories will serve as a very useful tool for most buyers.

In addition to the product hierarchy itself, we also need to have a rich vocabulary of names for categories. Life would be much easier if there were a one-to-one correspondence between categories and the character strings that name them. We have already seen the problem of **synonymy**—two names for the same category, such as "laptop computers" and "laptops." There is also the problem of **ambiguity**—one name for two or more different categories. For example, if we add the sentence

$$Name(\text{"}CDs\text{"}, CertificatesOfDeposit)$$

to the knowledge base in Figure 10.8(b), then "CDs" will name two different categories.

Synonymy and ambiguity can cause a significant increase in the number of paths that the agent has to follow, and can sometimes make it difficult to determine whether a given

page is indeed relevant. A much more serious problem is that there is a very broad range of descriptions that a user can type, or category names that a store can use. For example, the link might say "laptop" when the knowledge base has only "laptops;" or the user might ask for "a computer I can fit on the tray table of an economy-class seat in a Boeing 737." It is impossible to enumerate in advance all the ways a category can be named, so the agent will have to be able to do additional reasoning in some cases to determine if the *Name* relation holds. In the worst case, this requires full natural language understanding, a topic that we will defer to Chapter 22. In practice, a few simple rules—such as allowing "laptop" to match a category named "laptops"—go a long way. Exercise 10.15 asks you to develop a set of such rules after doing some research into online stores.

Given the logical definitions from the preceding paragraphs and suitable knowledge bases of product categories and naming conventions, are we ready to apply an inference algorithm to obtain a set of relevant offers for our query? Not quite! The missing element is the $GetPage(url)$ function, which refers to the HTML page at a given URL. The agent doesn't have the page contents of every URL in its knowledge base; nor does it have explicit rules for deducing what those contents might be. Instead, we can arrange for the right HTTP procedure to be executed whenever a subgoal involves the *GetPage* function. In this way, it appears to the inference engine as if the entire Web is inside the knowledge base. This is an example of a general technique called **procedural attachment**, whereby particular predicates and functions can be handled by special-purpose methods.

PROCEDURAL ATTACHMENT

Comparing offers

Let us assume that the reasoning processes of the preceding section have produced a set of offer pages for our "laptops" query. To compare those offers, the agent must extract the relevant information—price, speed, disk size, weight, and so on—from the offer pages. This can be a difficult task with real web pages, for all the reasons mentioned previously. A common way of dealing with this problem is to use programs called **wrappers** to extract information from a page. The technology of information extraction is discussed in Section 23.3. For now we assume that wrappers exist, and when given a page and a knowledge base, they add assertions to the knowledge base. Typically a hierarchy of wrappers would be applied to a page: a very general one to extract dates and prices, a more specific one to extract attributes for computer-related products, and if necessary a site-specific one that knows the format of a particular store. Given a page on the gen-store.com site with the text

WRAPPER

```
YVM ThinkBook 970.  Our price:  $1449.00
```

followed by various technical specifications, we would like a wrapper to extract information such as the following:

$$\begin{aligned}\exists\, &lc, offer \;\; lc \in LaptopComputers \wedge offer \in ProductOffers \wedge \\ &ScreenSize(lc, Inches(14)) \wedge ScreenType(lc, ColorLCD) \wedge \\ &MemorySize(lc, Megabytes(512)) \wedge CPUSpeed(lc, GHz(2.4)) \wedge \\ &OfferedProduct(offer, lc) \wedge Store(offer, GenStore) \wedge \\ &URL(offer, \text{“}\texttt{genstore.com/comps/34356.}html\text{”}) \wedge \\ &Price(offer, \$(449)) \wedge Date(offer, Today)\ .\end{aligned}$$

This example illustrates several issues that arise when we take seriously the task of knowledge engineering for commercial transactions. For example, notice that the price is an attribute of the *offer*, not the product itself. This is important because the offer at a given store may change from day to day even for the same individual laptop; for some categories—such as houses and paintings—the same individual object may even be offered simultaneously by different intermediaries at different prices. There are still more complications that we have not handled, such as the possibility that the price depends on method of payment and on the buyer's qualifications for certain discounts. All in all, there is much interesting work to do.

The final task is to compare the offers that have been extracted. For example, consider these three offers:

A : 2.4 GHz CPU, 512MB RAM, 80 GB disk, DVD, CDRW, \$1695 .
B : 2.0 GHz CPU, 1GB RAM, 120 GB disk, DVD, CDRW, \$1800 .
C : 2.2 GHz CPU, 512MB RAM, 80 GB disk, DVD, CDRW, \$1800 .

C is **dominated** by A; that is, A is cheaper and faster, and they are otherwise the same. In general, X dominates Y if X has a better value on at least one attribute, and is not worse on any attribute. But neither A nor B dominates the other. To decide which is better we need to know how the buyer weighs CPU speed and price against memory and disk space. The general topic of preferences among multiple attributes is addressed in Section 16.4; for now, our shopping agent will simply return a list of all undominated offers that meet the buyer's description. In this example, both A and B are undominated. Notice that this outcome relies on the assumption that everyone prefers cheaper prices, faster processors, and more storage. Some attributes, such as screen size on a notebook, depend on the user's particular preference (portability versus visibility); for these, the shopping agent will just have to ask the user.

The shopping agent we have described here is a simple one; many refinements are possible. Still, it has enough capability that with the right domain-specific knowledge it can actually be of use to a shopper. Because of its declarative construction, it extends easily to more complex applications. The main point of this section is to show that some knowledge representation—in particular, the product hierarchy—is necessary for an agent like this, and that once we have some knowledge in this form, it is not too hard to do the rest as a knowledge-based agent.

10.6 Reasoning Systems for Categories

We have seen that categories are the primary building blocks of any large-scale knowledge representation scheme. This section describes systems specially designed for organizing and reasoning with categories. There are two closely related families of systems: **semantic networks** provide graphical aids for visualizing a knowledge base and efficient algorithms for inferring properties of an object on the basis of its category membership; and **description logics** provide a formal language for constructing and combining category definitions and efficient algorithms for deciding subset and superset relationships between categories.

Semantic networks

EXISTENTIAL GRAPHS

In 1909, Charles Peirce proposed a graphical notation of nodes and arcs called **existential graphs** that he called "the logic of the future." Thus began a long-running debate between advocates of "logic" and advocates of "semantic networks." Unfortunately, the debate obscured the fact that semantics networks—at least those with well-defined semantics—*are* a form of logic. The notation that semantic networks provide for certain kinds of sentences is often more convenient, but if we strip away the "human interface" issues, the underlying concepts—objects, relations, quantification, and so on—are the same.

There are many variants of semantic networks, but all are capable of representing individual objects, categories of objects, and relations among objects. A typical graphical notation displays object or category names in ovals or boxes, and connects them with labeled arcs. For example, Figure 10.9 has a *MemberOf* link between *Mary* and *FemalePersons*, corresponding to the logical assertion $Mary \in FemalePersons$; similarly, the *SisterOf* link between *Mary* and *John* corresponds to the assertion $SisterOf(Mary, John)$. We can connect categories using *SubsetOf* links, and so on. It is such fun drawing bubbles and arrows that one can get carried away. For example, we know that persons have female persons as mothers, so can we draw a *HasMother* link from *Persons* to *FemalePersons*? The answer is no, because *HasMother* is a relation between a person and his or her mother, and categories do not have mothers.[8] For this reason, we have used a special notation—the double-boxed link—in Figure 10.9. This link asserts that

$$\forall x \;\; x \in Persons \;\Rightarrow\; [\forall y \;\; HasMother(x, y) \;\Rightarrow\; y \in FemalePersons] \; .$$

We might also want to assert that persons have two legs—that is,

$$\forall x \;\; x \in Persons \;\Rightarrow\; Legs(x, 2) \; .$$

As before, we need to be careful not to assert that a category has legs; the single-boxed link in Figure 10.9 is used to assert properties of every member of a category.

The semantic network notation makes it very convenient to perform **inheritance** reasoning of the kind introduced in Section 10.2. For example, by virtue of being a person, Mary inherits the property of having two legs. Thus, to find out how many legs Mary has, the inheritance algorithm follows the *MemberOf* link from *Mary* to the category she belongs to, and then follows *SubsetOf* links up the hierarchy until it finds a category for which there is a boxed *Legs* link—in this case, the *Persons* category. The simplicity and efficiency of this inference mechanism, compared with logical theorem proving, has been one of the main attractions of semantic networks.

MULTIPLE INHERITANCE

Inheritance becomes complicated when an object can belong to more than one category or when a category can be a subset of more than one other category; this is called **multiple inheritance**. In such cases, the inheritance algorithm might find two or more conflicting values

[8] Several early systems failed to distinguish between properties of members of a category and properties of the category as a whole. This can lead directly to inconsistencies, as pointed out by Drew McDermott (1976) in his article "Artificial Intelligence Meets Natural Stupidity." Another common problem was the use of *IsA* links for both subset and membership relations, in correspondence with English usage: "a cat is a mammal" and "Fifi is a cat." See Exercise 10.25 for more on these issues.

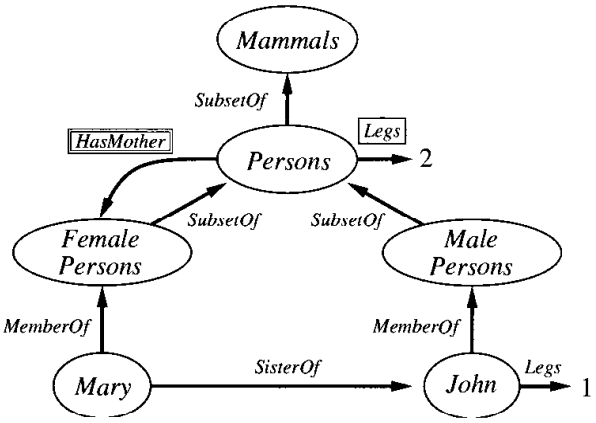


Figure 10.9 A semantic network with four objects (John, Mary, 1, and 2) and four categories. Relations are denoted by labeled links.

answering the query. For this reason, multiple inheritance is banned in some **object-oriented programming** (OOP) languages, such as Java, that use inheritance in a class hierarchy. It is usually allowed in semantic networks, but we defer discussion of that until Section 10.7.

INVERSE LINK

Another common form of inference is the use of **inverse links**. For example, *HasSister* is the inverse of *SisterOf*, which means that

$$\forall p, s \;\; HasSister(p, s) \Leftrightarrow SisterOf(s, p) \; .$$

This sentence can be asserted in a semantic network if links are **reified**—that is, made into objects in their own right. For example, we could have a *SisterOf* object, connected by an *Inverse* link to *HasSister*. Given a query asking who is a *SisterOf* John, the inference algorithm can discover that *HasSister* is the inverse of *SisterOf* and can therefore answer the query by following the *HasSister* link from *John* to *Mary*. Without the inverse information, it might be necessary to check every female person to see whether that person has a *SisterOf* link to John. This is because semantic networks provide direct indexing only for objects, categories, and the links emanating from them; in the vocabulary of first-order logic, it is as if the knowledge base were indexed only on the first argument of each predicate.

The reader might have noticed an obvious drawback of semantic network notation, compared to first-order logic: the fact that links between bubbles represent only *binary* relations. For example, the sentence $Fly(Shankar, NewYork, NewDelhi, Yesterday)$ cannot be asserted directly in a semantic network. Nonetheless, we *can* obtain the effect of n-ary assertions by reifying the proposition itself as an event (see Section 10.3) belonging to an appropriate event category. Figure 10.10 shows the semantic network structure for this particular event. Notice that the restriction to binary relations forces the creation of a rich ontology of reified concepts; indeed, much of the ontology developed in this chapter originated in semantic network systems.

Reification of propositions makes it possible to represent every ground, function-free atomic sentence of first-order logic in the semantic network notation. Certain kinds of universally quantified sentences can be asserted using inverse links and the singly boxed and doubly boxed arrows applied to categories, but that still leaves us a long way short of full first-order

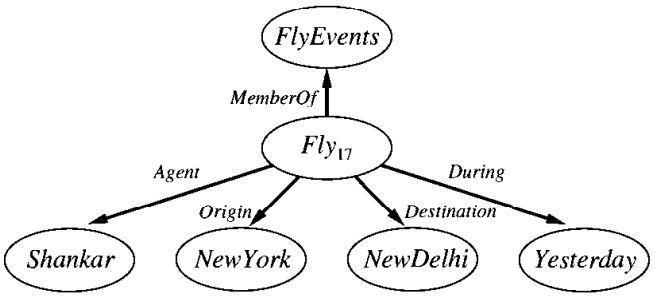


Figure 10.10 A fragment of a semantic network showing the representation of the logical assertion $Fly(Shankar, NewYork, NewDelhi, Yesterday)$.

logic. Negation, disjunction, nested function symbols, and existential quantification are all missing. Now it is *possible* to extend the notation to make it equivalent to first-order logic—as in Peirce's existential graphs or Hendrix's (1975) partitioned semantic networks—but doing so negates one of the main advantages of semantic networks, which is the simplicity and transparency of the inference processes. Designers can build a large network and still have a good idea about what queries will be efficient, because (a) it is easy to visualize the steps that the inference procedure will go through and (b) in some cases the query language is so simple that difficult queries cannot be posed. In cases where the expressive power proves to be too limiting, many semantic network systems provide for **procedural attachment** to fill in the gaps. Procedural attachment is a technique whereby a query about (or sometimes an assertion of) a certain relation results in a call to a special procedure designed for that relation rather than a general inference algorithm.

DEFAULT VALUES

One of the most important aspects of semantic networks is their ability to represent **default values** for categories. Examining Figure 10.9 carefully, one notices that John has one leg, despite the fact that he is a person and all persons have two legs. In a strictly logical KB, this would be a contradiction, but in a semantic network, the assertion that all persons have two legs has only default status; that is, a person is assumed to have two legs unless this is contradicted by more specific information. The default semantics is enforced naturally by the inheritance algorithm, because it follows links upwards from the object itself (John in this case) and stops as soon as it finds a value. We say that the default is **overridden** by the more specific value. Notice that we could also override the default number of legs by creating a category of $OneLeggedPersons$, a subset of $Persons$ of which $John$ is a member.

OVERRIDING

We can retain a strictly logical semantics for the network if we say that the $Legs$ assertion for $Persons$ includes an exception for John:

$$\forall x \;\; x \in Persons \wedge x \neq John \;\Rightarrow\; Legs(x, 2) \; .$$

For a *fixed* network, this is semantically adequate, but will be much less concise than the network notation itself if there are lots of exceptions. For a network that will be updated with more assertions, however, such an approach fails—we really want to say that any persons as yet unknown with one leg are exceptions too. Section 10.7 goes into more depth on this issue and on default reasoning in general.

Description logics

DESCRIPTION LOGICS

The syntax of first-order logic is designed to make it easy to say things about objects. **Description logics** are notations that are designed to make it easier to describe definitions and properties of categories. Description logic systems evolved from semantic networks in response to pressure to formalize what the networks mean while retaining the emphasis on taxonomic structure as an organizing principle.

SUBSUMPTION

CLASSIFICATION

The principal inference tasks for description logics are **subsumption**—checking if one category is a subset of another by comparing their definitions—and **classification**—checking whether an object belongs to a category. Some systems also include **consistency** of a category definition—whether the membership criteria are logically satisfiable.

The CLASSIC language (Borgida *et al.*, 1989) is a typical description logic. The syntax of CLASSIC descriptions is shown in Figure 10.11.[9] For example, to say that bachelors are unmarried adult males we would write

$$Bachelor = And(Unmarried, Adult, Male) .$$

The equivalent in first-order logic would be

$$Bachelor(x) \Leftrightarrow Unmarried(x) \wedge Adult(x) \wedge Male(x) .$$

Notice that the description logic effectively allows direct logical operations on predicates, rather than having to first create sentences to be joined by connectives. Any description in CLASSIC can be written in first-order logic, but some descriptions are more straightforward in CLASSIC. For example, to describe the set of men with at least three sons who are all unemployed and married to doctors and at most two daughters who are all professors in physics or math departments, we would use

$$\begin{aligned} &And(Man, AtLeast(3, Son), AtMost(2, Daughter),\\ &\quad All(Son, And(Unemployed, Married, All(Spouse, Doctor))),\\ &\quad All(Daughter, And(Professor, Fills(Department, Physics, Math)))) . \end{aligned}$$

We leave it as an exercise to translate this into first-order logic.

Perhaps the most important aspect of description logics is their emphasis on tractability of inference. A problem instance is solved by describing it and then asking if it is subsumed by one of several possible solution categories. In standard first-order logic systems, predicting the solution time is often impossible. It is frequently left to the user to engineer the representation to detour around sets of sentences that seem to be causing the system to take several weeks to solve a problem. The thrust in description logics, on the other hand, is to ensure that subsumption-testing can be solved in time polynomial in the size of the descriptions.[10]

This sounds wonderful in principle, until one realizes that it can only have one of two consequences: either hard problems cannot be stated at all, or they require exponentially large descriptions! However, the tractability results do shed light on what sorts of constructs cause problems and thus help the user to understand how different representations behave.

[9] Notice that the language does *not* allow one to simply state that one concept, or category, is a subset of another. This is a deliberate policy: subsumption between categories must be derivable from some aspects of the descriptions of the categories. If not, then something is missing from the descriptions.

[10] CLASSIC provides efficient subsumption testing in practice, but the worst-case runtime is exponential.

$$
\begin{aligned}
\mathit{Concept} &\rightarrow \textbf{Thing} \mid \mathit{ConceptName} \\
&\mid \textbf{And}(\mathit{Concept}, \ldots) \\
&\mid \textbf{All}(\mathit{RoleName}, \mathit{Concept}) \\
&\mid \textbf{AtLeast}(\mathit{Integer}, \mathit{RoleName}) \\
&\mid \textbf{AtMost}(\mathit{Integer}, \mathit{RoleName}) \\
&\mid \textbf{Fills}(\mathit{RoleName}, \mathit{IndividualName}, \ldots) \\
&\mid \textbf{SameAs}(\mathit{Path}, \mathit{Path}) \\
&\mid \textbf{OneOf}(\mathit{IndividualName}, \ldots) \\
\mathit{Path} &\rightarrow [\mathit{RoleName}, \ldots]
\end{aligned}
$$

Figure 10.11 The syntax of descriptions in a subset of the CLASSIC language.

For example, description logics usually lack *negation* and *disjunction*. Each forces first-order logical systems to go through a potentially exponential case analysis in order to ensure completeness. For the same reason, they are excluded from Prolog. CLASSIC allows only a limited form of disjunction in the *Fills* and *OneOf* constructs, which permit disjunction over explicitly enumerated individuals but not over descriptions. With disjunctive descriptions, nested definitions can lead easily to an exponential number of alternative routes by which one category can subsume another.

10.7 REASONING WITH DEFAULT INFORMATION

In the preceding section, we saw a simple example of an assertion with default status: people have two legs. This default can be overridden by more specific information, such as that Long John Silver has one leg. We saw that the inheritance mechanism in semantic networks implements the overriding of defaults in a simple and natural way. In this section, we study defaults more generally, with a view toward understanding the *semantics* of defaults rather than just providing a procedural mechanism.

Open and closed worlds

Suppose you were looking at a bulletin board in a university computer science department and saw a notice saying, "The following courses will be offered: CS 101, CS 102, CS 106, EE 101." Now, how many courses will be offered? If you answered "Four," you would be in agreement with a typical database system. Given a relational database with the equivalent of the four assertions

$$\mathit{Course}(\mathit{CS}, 101),\ \mathit{Course}(\mathit{CS}, 102),\ \mathit{Course}(\mathit{CS}, 106),\ \mathit{Course}(\mathit{EE}, 101), \quad (10.2)$$

the SQL query `count * from Course` returns 4. On the other hand, a first-order logical system would answer "Somewhere between one and infinity," not "four." The reason is that

the *Course* assertions do not deny the possibility that other unmentioned courses are also offered, nor do they say that the courses mentioned are different from each other.

This example shows that database systems and human communication conventions differ from first-order logic in at least two ways. First, databases (and people) assume that the information provided is *complete*, so that ground atomic sentences not asserted to be true are assumed to be false. This is called the **closed-world assumption**, or CWA. Second, we usually assume that distinct names refer to distinct objects. This is the **unique names assumption**, or UNA, which we introduced first in the context of action names in Section 10.3.

CLOSED-WORLD ASSUMPTION

First-order logic does not assume these conventions, and thus needs to be more explicit. To say that *only* the four distinct courses are offered, we would write:

$$Course(d,n) \Leftrightarrow [d,n]=[CS,101] \vee [d,n]=[CS,102] \vee [d,n]=[CS,106] \vee [d,n]=[EE,101] \ . \quad (10.3)$$

COMPLETION

Equation 10.3 is called the **completion**[11] of 10.2. In general, the completion will contain a definition—an if-and-only-if sentence—for each predicate, and each definition will contain a disjunct for each definite clause having that predicate in its head.[12] In general, the completion is constructed as follows:

1. Gather up all the clauses with the same predicate name (P) and the same arity (n).
2. Translate each clause to **Clark Normal Form**: replace

   CLARK NORMAL FORM

   $$P(t_1,\ldots,t_n) \leftarrow Body \ ,$$

   where t_i are terms, with

   $$P(v_1,\ldots,v_n) \leftarrow \exists w_1 \ldots w_m \ [v_1,\ldots,v_n]=[t_1,\ldots,t_n] \wedge Body \ ,$$

   where v_i are newly invented variables and w_i are the variables that appear in the original clause. Use the same set of v_i for every clause. This gives us a set of clauses

   $$P(v_1,\ldots,v_n) \leftarrow B_1$$
   $$\vdots$$
   $$P(v_1,\ldots,v_n) \leftarrow B_k \ .$$
3. Combine these together into one big disjunctive clause:

   $$P(v_1,\ldots,v_n) \leftarrow B_1 \vee \ldots \vee B_k \ .$$
4. Form the completion by replacing the $\leftarrow$ with an equivalence:

   $$P(v_1,\ldots,v_n) \Leftrightarrow B_1 \vee \ldots \vee B_k \ .$$

Figure 10.12 shows an example of the Clark completion for a knowledge base with both ground facts and rules. To add in the unique names assumption, we simply construct the Clark completion for the equality relation, where the only known facts are that $CS=CS$, $101=101$, and so on. This is left as an exercise.

The closed-world assumption allows us to find a **minimal model** of a relation. That is, we can find the model of the relation *Course* with the fewest elements. In Equation (10.2)

[11] Sometimes called "Clark Completion" after the inventor, Keith Clark.

[12] Notice that this is also the form of the successor-state axioms given in Section 10.3.

Horn Clauses	**Clark Completion**
$Course(CS, 101)$	$Course(d, n) \Leftrightarrow [d, n] = [CS, 101]$
$Course(CS, 102)$	$\vee\ [d, n] = [CS, 102]$
$Course(CS, 106)$	$\vee\ [d, n] = [CS, 106]$
$Course(EE, 101)$	$\vee\ [d, n] = [EE, 101]$
$Course(EE, i) \leftarrow Integer(i)$	$\vee\ \exists i\ [d, n] = [EE, i] \wedge Integer(i)$
$\wedge\ 101 \leq i \wedge i \leq 130$	$\wedge\ 101 \leq i \wedge i \leq 130$
$Course(CS, m + 100) \Leftarrow$	$\vee\ \exists m\ [d, n] = [CS, m + 100]$
$Course(CS, m) \wedge 100 \leq m$	$\wedge\ Course(CS, m) \wedge 100 \leq m$
$\wedge\ m < 200$	$\wedge\ m < 200$

Figure 10.12 The Clark Completion of a set of Horn clauses. The original Horn program (left) lists four courses explicitly and also asserts that there is a math class for every integer from 101 to 130, and that for every CS class in the 100 (undergraduate) series, there is a corresponding class in the 200 (graduate) series. The Clark completion (right) says that there are no other classes. With the completion and the unique names assumption (and the obvious definition of the *Integer* predicate), we get the desired conclusion that there are exactly 36 courses: 30 math courses and 6 CS courses.

the minimal model of *Course* has four elements; any less and we'd have a contradiction. For Horn knowledge bases, there is always a *unique* minimal model. Notice that, with the unique names assumption, this applies to the equality relation too: each term is equal only to itself. Paradoxically, this means that minimal models are maximal in the sense of having as many objects as possible.

It is possible to take a Horn program, generate the Clark completion, and hand that to a theorem prover to do inference. But it is usually more efficient to use a special-purpose inference mechanism such as Prolog, which has the closed world and unique names assumptions built into the inference mechanism.

Those who make the closed-world assumption must be careful about what kind of reasoning they will be doing. For example, in a census database it would be reasonable to make the CWA when reasoning about the current population of cities, but it would be wrong to conclude that no baby will ever be born in the future just because the database contains no entries with future birthdates. The CWA makes the database **complete**, in the sense that every atomic query is answered either positively or negatively; when we are genuinely ignorant of facts (such as future births) we cannot use the CWA. A more sophisticated knowledge representation system might allow the user to specify rules for when to apply the CWA.

Negation as failure and stable model semantics

We saw in Chapters 7 and 9 than Horn-form knowledge bases have desirable computational properties. In many applications, however, the requirement that every literal in the body of a clause be positive is rather inconvenient. We would like to say "You can go outside if it's not raining," without having to concoct predicates such as *NotRaining*. In this section, we explore the addition of a form of explicit negation to Horn clauses using the idea using of

NEGATION AS FAILURE

negation as failure. The idea is that a negative literal, *not* P, can be "proved" true just in case the proof of P fails. This is a form of default reasoning closely related to the closed world assumption: we assume something is false if it cannot be proved true. We use "*not*" to distinguish negation as failure from the logical "$\neg$" operator.

Prolog allows the *not* operator in the body of a clause. For example, consider the following Prolog program:

$$\begin{array}{l} IDEdrive \leftarrow Drive \wedge not\ SCSIdrive\ . \\ SCSIdrive \leftarrow Drive \wedge not\ IDEdrive\ . \\ SCSIcontroller \leftarrow SCSIdrive\ . \\ Drive\ . \end{array} \tag{10.4}$$

The first rule says that if we have a hard drive on a computer and it is not SCSI, then it must be IDE. The second says if it is not IDE it must be SCSI. The third says that having a SCSI drive implies having a SCSI controller, and the fourth says that we do indeed have a drive. This program has *two* minimal models:

$$\begin{array}{l} M_1 = \{Drive, IDEdrive\}\ , \\ M_2 = \{Drive, SCSIdrive, SCSIcontroller\}\ . \end{array}$$

Minimal models do not capture the intended semantics of programs with negation as failure. Consider the program

$$P \leftarrow not\ Q. \tag{10.5}$$

This has two minimal models, $\{P\}$ and $\{Q\}$. From an FOL point of view this makes sense, since $P \Leftarrow \neg Q$ is equivalent to $P \vee Q$. But from a Prolog point of view it is worrisome: Q never appears on the left hand side of an arrow, so how can it be a consequence?

STABLE MODEL
JUSTIFICATION

An alternative is the idea of a **stable model**, which is a minimal model where every atom in the model has a **justification**: a rule where the head is the atom and where every literal in the body is satisfied. Technically, we say that M is a stable model of a program H if M is the unique minimal model of the **reduct** of H with respect to M. The reduct of a program H is defined by first deleting from H any rule that has a literal *not* A in the body, where A is in the model, and then deleting any negative literals in the remaining rules. Since the reduct of H is now a list of Horn clauses, it must have a unique minimal model.

REDUCT

The reduct of $P \leftarrow not\ Q$ with respect to $\{P\}$ is P, which has minimal model $\{P\}$. Therefore $\{P\}$ is a stable model. The reduct with respect to $\{Q\}$ is the empty program, which has minimal model $\{\}$. Therefore $\{Q\}$ is not a stable model because Q has no justification in Equation (10.5). As another example, the reduct of 10.4 with respect to M_1 is as follows:

$$\begin{array}{l} IDEdrive \leftarrow Drive\ . \\ SCSIcontroller \leftarrow SCSIdrive\ . \\ Drive\ . \end{array}$$

ANSWER SET PROGRAMMING
ANSWER SETS

This has minimal model M_1, so M_1 is a stable model. **Answer set programming** is a kind of logic programming with negation as failure that works by translating the logic program into ground form and then searching for stable models (also known as **answer sets**) using propositional model checking techniques. Thus answer set programming is a descendant both of Prolog and of the fast propositional satisfiability provers such as WALKSAT. Indeed,

answer set programming has been successfully applied to problems in planning just as the propositional satisfiability provers have. The advantage of answer set planning over other planners is the degree of flexibility: the planning operators and constraints can be expressed as logic programs and are not bound to the restricted format of a particular planning formalism. The disadvantage of answer set planning is the same as for other propositional techniques: if there are very many objects in the universe, then there can be an exponential slow-down.

Circumscription and default logic

We have seen two examples where apparently natural reasoning processes violate the **monotonicity** property of logic that was proved in Chapter 7.[13] In the first example, a property inherited by all members of a category in a semantic network could be overridden by more specific information for a subcategory. In the second example, negated literals derived from a closed-world assumption could be overridden by the addition of positive literals.

Simple introspection suggests that these failures of monotonicity are widespread in commonsense reasoning. It seems that humans often "jump to conclusions." For example, when one sees a car parked on the street, one is normally willing to believe that it has four wheels even though only three are visible. (If you feel that the existence of the fourth wheel is dubious, consider also the question as to whether the three visible wheels are real or merely cardboard facsimiles.) Now, probability theory can certainly provide a conclusion that the fourth wheel exists with high probability, yet, for most people, the possibility of the car's not having four wheels *does not arise unless some new evidence presents itself.* Thus, it seems that the four-wheel conclusion is reached *by default*, in the absence of any reason to doubt it. If new evidence arrives—for example, if one sees the owner carrying a wheel and notices that the car is jacked up—then the conclusion can be retracted. This kind of reasoning is said to exhibit **nonmonotonicity**, because the set of beliefs does not grow monotonically over time as new evidence arrives. **Nonmonotonic logics** have been devised with modified notions of truth and entailment in order to capture such behavior. We will look at two such logics that have been studied extensively: circumscription and default logic.

NONMONOTONICITY

NONMONOTONIC LOGICS

CIRCUMSCRIPTION

Circumscription can be seen as a more powerful and precise version of the closed-world assumption. The idea is to specify particular predicates that are assumed to be "as false as possible"—that is, false for every object except those for which they are known to be true. For example, suppose we want to assert the default rule that birds fly. We would introduce a predicate, say $Abnormal_1(x)$, and write

$$Bird(x) \wedge \neg Abnormal_1(x) \Rightarrow Flies(x) .$$

If we say that $Abnormal_1$ is to be **circumscribed**, a circumscriptive reasoner is entitled to assume $\neg Abnormal_1(x)$ unless $Abnormal_1(x)$ is known to be true. This allows the conclusion $Flies(Tweety)$ to be drawn from the premise $Bird(Tweety)$, but the conclusion no longer holds if $Abnormal_1(Tweety)$ is asserted.

MODEL PREFERENCE

Circumscription can be viewed as an example of a **model preference** logic. In such logics, a sentence is entailed (with default status) if it is true in all *preferred* models of the KB,

[13] Recall that monotonicity requires all entailed sentences to remain entailed after new sentences are added to the KB. That is, if $KB \models \alpha$ then $KB \wedge \beta \models \alpha$.

as opposed to the requirement of truth in *all* models in classical logic. For circumscription, one model is preferred to another if it has fewer abnormal objects.[14] Let us see how this idea works in the context of multiple inheritance in semantic networks. The standard example for which multiple inheritance is problematic is called the "Nixon diamond." It arises from the observation that Richard Nixon was both a Quaker (and hence by default a pacifist) and a Republican (and hence by default not a pacifist). We can write this as follows:

$$Republican(Nixon) \wedge Quaker(Nixon) .$$
$$Republican(x) \wedge \neg Abnormal_2(x) \Rightarrow \neg Pacifist(x) .$$
$$Quaker(x) \wedge \neg Abnormal_3(x) \Rightarrow Pacifist(x) .$$

If we circumscribe $Abnormal_2$ and $Abnormal_3$, there are two preferred models: one in which $Abnormal_2(Nixon)$ and $Pacifist(Nixon)$ hold and one in which $Abnormal_3(Nixon)$ and $\neg Pacifist(Nixon)$ hold. Thus, the circumscriptive reasoner remains properly agnostic as to whether Nixon is a pacifist. If we wish, in addition, to assert that religious beliefs take precedence over political beliefs, we can use a formalism called **prioritized circumscription** to give preference to models where $Abnormal_3$ is minimized.

PRIORITIZED CIRCUMSCRIPTION

DEFAULT LOGIC

DEFAULT RULES

Default logic is a formalism in which **default rules** can be written to generate contingent, nonmonotonic conclusions. A default rule looks like this:

$$Bird(x) : Flies(x)/Flies(x) .$$

This rule means that if $Bird(x)$ is true, and if $Flies(x)$ is consistent with the knowledge base, then $Flies(x)$ may be concluded by default. In general, a default rule has the form

$$P : J_1, \ldots, J_n/C$$

where P is called the prerequisite, C is the conclusion, and J_i are the justifications—if any one of them can be proven false, then the conclusion cannot be drawn. Any variable that appears in J_i or C must also appear in P. The Nixon-diamond example can be represented in default logic with one fact and two default rules:

$$Republican(Nixon) \wedge Quaker(Nixon) .$$
$$Republican(x) : \neg Pacifist(x)/\neg Pacifist(x) .$$
$$Quaker(x) : Pacifist(x)/Pacifist(x) .$$

EXTENSION

To interpret what the default rules mean, we define the notion of an **extension** of a default theory to be a maximal set of consequences of the theory. That is, an extension S consists of the original known facts and a set of conclusions from the default rules, such that no additional conclusions can be drawn from S and the justifications of every default conclusion in S are consistent with S. As in the case of the preferred models in circumscription, we have two possible extensions for the Nixon diamond: one wherein he is a pacifist and one wherein he is not. Prioritized schemes exist in which some default rules can be given precedence over others, allowing some ambiguities to be resolved.

Since 1980, when nonmonotonic logics were first proposed, a great deal of progress has been made in understanding their mathematical properties. Beginning in the late 1990s,

[14] For the closed-world assumption, one model is preferred to another if it has fewer true atoms—that is, preferred models are **minimal** models. There is a natural connection between the CWA and definite clause KBs, because the fixed point reached by forward chaining on such KBs is the unique minimal model. (See page 219.)

practical systems based on logic programming have shown promise as knowledge representation tools. There are still unresolved questions, however. For example, if "Cars have four wheels" is false, what does it mean to have it in one's knowledge base? What is a good set of default rules to have? If we cannot decide, for each rule separately, whether it belongs in our knowledge base, then we have a serious problem of nonmodularity. Finally, how can beliefs that have default status be used to make decisions? This is probably the hardest issue for default reasoning. Decisions often involve tradeoffs, and one therefore needs to compare the *strengths* of belief in the outcomes of different actions. In cases where the same kinds of decisions are being made repeatedly, it is possible to interpret default rules as "threshold probability" statements. For example, the default rule "My brakes are always OK" really means "The probability that my brakes are OK, given no other information, is sufficiently high that the optimal decision is for me to drive without checking them." When the decision context changes—for example, when one is driving a heavily laden truck down a steep mountain road—the default rule suddenly becomes inappropriate, even though there is no new evidence to suggest that the brakes are faulty. These considerations have led some researchers to consider how to embed default reasoning in probability theory.

10.8 TRUTH MAINTENANCE SYSTEMS

The previous section argued that many of the inferences drawn by a knowledge representation system will have only default status, rather than being absolutely certain. Inevitably, some of these inferred facts will turn out to be wrong and will have to be retracted in the face of new information. This process is called **belief revision**.[15] Suppose that a knowledge base KB contains a sentence P—perhaps a default conclusion recorded by a forward-chaining algorithm, or perhaps just an incorrect assertion—and we want to execute TELL(KB, $\neg P$). To avoid creating a contradiction, we must first execute RETRACT(KB, P). This sounds easy enough. Problems arise, however, if any *additional* sentences were inferred from P and asserted in the KB. For example, the implication $P \Rightarrow Q$ might have been used to add Q. The obvious "solution"—retracting all sentences inferred from P—fails because such sentences may have other justifications besides P. For example, if R and $R \Rightarrow Q$ are also in the KB, then Q does not have to be removed after all. **Truth maintenance systems**, or TMSs, are designed to handle exactly these kinds of complications.

BELIEF REVISION

TRUTH MAINTENANCE SYSTEM

One very simple approach to truth maintenance is to keep track of the order in which sentences are told to the knowledge base by numbering them from P_1 to P_n. When the call RETRACT(KB, P_i) is made, the system reverts to the state just before P_i was added, thereby removing both P_i and any inferences that were derived from P_i. The sentences P_{i+1} through P_n can then be added again. This is simple, and it guarantees that the knowledge base will be consistent, but retracting P_i requires retracting and reasserting $n - i$ sentences as well as

[15] Belief revision is often contrasted with **belief update**, which occurs when a knowledge base is revised to reflect a change in the world rather than new information about a fixed world. Belief update combines belief revision with reasoning about time and change; it is also related to the process of **filtering** described in Chapter 15.

undoing and redoing all the inferences drawn from those sentences. For systems to which many facts are being added—such as large commercial databases—this is impractical.

JTMS

A more efficient approach is the justification-based truth maintenance system, or **JTMS**.

JUSTIFICATION

In a JTMS, each sentence in the knowledge base is annotated with a **justification** consisting of the set of sentences from which it was inferred. For example, if the knowledge base already contains $P \Rightarrow Q$, then TELL(P) will cause Q to be added with the justification $\{P,\ P \Rightarrow Q\}$. In general, a sentence can have any number of justifications. Justifications are used to make retraction efficient. Given the call RETRACT(P), the JTMS will delete exactly those sentences for which P is a member of every justification. So, if a sentence Q had the single justification $\{P,\ P \Rightarrow Q\}$ it would be removed, if it had the additional justification $\{P,\ P \vee R \Rightarrow Q\}$ it would still be removed, but if it also had the justification $\{R,\ P \vee R \Rightarrow Q\}$, then it would be spared. In this way, the time required for retraction of P depends only on the number of sentences derived from P rather than on the number of other sentences added since P entered the knowledge base.

The JTMS assumes that sentences that are considered once will probably be considered again, so rather than deleting a sentence from the knowledge base entirely when it loses all justifications, we merely mark the sentence as being *out* of the knowledge base. If a subsequent assertion restores one of the justifications, then we mark the sentence as being back *in*. In this way, the JTMS retains all of the inference chains that it uses and need not rederive sentences when a justification becomes valid again.

In addition to handling the retraction of incorrect information, TMSs can be used to speed up the analysis of multiple hypothetical situations. Suppose, for example, that the Romanian Olympic Committee is choosing sites for the swimming, athletics, and equestrian events at the 2048 Games to be held in Romania. For example, let the first hypothesis be $Site(Swimming, Pitesti)$, $Site(Athletics, Bucharest)$, and $Site(Equestrian, Arad)$. A great deal of reasoning must then be done to work out the logistical consequences and hence the desirability of this selection. If we want to consider $Site(Athletics, Sibiu)$ instead, the TMS avoids the need to start again from scratch. Instead, we simply retract $Site(Athletics, Bucharest)$ and assert $Site(Athletics, Sibiu)$ and the TMS takes care of the necessary revisions. Inference chains generated from the choice of Bucharest can be reused with Sibiu, provided that the conclusions are the same.

ATMS

An assumption-based truth maintenance system, or **ATMS**, is designed to make this type of context-switching between hypothetical worlds particularly efficient. In a JTMS, the maintenance of justifications allows you to move quickly from one state to another by making a few retractions and assertions, but at any time only one state is represented. An ATMS represents *all* the states that have ever been considered at the same time. Whereas a JTMS simply labels each sentence as being *in* or *out*, an ATMS keeps track, for each sentence, of which assumptions would cause the sentence to be true. In other words, each sentence has a label that consists of a set of assumption sets. The sentence holds just in those cases where all the assumptions in one of the assumption sets hold.

EXPLANATIONS

Truth maintenance systems also provide a mechanism for generating **explanations**. Technically, an explanation of a sentence P is a set of sentences E such that E entails P. If the sentences in E are already known to be true, then E simply provides a sufficient ba-

ASSUMPTIONS

sis for proving that P must be the case. But explanations can also include **assumptions**—sentences that are not known to be true, but would suffice to prove P if they were true. For example, one might not have enough information to prove that one's car won't start, but a reasonable explanation might include the assumption that the battery is dead. This, combined with knowledge of how cars operate, explains the observed nonbehavior. In most cases, we will prefer an explanation E that is minimal, meaning that there is no proper subset of E that is also an explanation. An ATMS can generate explanations for the "car won't start" problem by making assumptions (such as "gas in car" or "battery dead") in any order we like, even if some assumptions are contradictory. Then we look at the label for the sentence "car won't start" to read off the sets of assumptions that would justify the sentence.

The exact algorithms used to implement truth maintenance systems are a little complicated, and we do not cover them here. The computational complexity of the truth maintenance problem is at least as great as that of propositional inference—that is, NP-hard. Therefore, you should not expect truth maintenance to be a panacea. When used carefully, however, a TMS can provide a substantial increase in the ability of a logical system to handle complex environments and hypotheses.

10.9 SUMMARY

This has been the most detailed chapter of the book so far. By delving into the details of how one represents a variety of knowledge, we hope we have given the reader a sense of how real knowledge bases are constructed. The major points are as follows:

- Large-scale knowledge representation requires a general-purpose ontology to organize and tie together the various specific domains of knowledge.
- A general-purpose ontology needs to cover a wide variety of knowledge and should be capable, in principle, of handling any domain.
- We presented an **upper ontology** based on categories and the event calculus. We covered structured objects, time and space, change, processes, substances, and beliefs.
- Actions, events, and time can be represented either in situation calculus or in more expressive representations such as event calculus and fluent calculus. Such representations enable an agent to construct plans by logical inference.
- The mental states of agents can be represented by strings that denote beliefs.
- We presented a detailed analysis of the Internet shopping domain, exercising the general ontology and showing how the domain knowledge can be used by a shopping agent.
- Special-purpose representation systems, such as **semantic networks** and **description logics**, have been devised to help in organizing a hierarchy of categories. **Inheritance** is an important form of inference, allowing the properties of objects to be deduced from their membership in categories.
- The **closed-world assumption**, as implemented in logic programs, provides a simple way to avoid having to specify lots of negative information. It is best interpreted as a **default** that can be overridden by additional information.

- **Nonmonotonic logics**, such as **circumscription** and **default logic**, are intended to capture default reasoning in general. **Answer set programming** speeds up nonmonotonic inference, much as WALKSAT speeds up propositional inference.
- **Truth maintenance systems** handle knowledge updates and revisions efficiently.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

There are plausible claims (Briggs, 1985) that formal knowledge representation research began with classical Indian theorizing about the grammar of Shastric Sanskrit, which dates back to the first millennium B.C. In the West, the use of definitions of terms in ancient Greek mathematics can be regarded as the earliest instance. Indeed, the development of technical terminology in any field can be regarded as a form of knowledge representation.

Early discussions of representation in AI tended to focus on "*problem* representation" rather than "*knowledge* representation." (See, for example, Amarel's (1968) discussion of the Missionaries and Cannibals problem.) In the 1970s, AI emphasized the development of "expert systems" (also called "knowledge-based systems") that could, if given the appropriate domain knowledge, match or exceed the performance of human experts on narrowly defined tasks. For example, the first expert system, DENDRAL (Feigenbaum *et al.*, 1971; Lindsay *et al.*, 1980), interpreted the output of a mass spectrometer (a type of instrument used to analyze the structure of organic chemical compounds) as accurately as expert chemists. Although the success of DENDRAL was instrumental in convincing the AI research community of the importance of knowledge representation, the representational formalisms used in DENDRAL are highly specific to the domain of chemistry. Over time, researchers became interested in standardized knowledge representation formalisms and ontologies that could streamline the process of creating new expert systems. In so doing, they ventured into territory previously explored by philosophers of science and of language. The discipline imposed in AI by the need for one's theories to "work" has led to more rapid and deeper progress than was the case when these problems were the exclusive domain of philosophy (although it has at times also led to the repeated reinvention of the wheel).

The creation of comprehensive taxonomies or classifications dates back to ancient times. Aristotle (384–322 B.C.) strongly emphasized classification and categorization schemes. His *Organon*, a collection of works on logic assembled by his students after his death, included a treatise called *Categories* in which he attempted to construct what we would now call an upper ontology. He also introduced the notions of **genus** and **species** for lower-level classification, although not with their modern, specifically biological meaning. Our present system of biological classification, including the use of "binomial nomenclature" (classification via genus and species in the technical sense), was invented by the Swedish biologist Carolus Linnaeus, or Carl von Linne (1707–1778). The problems associated with natural kinds and inexact category boundaries have been addressed by Wittgenstein (1953), Quine (1953), Lakoff (1987), and Schwartz (1977), among others.

Interest in larger-scale ontologies is increasing. The CYC project (Lenat, 1995; Lenat and Guha, 1990) has released a 6,000-concept upper ontology with 60,000 facts, and licenses

a much larger global ontology. The IEEE has established subcommittee P1600.1, the Standard Upper Ontology Working Group, and the Open Mind Initiative has enlisted over 7,000 Internet users to enter more than 400,000 facts about commonsense concepts. On the Web, standards such as RDF, XML, and the Semantic Web (Berners-Lee *et al.*, 2001) are emerging, but are not yet widely used. The conferences on *Formal Ontology in Information Systems* (FOIS) contain many interesting papers on both general and domain-specific ontologies.

The taxonomy used in this chapter was developed by the authors and is based in part on their experience in the CYC project and in part on work by Hwang and Schubert (1993) and Davis (1990). An inspirational discussion of the general project of commonsense knowledge representation appears in Hayes's (1978, 1985b) "The Naive Physics Manifesto."

The representation of time, change, actions, and events has been studied extensively in philosophy and theoretical computer science as well as in AI. The oldest approach is **temporal logic**, which is a specialized logic in which each model describes a complete trajectory through time (usually either linear or branching), rather than just a static relational structure. The logic includes **modal operators** that are applied to formulas; $\Box p$ means "p will be true at all times in the future," and $\Diamond p$ means "p will be true at some time in the future." The study of temporal logic was initiated by Aristotle and the Megarian and Stoic schools in ancient Greece. In modern times, Findlay (1941) was the first to suggest a formal calculus for reasoning about time, but the work of Arthur Prior (1967) is considered the most influential. Textbooks on temporal logic include those by Rescher and Urquhart (1971) and van Benthem (1983).

Theoretical computer scientists have long been interested in formalizing the properties of programs, viewed as sequences of computational actions. Burstall (1974) introduced the idea of using modal operators to reason about computer programs. Soon thereafter, Vaughan Pratt (1976) designed **dynamic logic**, in which modal operators indicate the effects of programs or other actions (see also Harel, 1984). For instance, in dynamic logic, if α is the name of a program, then "$[\alpha]p$" means "p would be true in all world states resulting from executing program α in the current world state", and "$\langle\alpha\rangle p$" means "p would be true in at least one world state resulting from executing program α in the current world state." Dynamic logic was applied to the actual analysis of programs by Fischer and Ladner (1977). Pnueli (1977) introduced the idea of using classical temporal logic to reason about programs.

Whereas temporal logic puts time directly into the model theory of the language, representations of time in AI have tended to incorporate axioms about times and events explicitly in the knowledge base, giving time no special status in the logic. This approach can allow for greater clarity and flexibility in some cases. Also, temporal knowledge expressed in first-order logic can be more easily integrated with other knowledge that has been accumulated in that notation.

The earliest treatment of time and action in AI was John McCarthy's (1963) situation calculus. The first AI system to make substantial use of general-purpose reasoning about actions in first-order logic was QA3 (Green, 1969b). Kowalski (1979b) developed the idea of reifying propositions within situation calculus.

The **frame problem** was first recognized by McCarthy and Hayes (1969). Many researchers considered the problem insoluble within first-order logic, and it spurred a great

deal of research into nonmonotonic logics. Philosophers from Dreyfus (1972) to Crockett (1994) have cited the frame problem as one symptom of the inevitable failure of the entire AI enterprise. The partial solution of the representational frame problem using successor-state axioms is due to Ray Reiter (1991); a solution of the inferential frame problem can be traced to work by Holldobler and Schneeberger (1990) on what became known as fluent calculus (Thielscher, 1999). The discussion in this chapter is based partly on the analyses by Lin and Reiter (1997) and Thielscher (1999). Books by Shanahan (1997) and Reiter (2001b) give complete, modern treatments of reasoning about action in situation calculus.

The partial resolution of the frame problem has rekindled interest in the declarative approach to reasoning about actions, which had been eclipsed by special-purpose planning systems since the early 1970s. (See Chapter 11.) Under the banner of **cognitive robotics**, much progress has been made on logical representations of action and time. The GOLOG language uses the full expressive power of logic programming to describe actions and plans (Levesque *et al.*, 1997a) and has been extended to handle concurrent actions (Giacomo *et al.*, 2000), stochastic environments (Boutilier *et al.*, 2000), and sensing (Reiter, 2001a).

COGNITIVE ROBOTICS

The event calculus was introduced by Kowalski and Sergot (1986) to handle continuous time, and there have been several variations (Sadri and Kowalski, 1995). Shanahan (1999) presents a good short overview. James Allen introduced time intervals for the same reason (Allen, 1983, 1984), arguing that intervals were much more natural than situations for reasoning about extended and concurrent events. Peter Ladkin (1986a, 1986b) introduced "concave" time intervals (intervals with gaps; essentially, unions of ordinary "convex" time intervals) and applied the techniques of mathematical abstract algebra to time representation. Allen (1991) systematically investigates the wide variety of techniques available for time representation. Shoham (1987) describes the reification of events and sets forth a novel scheme of his own for the purpose. There are significant commonalities between the event-based ontology given in this chapter and an analysis of events due to the philosopher Donald Davidson (1980). The **histories** in Pat Hayes's (1985a) ontology of liquids also have much the same flavor.

The question of the ontological status of substances has a long history. Plato proposed that substances were abstract entities entirely distinct from physical objects; he would say $MadeOf(Butter_3, Butter)$ rather than $Butter_3 \in Butter$. This leads to a substance hierarchy in which, for example, $UnsaltedButter$ is a more specific substance than $Butter$. The position adopted in this chapter, in which substances are categories of objects, was championed by Richard Montague (1973). It has also been adopted in the CYC project. Copeland (1993) mounts a serious, but not invincible, attack. The alternative approach mentioned in the chapter, in which butter is one object consisting of all buttery objects in the universe, was proposed originally by the Polish logician Leśniewski (1916). His **mereology** (the name is derived from the Greek word for "part") used the part–whole relation as a substitute for mathematical set theory, with the aim of eliminating abstract entities such as sets. A more readable exposition of these ideas is given by Leonard and Goodman (1940), and Goodman's *The Structure of Appearance* (1977) applies the ideas to various problems in knowledge representation. While some aspects of the mereological approach are awkward—for example, the need for a separate inheritance mechanism based on part–whole relations—the approach gained the

MEREOLOGY

support of Quine (1960). Harry Bunt (1985) has provided an extensive analysis of its use in knowledge representation.

MODAL LOGIC

Mental objects and states have been the subject of intensive study in philosophy and AI. **Modal logic** is the classical method for reasoning about knowledge in philosophy. Modal logic augments first-order logic with modal operators, such as B (believes) and K (knows), that take *sentences* rather than terms as arguments. The proof theory for modal logic restricts substitution within modal contexts, thereby achieving referential opacity. The modal logic of knowledge was invented by Jaakko Hintikka (1962). Saul Kripke (1963) defined the semantics of the modal logic of knowledge in terms of **possible worlds**. Roughly speaking, a world is possible for an agent if it is consistent with everything the agent knows. From this, one can derive rules of inference involving the K operator. Robert C. Moore relates the modal logic of knowledge to a style of reasoning about knowledge that refers directly to possible worlds in first-order logic (Moore, 1980, 1985). Modal logic can be an intimidatingly arcane field, but it has found significant applications in reasoning about information in distributed computer systems. The book *Reasoning about Knowledge* by Fagin *et al.* (1995) provides a thorough introduction to the modal approach. The biennial conference on *Theoretical Aspects of Reasoning About Knowledge* (TARK) covers applications of the theory of knowledge in AI, economics, and distributed systems.

The syntactic theory of mental objects was first studied in depth by Kaplan and Montague (1960), who showed that it led to paradoxes if not handled carefully. Because it has a natural model in terms of beliefs as physical configurations of a computer or a brain, it has been popular in AI in recent years. Konolige (1982) and Haas (1986) used it to describe inference engines of limited power, and Morgenstern (1987) showed how it could be used to describe knowledge preconditions in planning. The methods for planning observation actions in Chapter 12 are based on the syntactic theory. Ernie Davis (1990) gives an excellent comparison of the syntactic and modal theories of knowledge.

The Greek philosopher Porphyry (c. 234–305 A.D.), commenting on Aristotle's *Categories*, drew what might qualify as the first semantic network. Charles S. Peirce (1909) developed existential graphs as the first semantic network formalism using modern logic. Ross Quillian (1961), driven by an interest in human memory and language processing, initiated work on semantic networks within AI. An influential paper by Marvin Minsky (1975) presented a version of semantic networks called **frames**; a frame was a representation of an object or category, with attributes and relations to other objects or categories. Although the paper served to initiate interest in the field of knowledge representation *per se*, it was criticized as a recycling of earlier ideas developed in object-oriented programming, such as inheritance and the use of default values (Dahl *et al.*, 1970; Birtwistle *et al.*, 1973). It is not clear to what extent the latter papers on object-oriented programming were influenced in turn by early AI work on semantic networks.

The question of semantics arose quite acutely with respect to Quillian's semantic networks (and those of others who followed his approach), with their ubiquitous and very vague "IS-A links," as well as other early knowledge representation formalisms such as that of MERLIN (Moore and Newell, 1973) with its mysterious "flat" and "cover" operations. Woods' (1975) famous article "What's In a Link?" drew the attention of AI researchers to the

need for precise semantics in knowledge representation formalisms. Brachman (1979) elaborated on this point and proposed solutions. Patrick Hayes's (1979) "The Logic of Frames" cut even deeper, claiming that "Most of 'frames' is just a new syntax for parts of first-order logic." Drew McDermott's (1978b) "Tarskian Semantics, or, No Notation without Denotation!" argued that the model-theoretic approach to semantics used in first-order logic should be applied to all knowledge representation formalisms. This remains a controversial idea; notably, McDermott himself has reversed his position in "A Critique of Pure Reason" (McDermott, 1987). NETL (Fahlman, 1979) was a sophisticated semantic network system whose IS-A links (called "virtual copy," or VC, links) were based more on the notion of "inheritance" characteristic of frame systems or of object-oriented programming languages than on the subset relation and were much more precisely defined than Quillian's links from the pre-Woods era. NETL is particularly intriguing because it was intended to be implemented in parallel hardware to overcome the difficulty of retrieving information from large semantic networks. David Touretzky (1986) subjects inheritance to rigorous mathematical analysis. Selman and Levesque (1993) discuss the complexity of inheritance with exceptions, showing that in most formulations it is NP-complete.

The development of description logics is the most recent stage in a long line of research aimed at finding useful subsets of first-order logic for which inference is computationally tractable. Hector Levesque and Ron Brachman (1987) showed that certain logical constructs—notably, certain uses of disjunction and negation—were primarily responsible for the intractability of logical inference. Building on the KL-ONE system (Schmolze and Lipkis, 1983), a number of systems have been developed whose designs incorporate the results of theoretical complexity analysis, most notably KRYPTON (Brachman *et al.*, 1983) and Classic (Borgida *et al.*, 1989). The result has been a marked increase in the speed of inference and a much better understanding of the interaction between complexity and expressiveness in reasoning systems. Calvanese *et al.* (1999) summarize the state of the art. Against this trend, Doyle and Patil (1991) have argued that restricting the expressiveness of a language either makes it impossible to solve certain problems or encourages the user to circumvent the language restrictions through nonlogical means.

The three main formalisms for dealing with nonmonotonic inference—circumscription (McCarthy, 1980), default logic (Reiter, 1980), and modal nonmonotonic logic (McDermott and Doyle, 1980)—were all introduced in one special issue of the AI Journal. Answer set programming can be seen as an extension of negation as failure or as a refinement of circumscription; the underlying theory of stable model semantics was introduced by Gelfond and Lifschitz (1988) and the leading answer set programming systems are DLV (Eiter *et al.*, 1998) and SMODELS (Niemelä *et al.*, 2000). The disk drive example comes from the SMODELS user manual (Syrjänen, 2000). Lifschitz (2001) discusses the use of answer set programming for planning. Brewka *et al.* (1997) give a good overview of the various approaches to nonmonotonic logic. Clark (1978) covers the negation-as-failure approach to logic programming and Clark completion. Van Emden and Kowalski (1976) show that every Prolog program without negation has a unique minimal model. Recent years have seen renewed interest in applications of nonmonotonic logics to large-scale knowledge representation systems. The BENINQ systems for handling insurance benefits inquiries was perhaps the first commercially success-

ful application of a nonmonotonic inheritance system (Morgenstern, 1998). Lifschitz (2001) discusses the application of answer set programming to planning. A variety of nonmonotonic reasoning systems based on logic programming are documented in the proceedings of the conferences on *Logic Programming and Nonmonotonic Reasoning* (LPNMR).

The study of truth maintenance systems began with the TMS (Doyle, 1979) and RUP (McAllester, 1980) systems, both of which were essentially JTMSs. The ATMS approach was described in a series of papers by Johan de Kleer (1986a, 1986b, 1986c). *Building Problem Solvers* (Forbus and de Kleer, 1993) explains in depth how TMSs can be used in AI applications. Nayak and Williams (1997) show how an efficient TMS makes it feasible to plan the operations of a NASA spacecraft in real time.

For obvious reasons, this chapter does not cover *every* area of knowledge representation in depth. The three principal topics omitted are the following:

QUALITATIVE PHYSICS

◇ **Qualitative physics**: Qualitative physics is a subfield of knowledge representation concerned specifically with constructing a logical, nonnumeric theory of physical objects and processes. The term was coined by Johan de Kleer (1975), although the enterprise could be said to have started in Fahlman's (1974) BUILD, a sophisticated planner for constructing complex towers of blocks. Fahlman discovered in the process of designing it that most of the effort (80%, by his estimate) went into modeling the physics of the blocks world to calculate the stability of various subassemblies of blocks, rather than into planning per se. He sketches a hypothetical naive-physics-like process to explain why young children can solve BUILD-like problems without access to the high-speed floating-point arithmetic used in BUILD's physical modeling. Hayes (1985a) uses "histories"—four-dimensional slices of space-time similar to Davidson's events—to construct a fairly complex naive physics of liquids. Hayes was the first to prove that a bath with the plug in will eventually overflow if the tap keeps running and that a person who falls into a lake will get wet all over. De Kleer and Brown (1985) and Ken Forbus (1985) attempted to construct something like a general-purpose theory of the physical world, based on qualitative abstractions of physical equations. In recent years, qualitative physics has developed to the point where it is possible to analyze an impressive variety of complex physical systems (Sacks and Joskowicz, 1993; Yip, 1991). Qualitative techniques have been used to construct novel designs for clocks, windscreen wipers, and six-legged walkers (Subramanian, 1993; Subramanian and Wang, 1994). The collection *Readings in Qualitative Reasoning about Physical Systems* (Weld and de Kleer, 1990) provides a good introduction to the field.

SPATIAL REASONING

◇ **Spatial reasoning**: The reasoning necessary to navigate in the wumpus world and shopping world is trivial in comparison to the rich spatial structure of the real world. The earliest serious attempt to capture commonsense reasoning about space appears in the work of Ernest Davis (1986, 1990). The region connection calculus of Cohn *et al.* (1997) supports a form of qualitative spatial reasoning and has led to new kinds of geographical information system. As with qualitative physics, an agent can go a long way, so to speak, without resorting to a full metric representation. When such a representation is necessary, techniques developed in robotics (Chapter 25) can be used.

PSYCHOLOGICAL REASONING

◇ **Psychological reasoning**: Psychological reasoning involves the development of a working *psychology* for artificial agents to use in reasoning about themselves and other agents. This is often based on so-called folk psychology, the theory that humans in general are believed to use in reasoning about themselves and other humans. When AI researchers provide their artificial agents with psychological theories for reasoning about other agents, the theories are frequently based on the researchers' description of the logical agents' own design. Psychological reasoning is currently most useful within the context of natural language understanding, where divining the speaker's intentions is of paramount importance.

The proceedings of the international conferences on *Principles of Knowledge Representation and Reasoning* provide the most up-to-date sources for work in this area. *Readings in Knowledge Representation* (Brachman and Levesque, 1985) and *Formal Theories of the Commonsense World* (Hobbs and Moore, 1985) are excellent anthologies on knowledge representation; the former focuses more on historically important papers in representation languages and formalisms, the latter on the accumulation of the knowledge itself. Davis (1990), Stefik (1995), and Sowa (1999) provide textbook introductions to knowledge representation.

EXERCISES

10.1 Write sentences to define the effects of the *Shoot* action in the wumpus world. Describe its effects on the wumpus and remember that shooting uses the agent's arrow.

10.2 Within situation calculus, write an axiom to associate time 0 with the situation S_0 and another axiom to associate the time t with any situation that is derived from S_0 by a sequence of t actions.

10.3 In this exercise, we will consider the problem of planning a route for a robot to take from one city to another. The basic action taken by the robot is $Go(x, y)$, which takes it from city x to city y if there is a direct route between those cities. $DirectRoute(x, y)$ is true if and only if there is a direct route from x to y; you can assume that all such facts are already in the KB. (See the map on page 63.) The robot begins in Arad and must reach Bucharest.

a. Write a suitable logical description of the initial situation of the robot.

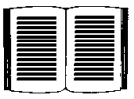
b. Write a suitable logical query whose solutions will provide possible paths to the goal.

c. Write a sentence describing the Go action.

d. Now suppose that following the direct route between two cities consumes an amount of fuel equal to the distance between the cities. The robot starts with fuel at full capacity. Augment your representation to include these considerations. Your action description should be such that the query you specified earlier will still result in feasible plans.

e. Describe the initial situation, and write a new rule or rules describing the Go action.

f. Now suppose some of the vertices are also gas stations, at which the robot can fill its tank. Extend your representation and write all the rules needed to describe gas stations, including the *Fillup* action.



10.4 Investigate ways to extend the event calculus to handle *simultaneous* events. Is it possible to avoid a combinatorial explosion of axioms?

10.5 Represent the following seven sentences using and extending the representations developed in the chapter:

a. Water is a liquid between 0 and 100 degrees.

b. Water boils at 100 degrees.

c. The water in John's water bottle is frozen.

d. Perrier is a kind of water.

e. John has Perrier in his water bottle.

f. All liquids have a freezing point.

g. A liter of water weighs more than a liter of alcohol.

Now repeat the exercise using a representation based on the mereological approach, in which, for example, $Water$ is an object containing as parts all the water in the world.

10.6 Write definitions for the following:

a. $ExhaustivePartDecomposition$

b. $PartPartition$

c. $PartwiseDisjoint$

These should be analogous to the definitions for $ExhaustiveDecomposition$, $Partition$, and $Disjoint$. Is it the case that $PartPartition(s, BunchOf(s))$? If so, prove it; if not, give a counterexample and define sufficient conditions under which it does hold.

10.7 Write a set of sentences that allows one to calculate the price of an individual tomato (or other object), given the price per pound. Extend the theory to allow the price of a bag of tomatoes to be calculated.

10.8 An alternative scheme for representing measures involves applying the units function to an abstract length object. In such a scheme, one would write $Inches(Length(L_1)) = 1.5$. How does this scheme compare with the one in the chapter? Issues include conversion axioms, names for abstract quantities (such as "50 dollars"), and comparisons of abstract measures in different units (50 inches is more than 50 centimeters).

10.9 Construct a representation for exchange rates between currencies that allows fluctuations on a daily basis.

10.10 This exercise concerns the relationships between event categories and the time intervals in which they occur.

a. Define the predicate $T(c, i)$ in terms of $During$ and $\in$.

b. Explain precisely why we do not need two different notations to describe conjunctive event categories.

c. Give a formal definition for $T(OneOf(p, q), i)$ and $T(Either(p, q), i)$.

d. Explain why it makes sense to have two forms of negation of events, analogous to the two forms of disjunction. Call them Not and $Never$ and give them formal definitions.

10.11 Define the predicate *Fixed*, where $Fixed(Location(x))$ means that the location of object x is fixed over time.

10.12 Define the predicates *Before*, *After*, *During*, and *Overlap*, using the predicate *Meet* and the functions *Start* and *End*, but not the function *Time* or the predicate $<$.

10.13 Section 10.5 used the predicates *Link* and *LinkText* to describe connections between web pages. Using the *InTag* and *GetPage* predicates, among others, write definitions for *Link* and *LinkText*.

10.14 One part of the shopping process that was not covered in this chapter is checking for compatibility between items. For example, if a customer orders a computer, is it matched with the right peripherals? If a digital camera is ordered, does it have the right memory card and batteries? Write a knowledge base that will decide whether a set of items is compatible and that can be used to suggest replacements or additional items if they are not compatible. Make sure that the knowledge base works with at least one line of products, and is easily extensible to other lines.

10.15 Add rules to extend the definition of the predicate $Name(s, c)$ so that a string such as "laptop computer" matches against the appropriate category names from a variety of stores. Try to make your definition general. Test it by looking at ten online stores, and at the category names they give for three different categories. For example, for the category of laptops, we found the names "Notebooks," "Laptops," "Notebook Computers," "Notebook," "Laptops and Notebooks," and "Notebook PCs." Some of these can be covered by explicit *Name* facts, while others could be covered by rules for handling plurals, conjunctions, etc.

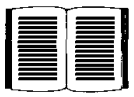
10.16 A complete solution to the problem of inexact matches to the buyer's description in shopping is very difficult and requires a full array of natural language processing and information retrieval techniques. (See Chapters 22 and 23.) One small step is to allow the user to specify minimum and maximum values for various attributes. We will insist that the buyer use the following grammar for product descriptions:

$$
\begin{array}{lcl}
\textit{Description} & \rightarrow & \textit{Category}\ [\textit{Connector}\ \textit{Modifier}]* \\
\textit{Connector} & \rightarrow & \text{“}\textit{with}\text{”} \mid \text{“}\textit{and}\text{”} \mid \text{“,”} \\
\textit{Modifier} & \rightarrow & \textit{Attribute} \mid \textit{Attribute}\ \textit{Op}\ \textit{Value} \\
\textit{Op} & \rightarrow & \text{“}=\text{”} \mid \text{“}>\text{”} \mid \text{“}<\text{”}
\end{array}
$$

Here, *Category* names a product category, *Attribute* is some feature such as "CPU" or "price," and *Value* is the target value for the attribute. So the query "computer with at least a 2.5-GHz CPU for under $1000" must be re-expressed as "computer with CPU $>$ 2.5 GHz and price $<$ $1000." Implement a shopping agent that accepts descriptions in this language.

10.17 Our description of Internet shopping omitted the all-important step of actually *buying* the product. Provide a formal logical description of buying, using event calculus. That is, define the sequence of events that occurs when a buyer submits a credit card purchase and then eventually gets billed and receives the product.

10.18 Describe the event of trading something for something else. Describe buying as a kind of trading in which one of the objects traded is a sum of money.



10.19 The two preceding exercises assume a fairly primitive notion of ownership. For example, the buyer starts by *owning* the dollar bills. This picture begins to break down when, for example, one's money is in the bank, because there is no longer any specific collection of dollar bills that one owns. The picture is complicated still further by borrowing, leasing, renting, and bailment. Investigate the various commonsense and legal concepts of ownership, and propose a scheme by which they can be represented formally.

10.20 You are to create a system for advising computer science undergraduates on what courses to take over an extended period in order to satisfy the program requirements. (Use whatever requirements are appropriate for your institution.) First, decide on a vocabulary for representing all the information, and then represent it; then use an appropriate query to the system, that will return a legal program of study as a solution. You should allow for some tailoring to individual students, in that your system should ask what courses or equivalents the student has already taken, and not generate programs that repeat those courses.

Suggest ways in which your system could be improved—for example to take into account knowledge about student preferences, the workload, good and bad instructors, and so on. For each kind of knowledge, explain how it could be expressed logically. Could your system easily incorporate this information to find the *best* program of study for a student?

10.21 Figure 10.1 shows the top levels of a hierarchy for everything. Extend it to include as many real categories as possible. A good way to do this is to cover all the things in your everyday life. This includes objects and events. Start with waking up, and proceed in an orderly fashion noting everything that you see, touch, do, and think about. For example, a random sampling produces music, news, milk, walking, driving, gas, Soda Hall, carpet, talking, Professor Fateman, chicken curry, tongue, $7, sun, the daily newspaper, and so on.

You should produce both a single hierarchy chart (on a large sheet of paper) and a listing of objects and categories with the relations satisfied by members of each category. Every object should be in a category, and every category should be in the hierarchy.

10.22 (Adapted from an example by Doug Lenat.) Your mission is to capture, in logical form, enough knowledge to answer a series of questions about the following simple sentence:

> Yesterday John went to the North Berkeley Safeway supermarket and bought two pounds of tomatoes and a pound of ground beef.

Start by trying to represent the content of the sentence as a series of assertions. You should write sentences that have straightforward logical structure (e.g., statements that objects have certain properties, that objects are related in certain ways, that all objects satisfying one property satisfy another). The following might help you get started:

- Which classes, objects, and relations would you need? What are their parents, siblings and so on? (You will need events and temporal ordering, among other things.)
- Where would they fit in a more general hierarchy?
- What are the constraints and interrelationships among them?
- How detailed must you be about each of the various concepts?

The knowledge base you construct must be capable of answering a list of questions that we will give shortly. Some of the questions deal with the material stated explicitly in the story,

but most of them require one to have other background knowledge—to read between the lines. You'll have to deal with what kind of things are at a supermarket, what is involved with purchasing the things one selects, what will purchases be used for, and so on. Try to make your representation as general as possible. To give a trivial example: don't say "People buy food from Safeway," because that won't help you with those who shop at another supermarket. Don't say "Joe made spaghetti with the tomatoes and ground beef," because that won't help you with anything else at all. Also, don't turn the questions into answers; for example, question (c) asks "Did John buy any meat?"—not "Did John buy a pound of ground beef?"

Sketch the chains of reasoning that would answer the questions. In the process of doing so, you will no doubt need to create additional concepts, make additional assertions, and so on. If possible, use a logical reasoning system to demonstrate the sufficiency of your knowledge base. Many of the things you write might be only approximately correct in reality, but don't worry too much; the idea is to extract the common sense that lets you answer these questions at all. A truly complete answer to this question is *extremely* difficult, probably beyond the state of the art of current knowledge representation. But you should be able to put together a consistent set of axioms for the limited questions posed here.

a. Is John a child or an adult? [Adult]

b. Does John now have at least two tomatoes? [Yes]

c. Did John buy any meat? [Yes]

d. If Mary was buying tomatoes at the same time as John, did he see her? [Yes]

e. Are the tomatoes made in the supermarket? [No]

f. What is John going to do with the tomatoes? [Eat them]

g. Does Safeway sell deodorant? [Yes]

h. Did John bring any money to the supermarket? [Yes]

i. Does John have less money after going to the supermarket? [Yes]

10.23 Make the necessary additions or changes to your knowledge base from the previous exercise so that the questions that follow can be answered. Show that they can indeed be answered by the KB, and include in your report a discussion of the fixes, explaining why they were needed, whether they were minor or major, and so on.

a. Are there other people in Safeway while John is there? [Yes—staff!]

b. Is John a vegetarian? [No]

c. Who owns the deodorant in Safeway? [Safeway Corporation]

d. Did John have an ounce of ground beef? [Yes]

e. Does the Shell station next door have any gas? [Yes]

f. Do the tomatoes fit in John's car trunk? [Yes]

10.24 Recall that inheritance information in semantic networks can be captured logically by suitable implication sentences. In this exercise, we will consider the efficiency of using such sentences for inheritance.

a. Consider the information content in a used-car catalog such as Kelly's Blue Book—for example, that 1973 Dodge Vans are worth $575. Suppose all this information (for 11,000 models) is encoded as logical rules, as suggested in the chapter. Write down three such rules, including that for 1973 Dodge Vans. How would you use the rules to find the value of a *particular* car (e.g., JB, which is a 1973 Dodge Van), given a backward-chaining theorem prover such as Prolog?

b. Compare the time efficiency of the backward-chaining method for solving this problem with the inheritance method used in semantic nets.

c. Explain how forward chaining allows a logic-based system to solve the same problem efficiently, assuming that the KB contains only the 11,000 rules about prices.

d. Describe a situation in which neither forward nor backward chaining on the rules will allow the price query for an individual car to be handled efficiently.

e. Can you suggest a solution enabling this type of query to be solved efficiently in all cases in logic systems? [*Hint:* Remember that two cars of the same category have the same price.]

10.25 One might suppose that the syntactic distinction between unboxed links and singly boxed links in semantic networks is unnecessary, because singly boxed links are always attached to categories; an inheritance algorithm could simply assume that an unboxed link attached to a category is intended to apply to all members of that category. Show that this argument is fallacious, giving examples of errors that would arise.

11 PLANNING

In which we see how an agent can take advantage of the structure of a problem to construct complex plans of action.

The task of coming up with a sequence of actions that will achieve a goal is called **planning**. We have seen two examples of planning agents so far: the search-based problem-solving agent of Chapter 3 and the logical planning agent of Chapter 10. This chapter is concerned primarily with *scaling up* to complex planning problems that defeat the approaches we have seen so far.

Section 11.1 develops an expressive yet carefully constrained language for representing planning problems, including actions and states. The language is closely related to the propositional and first-order representations of actions in Chapters 7 and 10. Section 11.2 shows how forward and backward search algorithms can take advantage of this representation, primarily through accurate heuristics that can be derived automatically from the structure of the representation. (This is analogous to the way in which effective heuristics were constructed for constraint satisfaction problems in Chapter 5.) Sections 11.3 through 11.5 describe planning algorithms that go beyond forward and backward search, taking advantage of the representation of the problem. In particular, we explore approaches that are not constrained to consider only totally ordered sequences of actions.

For this chapter, we consider only environments that are fully observable, deterministic, finite, static (change happens only when the agent acts), and discrete (in time, action, objects, and effects). These are called **classical planning** environments. In contrast, nonclassical planning is for partially observable or stochastic environments and involves a different set of algorithms and agent designs, outlined in Chapters 12 and 17.

CLASSICAL PLANNING

11.1 THE PLANNING PROBLEM

Let us consider what can happen when an ordinary problem-solving agent using standard search algorithms—depth-first, A*, and so on—comes up against large, real-world problems. That will help us design better planning agents.

The most obvious difficulty is that the problem-solving agent can be overwhelmed by irrelevant actions. Consider the task of buying a copy of *AI: A Modern Approach* from an online bookseller. Suppose there is one buying action for each 10-digit ISBN number, for a total of 10 billion actions. The search algorithm would have to examine the outcome states of all 10 billion actions to find one that satisfies the goal, which is to own a copy of ISBN 0137903952. A sensible planning agent, on the other hand, should be able to work back from an explicit goal description such as $Have(ISBN0137903952)$ and generate the action $Buy(ISBN0137903952)$ directly. To do this, the agent simply needs the general knowledge that $Buy(x)$ results in $Have(x)$. Given this knowledge and the goal, the planner can decide in a single unification step that $Buy(ISBN0137903952)$ is the right action.

The next difficulty is finding a good **heuristic function**. Suppose the agent's goal is to buy four different books online. Then there will be 10^{40} plans of just four steps, so searching without an accurate heuristic is out of the question. It is obvious to a human that a good heuristic estimate for the cost of a state is the number of books that remain to be bought; unfortunately, this insight is not obvious to a problem-solving agent, because it sees the goal test only as a black box that returns true or false for each state. Therefore, the problem-solving agent lacks autonomy; it requires a human to supply a heuristic function for each new problem. On the other hand, if a planning agent has access to an explicit representation of the goal as a conjunction of subgoals, then it can use a single *domain-independent* heuristic: the number of unsatisfied conjuncts. For the book-buying problem, the goal would be $Have(A) \wedge Have(B) \wedge Have(C) \wedge Have(D)$, and a state containing $Have(A) \wedge Have(C)$ would have cost 2. Thus, the agent automatically gets the right heuristic for this problem, and for many others. We shall see later in the chapter how to construct more sophisticated heuristics that examine the available actions as well as the structure of the goal.

PROBLEM DECOMPOSITION

Finally, the problem solver might be inefficient because it cannot take advantage of **problem decomposition**. Consider the problem of delivering a set of overnight packages to their respective destinations, which are scattered across Australia. It makes sense to find out the nearest airport for each destination and divide the overall problem into several subproblems, one for each airport. Within the set of packages routed through a given airport, whether further decomposition is possible depends on the destination city. We saw in Chapter 5 that the ability to do this kind of decomposition contributes to the efficiency of constraint satisfaction problem solvers. The same holds true for planners: in the worst case, it can take $O(n!)$ time to find the best plan to deliver n packages, but only $O((n/k)! \times k)$ time if the problem can be decomposed into k equal parts.

As we noted in Chapter 5, perfectly decomposable problems are delicious but rare.[1] The design of many planning systems—particularly the partial-order planners described in Section 11.3—is based on the assumption that most real-world problems are **nearly decomposable**. That is, the planner can work on subgoals independently, but might need to do some additional work to combine the resulting subplans. For some problems, this assump-

NEARLY DECOMPOSABLE

[1] Notice that even the delivery of a package is not perfectly decomposable. There may be cases in which it is better to assign packages to a more distant airport if that renders a flight to the nearest airport unnecessary. Nevertheless, most delivery companies prefer the computational and organizational simplicity of sticking with decomposed solutions.

tion breaks down because working on one subgoal is likely to undo another subgoal. These interactions among subgoals are what makes puzzles (like the 8-puzzle) puzzling.

The language of planning problems

The preceding discussion suggests that the representation of planning problems—states, actions, and goals—should make it possible for planning algorithms to take advantage of the logical structure of the problem. The key is to find a language that is expressive enough to describe a wide variety of problems, but restrictive enough to allow efficient algorithms to operate over it. In this section, we first outline the basic representation language of classical planners, known as the STRIPS language.[2] Later, we point out some of the many possible variations in STRIPS-like languages.

Representation of states. Planners decompose the world into logical conditions and represent a state as a conjunction of positive literals. We will consider propositional literals; for example, $Poor \wedge Unknown$ might represent the state of a hapless agent. We will also use first-order literals; for example, $At(Plane_1, Melbourne) \wedge At(Plane_2, Sydney)$ might represent a state in the package delivery problem. Literals in first-order state descriptions must be **ground** and **function-free**. Literals such as $At(x, y)$ or $At(Father(Fred), Sydney)$ are not allowed. The **closed-world assumption** is used, meaning that any conditions that are not mentioned in a state are assumed false.

Representation of goals. A goal is a partially specified state, represented as a conjunction of positive ground literals, such as $Rich \wedge Famous$ or $At(P_2, Tahiti)$. A propositional

GOAL SATISFACTION

state s **satisfies** a goal g if s contains all the atoms in g (and possibly others). For example, the state $Rich \wedge Famous \wedge Miserable$ satisfies the goal $Rich \wedge Famous$.

Representation of actions. An action is specified in terms of the preconditions that must hold before it can be executed and the effects that ensue when it is executed. For example, an action for flying a plane from one location to another is:

$$\begin{array}{l} Action(Fly(p, from, to), \\ \quad \text{PRECOND:} At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to) \\ \quad \text{EFFECT:} \neg At(p, from) \wedge At(p, to)) \end{array}$$

ACTION SCHEMA

This is more properly called an **action schema**, meaning that it represents a number of different actions that can be derived by instantiating the variables p, $from$, and to to different constants. In general, an action schema consists of three parts:

- The action name and parameter list—for example, $Fly(p, from, to)$—serves to identify the action.

PRECONDITION

- The **precondition** is a conjunction of function-free positive literals stating what must be true in a state before the action can be executed. Any variables in the precondition must also appear in the action's parameter list.

EFFECT

- The **effect** is a conjunction of function-free literals describing how the state changes when the action is executed. A positive literal P in the effect is asserted to be true in

[2] STRIPS stands for STanford Research Institute Problem Solver.

the state resulting from the action, whereas a negative literal $\neg P$ is asserted to be false. Variables in the effect must also appear in the action's parameter list.

ADD LIST
DELETE LIST

To improve readability, some planning systems divide the effect into the **add list** for positive literals and the **delete list** for negative literals.

Having defined the syntax for representations of planning problems, we can now define the semantics. The most straightforward way to do this is to describe how actions affect states. (An alternative method is to specify a direct translation into successor-state axioms, whose semantics comes from first-order logic; see Exercise 11.3.) First, we say that an action

APPLICABLE

is **applicable** in any state that satisfies the precondition; otherwise, the action has no effect. For a first-order action schema, establishing applicability will involve a substitution θ for the variables in the precondition. For example, suppose the current state is described by

$$At(P_1, JFK) \wedge At(P_2, SFO) \wedge Plane(P_1) \wedge Plane(P_2) \\ \wedge Airport(JFK) \wedge Airport(SFO) .$$

This state satisfies the precondition

$$At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$$

with substitution $\{p/P_1, from/JFK, to/SFO\}$ (among others—see Exercise 11.2). Thus, the concrete action $Fly(P_1, JFK, SFO)$ is applicable.

RESULT

Starting in state s, the **result** of executing an applicable action a is a state s' that is the same as s except that any positive literal P in the effect of a is added to s' and any negative literal $\neg P$ is removed from s'. Thus, after $Fly(P_1, JFK, SFO)$, the current state becomes

$$At(P_1, SFO) \wedge At(P_2, SFO) \wedge Plane(P_1) \wedge Plane(P_2) \\ \wedge Airport(JFK) \wedge Airport(SFO) .$$

Note that if a positive effect is already in s it is not added twice, and if a negative effect is not in s, then that part of the effect is ignored. This definition embodies the so-called STRIPS

STRIPS ASSUMPTION

assumption: that every literal not mentioned in the effect remains unchanged. In this way, STRIPS avoids the representational **frame problem** described in Chapter 10.

SOLUTION

Finally, we can define the **solution** for a planning problem. In its simplest form, this is just an action sequence that, when executed in the initial state, results in a state that satisfies the goal. Later in the chapter, we will allow solutions to be partially ordered sets of actions, provided that every action sequence that respects the partial order is a solution.

Expressiveness and extensions

The various restrictions imposed by the STRIPS representation were chosen in the hope of making planning algorithms simpler and more efficient, without making it too difficult to describe real problems. One of the most important restrictions is that literals be *function-free*. With this restriction, we can be sure that any action schema for a given problem can be propositionalized—that is, turned into a finite collection of purely propositional action representations with no variables. (See Chapter 9 for more on this topic.) For example, in the air cargo domain for a problem with 10 planes and five airports, we could translate the $Fly(p, from, to)$ schema into $10 \times 5 \times 5 = 250$ purely propositional actions. The planners

STRIPS **Language**	ADL **Language**
Only positive literals in states: $Poor \wedge Unknown$	Positive and negative literals in states: $\neg Rich \wedge \neg Famous$
Closed World Assumption: Unmentioned literals are false.	Open World Assumption: Unmentioned literals are unknown.
Effect $P \wedge \neg Q$ means add P and delete Q.	Effect $P \wedge \neg Q$ means add P and $\neg Q$ and delete $\neg P$ and Q.
Only ground literals in goals: $Rich \wedge Famous$	Quantified variables in goals: $\exists x\, At(P_1, x) \wedge At(P_2, x)$ is the goal of having P_1 and P_2 in the same place.
Goals are conjunctions: $Rich \wedge Famous$	Goals allow conjunction and disjunction: $\neg Poor \wedge (Famous \vee Smart)$
Effects are conjunctions.	Conditional effects allowed: **when** P: E means E is an effect only if P is satisfied.
No support for equality.	Equality predicate $(x = y)$ is built in.
No support for types.	Variables can have types, as in $(p : Plane)$.

Figure 11.1 Comparison of STRIPS and ADL languages for representing planning problems. In both cases, goals behave as the preconditions of an action with no parameters.

in Sections 11.4 and 11.5 work directly with propositionalized descriptions. If we allow function symbols, then infinitely many states and actions can be constructed.

In recent years, it has become clear that STRIPS is insufficiently expressive for some real domains. As a result, many language variants have been developed. Figure 11.1 briefly describes one important one, the Action Description Language or **ADL**, by comparing it with the basic STRIPS language. In ADL, the *Fly* action could be written as

ADL

$$\begin{aligned} &Action(Fly(p : Plane, from : Airport, to : Airport),\\ &\quad \text{PRECOND:} At(p, from) \wedge (from \neq to)\\ &\quad \text{EFFECT:} \neg At(p, from) \wedge At(p, to)) \ . \end{aligned}$$

The notation $p : Plane$ in the parameter list is an abbreviation for $Plane(p)$ in the precondition; this adds no expressive power, but can be easier to read. (It also cuts down on the number of possible propositional actions that can be constructed.) The precondition $(from \neq to)$ expresses the fact that a flight cannot be made from an airport to itself. This could not be expressed succinctly in STRIPS.

The various planning formalisms used in AI have been systematized within a standard syntax called the Planning Domain Definition Language, or PDDL. This language allows researchers to exchange benchmark problems and compare results. PDDL includes sublanguages for STRIPS, ADL, and the hierarchical task networks we will see in Chapter 12.

$Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK)$
$\quad \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2)$
$\quad \land Airport(JFK) \land Airport(SFO))$
$Goal(At(C_1, JFK) \land At(C_2, SFO))$
$Action(Load(c, p, a),$
$\quad$ PRECOND: $At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
$\quad$ EFFECT: $\neg At(c, a) \land In(c, p))$
$Action(Unload(c, p, a),$
$\quad$ PRECOND: $In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
$\quad$ EFFECT: $At(c, a) \land \neg In(c, p))$
$Action(Fly(p, from, to),$
$\quad$ PRECOND: $At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)$
$\quad$ EFFECT: $\neg At(p, from) \land At(p, to))$

Figure 11.2 A STRIPS problem involving transportation of air cargo between airports.

The STRIPS and ADL notations are adequate for many real domains. The subsections that follow show some simple examples. There are still some significant restrictions, however. The most obvious is that they cannot represent in a natural way the **ramifications** of actions. For example, if there are people, packages, or dust motes in the airplane, then they too change location when the plane flies. We can represent these changes as the direct effects of flying, whereas it seems more natural to represent the location of the plane's contents as a logical consequence of the location of the plane. We will see more examples of such **state constraints** in Section 11.5. Classical planning systems do not even attempt to address the **qualification** problem: the problem of unrepresented circumstances that could cause an action to fail. We will see how to address qualifications in Chapter 12.

STATE CONSTRAINTS

Example: Air cargo transport

Figure 11.2 shows an air cargo transport problem involving loading and unloading cargo onto and off of planes and flying it from place to place. The problem can be defined with three actions: *Load*, *Unload*, and *Fly*. The actions affect two predicates: $In(c, p)$ means that cargo c is inside plane p, and $At(x, a)$ means that object x (either plane or cargo) is at airport a. Note that cargo is not At anywhere when it is In a plane, so At really means "available for use at a given location." It takes some experience with action definitions to handle such details consistently. The following plan is a solution to the problem:

$$[Load(C_1, P_1, SFO), Fly(P_1, SFO, JFK), Unload(C_1, P_1, JFK),$$
$$Load(C_2, P_2, JFK), Fly(P_2, JFK, SFO), Unload(C_2, P_2, SFO)] .$$

Our representation is pure STRIPS. In particular, it allows a plane to fly to and from the same airport. Inequality literals in ADL could prevent this.

Example: The spare tire problem

Consider the problem of changing a flat tire. More precisely, the goal is to have a good spare tire properly mounted onto the car's axle, where the initial state has a flat tire on the axle and a good spare tire in the trunk. To keep it simple, our version of the problem is a very abstract one, with no sticky lug nuts or other complications. There are just four actions: removing the spare from the trunk, removing the flat tire from the axle, putting the spare on the axle, and leaving the car unattended overnight. We assume that the car is in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear.

The ADL description of the problem is shown in Figure 11.3. Notice that it is purely propositional. It goes beyond STRIPS in that it uses a negated precondition, $\neg At(Flat, Axle)$, for the $PutOn(Spare, Axle)$ action. This could be avoided by using $Clear(Axle)$ instead, as we will see in the next example.

$Init(At(Flat, Axle) \wedge At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(Spare, Trunk),$
  PRECOND: $At(Spare, Trunk)$
  EFFECT: $\neg At(Spare, Trunk) \wedge At(Spare, Ground))$
$Action(Remove(Flat, Axle),$
  PRECOND: $At(Flat, Axle)$
  EFFECT: $\neg At(Flat, Axle) \wedge At(Flat, Ground))$
$Action(PutOn(Spare, Axle),$
  PRECOND: $At(Spare, Ground) \wedge \neg At(Flat, Axle)$
  EFFECT: $\neg At(Spare, Ground) \wedge At(Spare, Axle))$
$Action(LeaveOvernight,$
  PRECOND:
  EFFECT: $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)$
    $\wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle))$

Figure 11.3 The simple spare tire problem.

Example: The blocks world

BLOCKS WORLD

One of the most famous planning domains is known as the **blocks world**. This domain consists of a set of cube-shaped blocks sitting on a table.[3] The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it to another position, either on the table or on top of another block. The arm can pick up only one block at a time, so it cannot pick up a block that has another one on it. The goal will always be to build one or more stacks of blocks, specified in terms of what blocks are on top of what other blocks. For example, a goal might be to get block A on B and block C on D.

[3] The blocks world used in planning research is much simpler than SHRDLU's version, shown on page 20.

We will use $On(b, x)$ to indicate that block b is on x, where x is either another block or the table. The action for moving block b from the top of x to the top of y will be $Move(b, x, y)$. Now, one of the preconditions on moving b is that no other block be on it. In first-order logic, this would be $\neg\exists\, x\ On(x, b)$ or, alternatively, $\forall\, x\ \neg On(x, b)$. These could be stated as preconditions in ADL. We can stay within the STRIPS language, however, by introducing a new predicate, $Clear(x)$, that is true when nothing is on x.

The action $Move$ moves a block b from x to y if both b and y are clear. After the move is made, x is clear but y is not. A formal description of $Move$ in STRIPS is

$Action(Move(b, x, y)$,
PRECOND:$On(b, x) \wedge Clear(b) \wedge Clear(y)$,
EFFECT:$On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y))$.

Unfortunately, this action does not maintain $Clear$ properly when x or y is the table. When $x = Table$, this action has the effect $Clear(Table)$, but the table should not become clear, and when $y = Table$, it has the precondition $Clear(Table)$, but the table does not have to be clear to move a block onto it. To fix this, we do two things. First, we introduce another action to move a block b from x to the table:

$Action(MoveToTable(b, x)$,
PRECOND:$On(b, x) \wedge Clear(b))$,
EFFECT:$On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))$.

Second, we take the interpretation of $Clear(b)$ to be "there is a clear space on b to hold a block." Under this interpretation, $Clear(Table)$ will always be true. The only problem is that nothing prevents the planner from using $Move(b, x, Table)$ instead of $MoveToTable(b, x)$. We could live with this problem—it will lead to a larger-than-necessary search space, but will not lead to incorrect answers—or we could introduce the predicate $Block$ and add $Block(b) \wedge Block(y)$ to the precondition of $Move$.

Finally, there is the problem of spurious actions such as $Move(B, C, C)$, which should be a no-op, but which has contradictory effects. It is common to ignore such problems, because they seldom cause incorrect plans to be produced. The correct approach is add inequality preconditions as shown in Figure 11.4.

11.2 PLANNING WITH STATE-SPACE SEARCH

Now we turn our attention to planning algorithms. The most straightforward approach is to use state-space search. Because the descriptions of actions in a planning problem specify both preconditions and effects, it is possible to search in either direction: either forward from the initial state or backward from the goal, as shown in Figure 11.5. We can also use the explicit action and goal representations to derive effective heuristics automatically.

Forward state-space search

Planning with forward state-space search is similar to the problem-solving approach of Chapter 3. It is sometimes called **progression** planning, because it moves in the forward direction.

PROGRESSION

$Init(On(A, Table) \wedge On(B, Table) \wedge On(C, Table)$
$\wedge Block(A) \wedge Block(B) \wedge Block(C)$
$\wedge Clear(A) \wedge Clear(B) \wedge Clear(C))$
$Goal(On(A, B) \wedge On(B, C))$
$Action(Move(b, x, y),$
PRECOND: $On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge$
$(b \neq x) \wedge (b \neq y) \wedge (x \neq y),$
EFFECT: $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y))$
$Action(MoveToTable(b, x),$
PRECOND: $On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x),$
EFFECT: $On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))$

Figure 11.4 A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[Move(B, Table, C), Move(A, Table, B)]$.

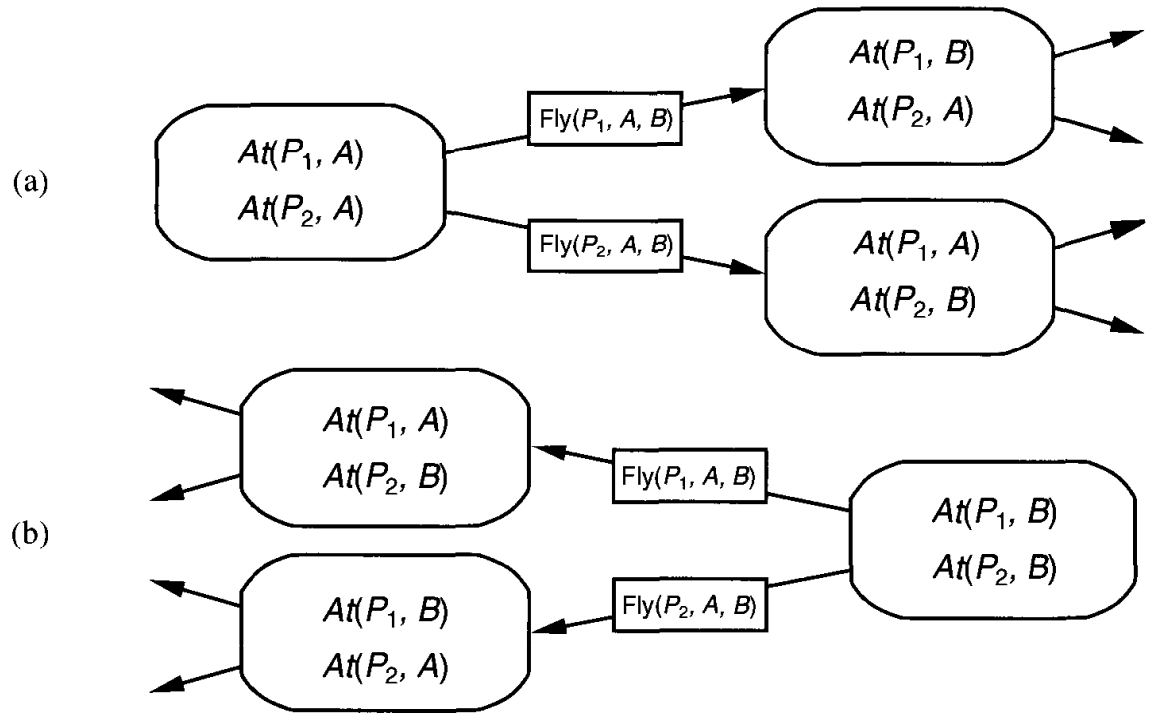


Figure 11.5 Two approaches to searching for a plan. (a) Forward (progression) state-space search, starting in the initial state and using the problem's actions to search forward for the goal state. (b) Backward (regression) state-space search: a belief-state search (see page 84) starting at the goal state(s) and using the inverse of the actions to search backward for the initial state.

We start in the problem's initial state, considering sequences of actions until we find a sequence that reaches a goal state. The formulation of planning problems as state-space search problems is as follows:

- The **initial state** of the search is the initial state from the planning problem. In general, each state will be a set of positive ground literals; literals not appearing are false.

- The **actions** that are applicable to a state are all those whose preconditions are satisfied. The successor state resulting from an action is generated by adding the positive effect literals and deleting the negative effect literals. (In the first-order case, we must apply the unifier from the preconditions to the effect literals.) Note that a single successor function works for all planning problems—a consequence of using an explicit action representation.
- The **goal test** checks whether the state satisfies the goal of the planning problem.
- The **step cost** of each action is typically 1. Although it would be easy to allow different costs for different actions, this is seldom done by STRIPS planners.

Recall that, in the absence of function symbols, the state space of a planning problem is finite. Therefore, any graph search algorithm that is complete—for example, A*—will be a complete planning algorithm.

From the earliest days of planning research (around 1961) until recently (around 1998) it was assumed that forward state-space search was too inefficient to be practical. It is not hard to come up with reasons why—just refer back to the start of Section 11.1. First, forward search does not address the irrelevant action problem—all applicable actions are considered from each state. Second, the approach quickly bogs down without a good heuristic. Consider an air cargo problem with 10 airports, where each airport has 5 planes and 20 pieces of cargo. The goal is to move all the cargo at airport A to airport B. There is a simple solution to the problem: load the 20 pieces of cargo into one of the planes at A, fly the plane to B, and unload the cargo. But finding the solution can be difficult because the average branching factor is huge: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded), or loaded into any plane at its airport (if it is unloaded). On average, let's say there are about 1000 possible actions, so the search tree up to the depth of the obvious solution has about 1000^{41} nodes. It is clear that a very accurate heuristic will be needed to make this kind of search efficient. We will discuss some possible heuristics after looking at backward search.

Backward state-space search

Backward state-space search was described briefly as part of bidirectional search in Chapter 3. We noted there that backward search can be difficult to implement when the goal states are described by a set of constraints rather than being listed explicitly. In particular, it is not always obvious how to generate a description of the possible **predecessors** of the set of goal states. We will see that the STRIPS representation makes this quite easy because sets of states can be described by the literals that must be true in those states.

RELEVANCE

The main advantage of backward search is that it allows us to consider only **relevant** actions. An action is relevant to a conjunctive goal if it achieves one of the conjuncts of the goal. For example, the goal in our 10-airport air cargo problem is to have 20 pieces of cargo at airport B, or more precisely,

$$At(C_1, B) \wedge At(C_2, B) \wedge \ldots \wedge At(C_{20}, B) .$$

Now consider the conjunct $At(C_1, B)$. Working backwards, we can seek actions that have this as an effect. There is only one: $Unload(C_1, p, B)$, where plane p is unspecified.

Notice that there are many *irrelevant* actions that can also lead to a goal state. For example, we can fly an empty plane from JFK to SFO; this action reaches a goal state from a predecessor state in which the plane is at JFK and all the goal conjuncts are satisfied. A backward search that allows irrelevant actions will still be complete, but it will be much less efficient. If a solution exists, it will be found by a backward search that allows only relevant actions. The restriction to relevant actions means that backward search often has a much lower branching factor than forward search. For example, our air cargo problem has about 1000 actions leading forward from the initial state, but only 20 actions working backward from the goal.

REGRESSION

Searching backwards is sometimes called **regression** planning. The principal question in regression planning is this: what are the states from which applying a given action leads to the goal? Computing the description of these states is called **regressing** the goal through the action. To see how to do it, consider the air cargo example. We have the goal

$$At(C_1, B) \wedge At(C_2, B) \wedge \ldots \wedge At(C_{20}, B)$$

and the relevant action $Unload(C_1, p, B)$, which achieves the first conjunct. The action will work only if its preconditions are satisfied. Therefore, any predecessor state must include these preconditions: $In(C_1, p) \wedge At(p, B)$. Moreover, the subgoal $At(C_1, B)$ should not be true in the predecessor state.[4] Thus, the predecessor description is

$$In(C_1, p) \wedge At(p, B) \wedge At(C_2, B) \wedge \ldots \wedge At(C_{20}, B) \ .$$

CONSISTENCY

In addition to insisting that actions achieve some desired literal, we must insist that the actions *not undo* any desired literals. An action that satisfies this restriction is called **consistent**. For example, the action $Load(C_2, p)$ would not be consistent with the current goal, because it would negate the literal $At(C_2, B)$.

Given definitions of relevance and consistency, we can describe the general process of constructing predecessors for backward search. Given a goal description G, let A be an action that is relevant and consistent. The corresponding predecessor is as follows:

- Any positive effects of A that appear in G are deleted.
- Each precondition literal of A is added, unless it already appears.

Any of the standard search algorithms can be used to carry out the search. Termination occurs when a predecessor description is generated that is satisfied by the initial state of the planning problem. In the first-order case, satisfaction might require a substitution for variables in the predecessor description. For example, the predecessor description in the preceding paragraph is satisfied by the initial state

$$In(C_1, P_{12}) \wedge At(P_{12}, B) \wedge At(C_2, B) \wedge \ldots \wedge At(C_{20}, B)$$

with substitution $\{p/P_{12}\}$. The substitution must be applied to the actions leading from the state to the goal, producing the solution $[Unload(C_1, P_{12}, B)]$.

[4] If the subgoal were true in the predecessor state, the action would still lead to a goal state. On the other hand, such actions are irrelevant because they do not *make* the goal true.

Heuristics for state-space search

It turns out that neither forward nor backward search is efficient without a good heuristic function. Recall from Chapter 4 that a heuristic function estimates the distance from a state to the goal; in STRIPS planning, the cost of each action is 1, so the distance is the number of actions. The basic idea is to look at the effects of the actions and at the goals that must be achieved and to guess how many actions are needed to achieve all the goals. Finding the exact number is NP hard, but it is possible to find reasonable estimates most of the time without too much computation. We might also be able to derive an **admissible** heuristic—one that does not overestimate. This could be used with A* search to find optimal solutions.

There are two approaches that can be tried. The first is to derive a **relaxed problem** from the given problem specification, as described in Chapter 4. The optimal solution cost for the relaxed problem—which we hope is very easy to solve—gives an admissible heuristic for the original problem. The second approach is to pretend that a pure divide-and-conquer algorithm will work. This is called the **subgoal independence** assumption: the cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving each subgoal *independently*. The subgoal independence assumption can be optimistic or pessimistic. It is optimistic when there are negative interactions between the subplans for each subgoal—for example, when an action in one subplan deletes a goal achieved by another subplan. It is pessimistic, and therefore inadmissible, when subplans contain redundant actions—for instance, two actions that could be replaced by a single action in the merged plan.

SUBGOAL INDEPENDENCE

Let us consider how to derive relaxed planning problems. Since explicit representations of preconditions and effects are available, the process will work by modifying those representations. (Compare this approach with search problems, where the successor function is a black box.) The simplest idea is to relax the problem by *removing all preconditions* from the actions. Then every action will always be applicable, and any literal can be achieved in one step (if there is an applicable action—if not, the goal is impossible). This almost implies that the number of steps required to solve a conjunction of goals is the number of unsatisfied goals—almost but not quite, because (1) there may be two actions, each of which deletes the goal literal achieved by the other, and (2) some action may achieve multiple goals. If we combine our relaxed problem with the subgoal independence assumption, both of these issues are assumed away and the resulting heuristic is exactly the number of unsatisfied goals.

In many cases, a more accurate heuristic is obtained by considering at least the positive interactions arising from actions that achieve multiple goals. First, we relax the problem further by *removing negative effects* (see Exercise 11.6). Then, we count the minimum number of actions required such that the union of those actions' positive effects satisfies the goal. For example, consider

$$Goal(A \wedge B \wedge C)$$
$$Action(X, \text{EFFECT:}A \wedge P)$$
$$Action(Y, \text{EFFECT:}B \wedge C \wedge Q)$$
$$Action(Z, \text{EFFECT:}B \wedge P \wedge Q)\ .$$

The minimal set cover of the goal $\{A, B, C\}$ is given by the actions $\{X, Y\}$, so the set cover heuristic returns a cost of 2. This improves on the subgoal independence assumption, which

gives a heuristic value of 3. There is one minor irritation: the set cover problem is NP-hard. A simple greedy set-covering algorithm is guaranteed to return a value that is within a factor of $\log n$ of the true minimum value, where n is the number of literals in the goal, and usually works much better than this in practice. Unfortunately, the greedy algorithm loses the guarantee of admissibility for the heuristic.

It is also possible to generate relaxed problems by removing negative effects without removing preconditions. That is, if an action has the effect $A \wedge \neg B$ in the original problem, it will have the effect A in the relaxed problem. This means that we need not worry about negative interactions between subplans, because no action can delete the literals achieved by another action. The solution cost of the resulting relaxed problem gives what is called the

EMPTY-DELETE-LIST

empty-delete-list heuristic. The heuristic is quite accurate, but computing it involves actually running a (simple) planning algorithm. In practice, the search in the relaxed problem is often fast enough that the cost is worthwhile.

The heuristics described here can be used in either the progression or the regression direction. At the time of writing, progression planners using the empty-delete-list heuristic hold the lead. That is likely to change as new heuristics and new search techniques are explored. Since planning is exponentially hard,[5] no algorithm will be efficient for all problems, but many practical problems can be solved with the heuristic methods in this chapter—far more than could be solved just a few years ago.

11.3 PARTIAL-ORDER PLANNING

Forward and backward state-space search are particular forms of *totally ordered* plan search. They explore only strictly linear sequences of actions directly connected to the start or goal. This means that they cannot take advantage of problem decomposition. Rather than work on each subproblem separately, they must always make decisions about how to sequence actions from all the subproblems. We would prefer an approach that works on several subgoals independently, solves them with several subplans, and then combines the subplans.

Such an approach also has the advantage of flexibility in the order in which it *constructs* the plan. That is, the planner can work on "obvious" or "important" decisions first, rather than being forced to work on steps in chronological order. For example, a planning agent that is in Berkeley and wishes to be in Monte Carlo might first try to find a flight from San Francisco to Paris; given information about the departure and arrival times, it can then work on ways to get to and from the airports.

LEAST COMMITMENT

The general strategy of delaying a choice during search is called a **least commitment** strategy. There is no formal definition of least commitment, and clearly some degree of commitment is necessary, lest the search would make no progress. Despite the informality, least commitment is a useful concept for analyzing when decisions should be made in any search problem.

[5] Technically, STRIPS-style planning is PSPACE-complete unless actions have only positive preconditions and only one effect literal (Bylander, 1994).

Our first concrete example will be much simpler than planning a vacation. Consider the simple problem of putting on a pair of shoes. We can describe this as a formal planning problem as follows:

$$Goal(RightShoeOn \wedge LeftShoeOn)$$
$$Init()$$
$$Action(RightShoe, \text{PRECOND:}RightSockOn, \text{EFFECT:}RightShoeOn)$$
$$Action(RightSock, \text{EFFECT:}RightSockOn)$$
$$Action(LeftShoe, \text{PRECOND:}LeftSockOn, \text{EFFECT:}LeftShoeOn)$$
$$Action(LeftSock, \text{EFFECT:}LeftSockOn) .$$

A planner should be able to come up with the two-action sequence *RightSock* followed by *RightShoe* to achieve the first conjunct of the goal and the sequence *LeftSock* followed by *LeftShoe* for the second conjunct. Then the two sequences can be combined to yield the final plan. In doing this, the planner will be manipulating the two subsequences independently, without committing to whether an action in one sequence is before or after an action in the other. Any planning algorithm that can place two actions into a plan without specifying which comes first is called a **partial-order planner**. Figure 11.6 shows the partial-order plan that is the solution to the shoes and socks problem. Note that the solution is represented as a *graph* of actions, not a sequence. Note also the "dummy" actions called *Start* and *Finish*, which mark the beginning and end of the plan. Calling them actions symplifies things, because now every step of a plan is an action. The partial-order solution corresponds to six possible total-order plans; each of these is called a **linearization** of the partial-order plan.

PARTIAL-ORDER PLANNER

LINEARIZATION

Partial-order planning can be implemented as a search in the space of partial-order plans. (From now on, we will just call them "plans.") That is, we start with an empty plan. Then we consider ways of refining the plan until we come up with a complete plan that solves the problem. The actions in this search are not actions in the world, but actions on plans: adding a step to the plan, imposing an ordering that puts one action before another, and so on.

We will define the POP algorithm for partial-order planning. It is traditional to write out the POP algorithm as a stand-alone program, but we will instead formulate partial-order planning as an instance of a search problem. This allows us to focus on the plan refinement steps that can be applied, rather than worrying about how the algorithm explores the space. In fact, a wide variety of uninformed or heuristic search methods can be applied once the search problem is formulated.

Remember that the states of our search problem will be (mostly unfinished) plans. To avoid confusion with the states of the world, we will talk about plans rather than states. Each plan has the following four components, where the first two define the steps of the plan and the last two serve a bookkeeping function to determine how plans can be extended:

- A set of **actions** that make up the steps of the plan. These are taken from the set of actions in the planning problem. The "empty" plan contains just the *Start* and *Finish* actions. *Start* has no preconditions and has as its effect all the literals in the initial state of the planning problem. *Finish* has no effects and has as its preconditions the goal literals of the planning problem.

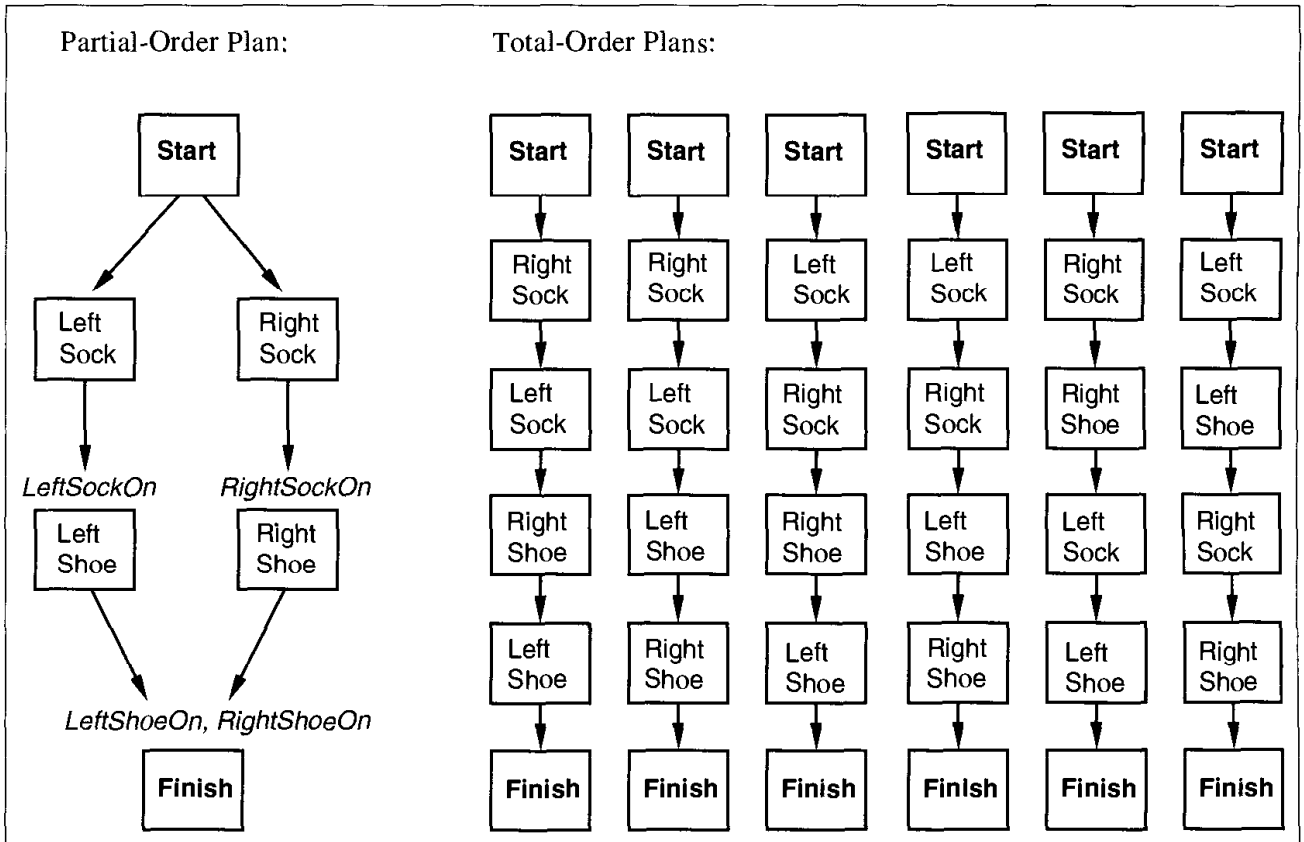


Figure 11.6 A partial-order plan for putting on shoes and socks, and the six corresponding linearizations into total-order plans.

ORDERING CONSTRAINTS

- A set of **ordering constraints**. Each ordering constraint is of the form $A \prec B$, which is read as "A before B" and means that action A must be executed sometime before action B, but not necessarily immediately before. The ordering constraints must describe a proper partial order. Any cycle—such as $A \prec B$ and $B \prec A$—represents a contradiction, so an ordering constraint cannot be added to the plan if it creates a cycle.

CAUSAL LINKS
ACHIEVES

- A set of **causal links**. A causal link between two actions A and B in the plan is written as $A \xrightarrow{p} B$ and is read as "A **achieves** p for B." For example, the causal link

$$RightSock \xrightarrow{RightSockOn} RightShoe$$

asserts that $RightSockOn$ is an effect of the $RightSock$ action and a precondition of $RightShoe$. It also asserts that $RightSockOn$ must remain true from the time of action $RightSock$ to the time of action $RightShoe$. In other words, the plan may not be extended by adding a new action C that **conflicts** with the causal link. An action C conflicts with $A \xrightarrow{p} B$ if C has the effect $\neg p$ and if C could (according to the ordering constraints) come after A and before B. Some authors call causal links **protection intervals**, because the link $A \xrightarrow{p} B$ protects p from being negated over the interval from A to B.

CONFLICTS

OPEN PRECONDITIONS

- A set of **open preconditions**. A precondition is open if it is not achieved by some action in the plan. Planners will work to reduce the set of open preconditions to the empty set, without introducing a contradiction.

For example, the final plan in Figure 11.6 has the following components (not shown are the ordering constraints that put every other action after *Start* and before *Finish*):

Actions:{*RightSock*, *RightShoe*, *LeftSock*, *LeftShoe*, *Start*, *Finish*}
Orderings:{*RightSock* $\prec$ *RightShoe*, *LeftSock* $\prec$ *LeftShoe*}
Links:{*RightSock* $\xrightarrow{RightSockOn}$ *RightShoe*, *LeftSock* $\xrightarrow{LeftSockOn}$ *LeftShoe*,
RightShoe $\xrightarrow{RightShoeOn}$ *Finish*, *LeftShoe* $\xrightarrow{LeftShoeOn}$ *Finish*}
Open Preconditions:{ } .

CONSISTENT PLAN

We define a **consistent plan** as a plan in which there are no cycles in the ordering constraints and no conflicts with the causal links. A consistent plan with no open preconditions is a **solution**. A moment's thought should convince the reader of the following fact: *every linearization of a partial-order solution is a total-order solution whose execution from the initial state will reach a goal state.* This means that we can extend the notion of "executing a plan" from total-order to partial-order plans. A partial-order plan is executed by repeatedly choosing *any* of the possible next actions. We will see in Chapter 12 that the flexibility available to the agent as it executes the plan can be very useful when the world fails to cooperate. The flexible ordering also makes it easier to combine smaller plans into larger ones, because each of the small plans can reorder its actions to avoid conflict with the other plans.

Now we are ready to formulate the search problem that POP solves. We will begin with a formulation suitable for propositional planning problems, leaving the first-order complications for later. As usual, the definition includes the initial state, actions, and goal test.

- The initial plan contains *Start* and *Finish*, the ordering constraint *Start* $\prec$ *Finish*, and no causal links and has all the preconditions in *Finish* as open preconditions.
- The successor function arbitrarily picks one open precondition p on an action B and generates a successor plan for every possible consistent way of choosing an action A that achieves p. Consistency is enforced as follows:
  1. The causal link $A \xrightarrow{p} B$ and the ordering constraint $A \prec B$ are added to the plan. Action A may be an existing action in the plan or a new one. If it is new, add it to the plan and also add *Start* $\prec A$ and $A \prec$ *Finish*.
  2. We resolve conflicts between the new causal link and all existing actions and between the action A (if it is new) and all existing causal links. A conflict between $A \xrightarrow{p} B$ and C is resolved by making C occur at some time outside the protection interval, either by adding $B \prec C$ or $C \prec A$. We add successor states for either or both if they result in consistent plans.
- The goal test checks whether a plan is a solution to the original planning problem. Because only consistent plans are generated, the goal test just needs to check that there are no open preconditions.

Remember that the actions considered by the search algorithms under this formulation are plan refinement steps rather than the real actions from the domain itself. The path cost is therefore irrelevant, strictly speaking, because the only thing that matters is the total cost of the real actions in the plan to which the path leads. Nonetheless, it *is* possible to specify a path cost function that reflects the real plan costs: we charge 1 for each real action added to

the plan and 0 for all other refinement steps. In this way, $g(n)$, where n is a plan, will be equal to the number of real actions in the plan. A heuristic estimate $h(n)$ can also be used.

At first glance, one might think that the successor function should include successors for *every* open p, not just for one of them. This would be redundant and inefficient, however, for the same reason that constraint satisfaction algorithms don't include successors for every possible variable: the order in which we consider open preconditions (like the order in which we consider CSP variables) is commutative. (See page 141.) Thus, we can choose an arbitrary ordering and still have a complete algorithm. Choosing the right ordering can lead to a faster search, but all orderings end up with the same set of candidate solutions.

A partial-order planning example

Now let's look at how POP solves the spare tire problem from Section 11.1. The problem description is repeated in Figure 11.7.

$Init(At(Flat, Axle) \wedge At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(Spare, Trunk),$
  PRECOND: $At(Spare, Trunk)$
  EFFECT: $\neg At(Spare, Trunk) \wedge At(Spare, Ground))$
$Action(Remove(Flat, Axle),$
  PRECOND: $At(Flat, Axle)$
  EFFECT: $\neg At(Flat, Axle) \wedge At(Flat, Ground))$
$Action(PutOn(Spare, Axle),$
  PRECOND: $At(Spare, Ground) \wedge \neg At(Flat, Axle)$
  EFFECT: $\neg At(Spare, Ground) \wedge At(Spare, Axle))$
$Action(LeaveOvernight,$
  PRECOND:
  EFFECT: $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)$
    $\wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle))$

Figure 11.7 The simple flat tire problem description.

The search for a solution begins with the initial plan, containing a *Start* action with the effect $At(Spare, Trunk) \wedge At(Flat, Axle)$ and a *Finish* action with the sole precondition $At(Spare, Axle)$. Then we generate successors by picking an open precondition to work on (irrevocably) and choosing among the possible actions to achieve it. For now, we will not worry about a heuristic function to help with these decisions; we will make seemingly arbitrary choices. The sequence of events is as follows:

1. Pick the only open precondition, $At(Spare, Axle)$ of *Finish*. Choose the only applicable action, $PutOn(Spare, Axle)$.
2. Pick the $At(Spare, Ground)$ precondition of $PutOn(Spare, Axle)$. Choose the only applicable action, $Remove(Spare, Trunk)$ to achieve it. The resulting plan is shown in Figure 11.8.

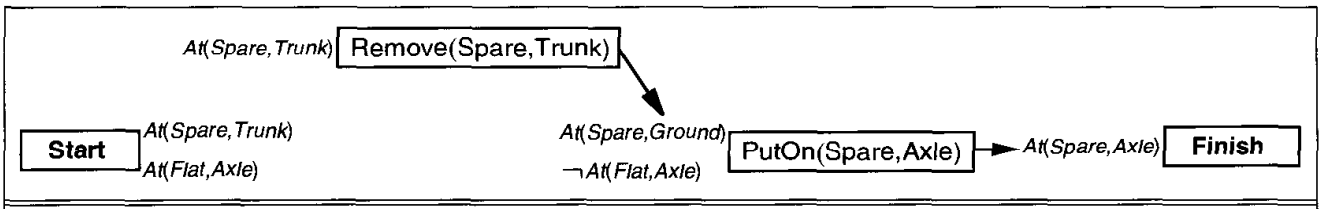


Figure 11.8 The incomplete partial-order plan for the tire problem, after choosing actions for the first two open preconditions. Boxes represent actions, with preconditions on the left and effects on the right. (Effects are omitted, except for that of the *Start* action.) Dark arrows represent causal links protecting the proposition at the head of the arrow.

3. Pick the $\neg At(Flat, Axle)$ precondition of $PutOn(Spare, Axle)$. Just to be contrary, choose the $LeaveOvernight$ action rather than the $Remove(Flat, Axle)$ action. Notice that $LeaveOvernight$ also has the effect $\neg At(Spare, Ground)$, which means it conflicts with the causal link

$$Remove(Spare, Trunk) \overset{At(Spare,Ground)}{\longrightarrow} PutOn(Spare, Axle) .$$

To resolve the conflict we add an ordering constraint putting $LeaveOvernight$ before $Remove(Spare, Trunk)$. The resulting plan is shown in Figure 11.9. (Why does this resolve the conflict, and why is there no other way to resolve it?)

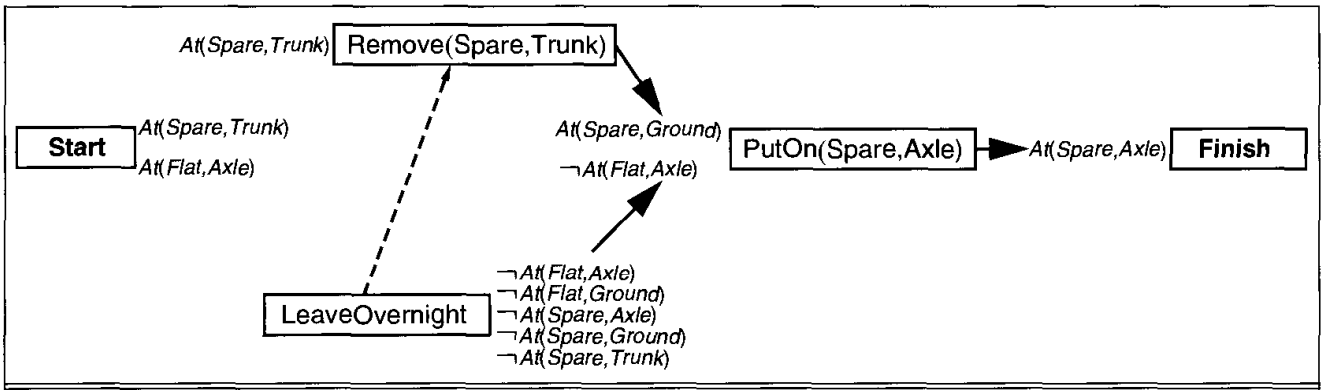


Figure 11.9 The plan after choosing $LeaveOvernight$ as the action for achieving $\neg At(Flat, Axle)$. To avoid a conflict with the causal link from $Remove(Spare, Trunk)$ that protects $At(Spare, Ground)$, $LeaveOvernight$ is constrained to occur before $Remove(Spare, Trunk)$, as shown by the dashed arrow.

4. The only remaining open precondition at this point is the $At(Spare, Trunk)$ precondition of the action $Remove(Spare, Trunk)$. The only action that can achieve it is the existing $Start$ action, but the causal link from $Start$ to $Remove(Spare, Trunk)$ is in conflict with the $\neg At(Spare, Trunk)$ effect of $LeaveOvernight$. This time there is no way to resolve the conflict with $LeaveOvernight$: we cannot order it before $Start$ (because nothing can come before $Start$), and we cannot order it after $Remove(Spare, Trunk)$ (because there is already a constraint ordering it before $Remove(Spare, Trunk)$). So we are forced to back up, remove the $LeaveOvernight$ action and the last two causal links, and return to the state in Figure 11.8. In essence, the planner has proved that $LeaveOvernight$ doesn't work as a way to change a tire.

5. Consider again the $\neg At(Flat, Axle)$ precondition of $PutOn(Spare, Axle)$. This time, we choose $Remove(Flat, Axle)$.
6. Once again, pick the $At(Spare, Trunk)$ precondition of $Remove(Spare, Trunk)$ and choose $Start$ to achieve it. This time there are no conflicts.
7. Pick the $At(Flat, Axle)$ precondition of $Remove(Flat, Axle)$, and choose $Start$ to achieve it. This gives us a complete, consistent plan—in other words a solution—as shown in Figure 11.10.

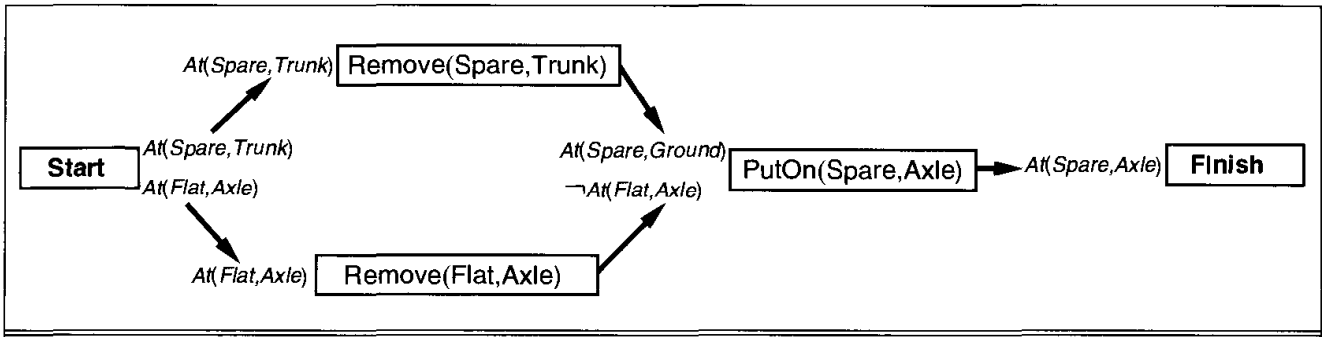


Figure 11.10 The final solution to the tire problem. Note that $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$ can be done in either order, as long as they are completed before the $PutOn(Spare, Axle)$ action.

Although this example is very simple, it illustrates some of the strengths of partial-order planning. First, the causal links lead to early pruning of portions of the search space that, because of irresolvable conflicts, contain no solutions. Second, the solution in Figure 11.10 is a partial-order plan. In this case the advantage is small, because there are only two possible linearizations; nonetheless, an agent might welcome the flexibility—for example, if the tire has to be changed in the middle of heavy traffic.

The example also points to some possible improvements that could be made. For example, there is duplication of effort: $Start$ is linked to $Remove(Spare, Trunk)$ before the conflict causes a backtrack and is then unlinked by backtracking even though it is not involved in the conflict. It is then relinked as the search continues. This is typical of chronological backtracking and might be mitigated by dependency-directed backtracking.

Partial-order planning with unbound variables

In this section, we consider the complications that can arise when POP is used with first-order action representations that include variables. Suppose we have a blocks world problem (Figure 11.4) with the open precondition $On(A, B)$ and the action

$$Action(Move(b, x, y),$$
$$\text{PRECOND:}On(b, x) \wedge Clear(b) \wedge Clear(y),$$
$$\text{EFFECT:}On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)) .$$

This action achieves $On(A, B)$ because the effect $On(b, y)$ unifies with $On(A, B)$ with the substitution $\{b/A, y/B\}$. We then apply this substitution to the action, yielding

$$Action(Move(A, x, B),$$
$$\text{PRECOND:}On(A, x) \wedge Clear(A) \wedge Clear(B),$$
$$\text{EFFECT:}On(A, B) \wedge Clear(x) \wedge \neg On(A, x) \wedge \neg Clear(B)) .$$

This leaves the variable x unbound. That is, the action says to move block A from *somewhere*, without yet saying whence. This is another example of the least commitment principle: we can delay making the choice until some other step in the plan makes it for us. For example, suppose we have $On(A, D)$ in the initial state. Then the $Start$ action can be used to achieve $On(A, x)$, binding x to D. This strategy of waiting for more information before choosing x is often more efficient than trying every possible value of x and backtracking for each one that fails.

The presence of variables in preconditions and actions complicates the process of detecting and resolving conflicts. For example, when $Move(A, x, B)$ is added to the plan, we will need a causal link

$$Move(A, x, B) \xrightarrow{On(A,B)} Finish .$$

If there is another action M_2 with effect $\neg On(A, z)$, then M_2 conflicts only if z is B. To accommodate this possibility, we extend the representation of plans to include a set of **inequality constraints** of the form $z \neq X$ where z is a variable and X is either another variable or a constant symbol. In this case, we would resolve the conflict by adding $z \neq B$, which means that future extensions to the plan can instantiate z to any value except B. Anytime we apply a substitution to a plan, we must check that the inequalities do not contradict the substitution. For example, a substitution that includes x/y conflicts with the inequality constraint $x \neq y$. Such conflicts cannot be resolved, so the planner must backtrack.

INEQUALITY CONSTRAINTS

A more extensive example of POP planning with variables in the blocks world is given in Section 12.6.

Heuristics for partial-order planning

Compared with total-order planning, partial-order planning has a clear advantage in being able to decompose problems into subproblems. It also has a disadvantage in that it does not represent states directly, so it is harder to estimate how far a partial-order plan is from achieving a goal. At present, there is less understanding of how to compute accurate heuristics for partial-order planning than for total-order planning.

The most obvious heuristic is to count the number of distinct open preconditions. This can be improved by subtracting the number of open preconditions that match literals in the $Start$ state. As in the total-order case, this overestimates the cost when there are actions that achieve multiple goals and underestimates the cost when there are negative interactions between plan steps. The next section presents an approach that allows us to get much more accurate heuristics from a relaxed problem.

The heuristic function is used to choose which plan to refine. Given this choice, the algorithm generates successors based on the selection of a single open precondition to work

on. As in the case of variable selection on constraint satisfaction algorithms, this selection has a large impact on efficiency. The **most-constrained-variable** heuristic from CSPs can be adapted for planning algorithms and seems to work well. The idea is to select the open condition that can be satisfied in the *fewest* number of ways. There are two special cases of this heuristic. First, if an open condition cannot be achieved by any action, the heuristic will select it; this is a good idea because early detection of impossibility can save a great deal of work. Second, if an open condition can be achieved in only one way, then it should be selected because the decision is unavoidable and could provide additional constraints on other choices still to be made. Although full computation of the number of ways to satisfy each open condition is expensive and not always worthwhile, experiments show that handling the two special cases provides very substantial speedups.

11.4 PLANNING GRAPHS

All of the heuristics we have suggested for total-order and partial-order planning can suffer from inaccuracies. This section shows how a special data structure called a **planning graph** can be used to give better heuristic estimates. These heuristics can be applied to any of the search techniques we have seen so far. Alternatively, we can extract a solution directly from the planning graph, using a specialized algorithm such as the one called GRAPHPLAN.

PLANNING GRAPH

LEVELS

A planning graph consists of a sequence of **levels** that correspond to time steps in the plan, where level 0 is the initial state. Each level contains a set of literals and a set of actions. Roughly speaking, the literals are all those that *could* be true at that time step, depending on the actions executed at preceding time steps. Also roughly speaking, the actions are all those actions that *could* have their preconditions satisfied at that time step, depending on which of the literals actually hold. We say "roughly speaking" because the planning graph records only a restricted subset of the possible negative interactions among actions; therefore, it might be optimistic about the minimum number of time steps required for a literal to become true. Nonetheless, this number of steps in the planning graph provides a good estimate of how difficult it is to achieve a given literal from the initial state. More importantly, the planning graph is defined in such a way that it can be constructed very efficiently.

Planning graphs work only for propositional planning problems—ones with no variables. As we mentioned in Section 11.1, both STRIPS and ADL representations can be propositionalized. For problems with large numbers of objects, this could result in a very substantial blowup in the number of action schemata. Despite this, planning graphs have proved to be effective tools for solving hard planning problems.

We will illustrate planning graphs with a simple example. (More complex examples lead to graphs that won't fit on the page.) Figure 11.11 shows a problem, and Figure 11.12 shows its planning graph. We start with state level S_0, which represents the problem's initial state. We follow that with action level A_0, in which we place all the actions whose preconditions are satisfied in the previous level. Each action is connected to its preconditions in S_0 and its effects in S_1, in this case introducing new literals into S_1 that were not in S_0.

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg\, Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg\, Have(Cake)$
EFFECT: $Have(Cake)$

Figure 11.11 The "have cake and eat cake too" problem.

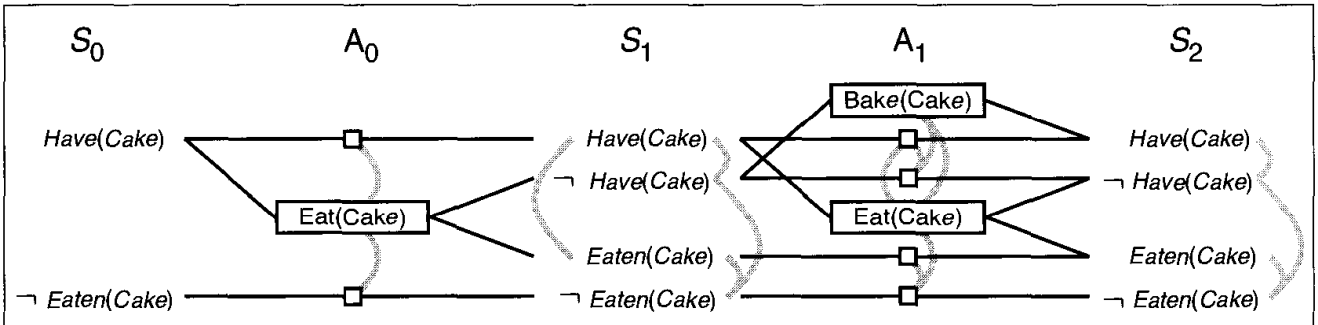


Figure 11.12 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions) and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines.

The planning graph needs a way to represent inaction as well as action. That is, it needs the equivalent of the frame axioms in situation calculus that allow a literal to remain true from one situation to the next if no action alters it. In a planning graph this is done with a set of **persistence actions**. For every positive and negative literal C, we add to the problem a persistence action with precondition C and effect C. Figure 11.12 shows one "real" action, $Eat(Cake)$ in A_0, along with two persistence actions drawn as small square boxes.

PERSISTENCE ACTIONS

Level A_0 contains all the actions that *could* occur in state S_0, but just as importantly it records conflicts between actions that would prevent them from occurring together. The gray lines in Figure 11.12 indicate **mutual exclusion** (or **mutex**) links. For example, $Eat(Cake)$ is mutually exclusive with the persistence of either $Have(Cake)$ or $\neg Eaten(Cake)$. We shall see shortly how mutex links are computed.

MUTUAL EXCLUSION
MUTEX

Level S_1 contains all the literals that could result from picking any subset of the actions in A_0. It also contains mutex links (gray lines) indicating literals that could not appear together, regardless of the choice of actions. For example, $Have(Cake)$ and $Eaten(Cake)$ are mutex: depending on the choice of actions in A_0, one or the other, but not both, could be the result. In other words, S_1 represents multiple states, just as regression state-space search does, and the mutex links are constraints that define the set of possible states.

We continue in this way, alternating between state level S_i and action level A_i until we reach a level where two consecutive levels are identical. At this point, we say that the graph

LEVELED OFF

has **leveled off**. Every subsequent level will be identical, so further expansion is unnecessary.

What we end up with is a structure where every A_i level contains all the actions that are applicable in S_i, along with constraints saying which pairs of actions cannot both be executed. Every S_i level contains all the literals that could result from any possible choice of actions in A_{i-1}, along with constraints saying which pairs of literals are not possible. It is important to note that the process of constructing the planning graph does *not* require choosing among actions, which would entail combinatorial search. Instead, it just records the impossibility of certain choices using mutex links. The complexity of constructing the planning graph is a low-order polynomial in the number of actions and literals, whereas the state space is exponential in the number of literals.

We now define mutex links for both actions and literals. A mutex relation holds between two *actions* at a given level if any of the following three conditions holds:

- *Inconsistent effects:* one action negates an effect of the other. For example $Eat(Cake)$ and the persistence of $Have(Cake)$ have inconsistent effects because they disagree on the effect $Have(Cake)$.
- *Interference:* one of the effects of one action is the negation of a precondition of the other. For example $Eat(Cake)$ interferes with the persistence of $Have(Cake)$ by negating its precondition.
- *Competing needs:* one of the preconditions of one action is mutually exclusive with a precondition of the other. For example, $Bake(Cake)$ and $Eat(Cake)$ are mutex because they compete on the value of the $Have(Cake)$ precondition.

A mutex relation holds between two *literals* at the same level if one is the negation of the other or if each possible pair of actions that could achieve the two literals is mutually exclusive. This condition is called *inconsistent support*. For example, $Have(Cake)$ and $Eaten(Cake)$ are mutex in S_1 because the only way of achieving $Have(Cake)$, the persistence action, is mutex with the only way of achieving $Eaten(Cake)$, namely $Eat(Cake)$. In S_2 the two literals are not mutex because there are new ways of achieving them, such as $Bake(Cake)$ and the persistence of $Eaten(Cake)$, that are not mutex.

Planning graphs for heuristic estimation

A planning graph, once constructed, is a rich source of information about the problem. For example, *a literal that does not appear in the final level of the graph cannot be achieved by any plan.* This observation can be used in backward search as follows: any state containing an unachievable literal has a cost $h(n) = \infty$. Similarly, in partial-order planning, any plan with an unachievable open condition has $h(n) = \infty$.

This idea can be made more general. We can estimate the cost of achieving any goal literal as the level at which it first appears in the planning graph. We will call this the **level cost** of the goal. In Figure 11.12, $Have(Cake)$ has level cost 0 and $Eaten(Cake)$ has level cost 1. It is easy to show (Exercise 11.9) that these estimates are admissible for the individual goals. The estimate might not be very good, however, because planning graphs allow several actions at each level whereas the heuristic counts just the level and not the number of actions.

LEVEL COST

SERIAL PLANNING GRAPH

For this reason, it is common to use a **serial planning graph** for computing heuristics. A

serial graph insists that only one action can actually occur at any given time step; this is done by adding mutex links between every pair of actions except persistence actions. Level costs extracted from serial graphs are often quite reasonable estimates of actual costs.

To estimate the cost of a conjunction of goals, there are three simple approaches. The
MAX-LEVEL **max-level** heuristic simply takes the maximum level cost of any of the goals; this is ad-
LEVEL SUM missible, but not necessarily very accurate. The **level sum** heuristic, following the subgoal independence assumption, returns the sum of the level costs of the goals; this is inadmissible but works very well in practice for problems that are largely decomposable. It is much more accurate than the number-of-unsatisfied-goals heuristic from Section 11.2. For our problem, the heuristic estimate for the conjunctive goal $Have(Cake) \wedge Eaten(Cake)$ will be $0+1 = 1$, whereas the correct answer is 2. Moreover, if we eliminated the $Bake(Cake)$ action, the es-
SET-LEVEL timate would still be 1, but the conjunctive goal would be impossible. Finally, the **set-level** heuristic finds the level at which all the literals in the conjunctive goal appear in the planning graph without any pair of them being mutually exclusive. This heuristic gives the correct values of 2 for our original problem and infinity for the problem without $Bake(Cake)$. It dominates the max-level heuristic and works extremely well on tasks in which there is a good deal of interaction among subplans.

As a tool for generating accurate heuristics, we can view the planning graph as a relaxed problem that is efficiently soluble. To understand the nature of the relaxed problem, we need to understand exactly what it means for a literal g to appear at level S_i in the planning graph. Ideally, we would like it to be a guarantee that there exists a plan with i action levels that achieves g, and also that if g does not appear that there is no such plan. Unfortunately, making that guarantee is as difficult as solving the original planning problem. So the planning graph makes the second half of the guarantee (if g does not appear, there is no plan), but if g does appear, then all the planning graph promises is that there is a plan that *possibly* achieves g and has no "obvious" flaws. An obvious flaw is defined as a flaw that can be detected by considering two actions or two literals at a time—in other words, by looking at the mutex relations. There could be more subtle flaws involving three, four, or more actions, but experience has shown that it is not worth the computational effort to keep track of these possible flaws. This is similar to the lesson learned from constraint satisfaction problems that it is often worthwhile to compute 2-consistency before searching for a solution, but less often worthwhile to compute 3-consistency or higher. (See Section 5.2.)

The GRAPHPLAN algorithm

This subsection shows how to extract a plan directly from the planning graph, rather than just using the graph to provide a heuristic. The GRAPHPLAN algorithm (Figure 11.13) has two main steps, which alternate within a loop. First, it checks whether all the goal literals are present in the current level with no mutex links between any pair of them. If this is the case, then a solution *might* exist within the current graph, so the algorithm tries to extract that solution. Otherwise, it expands the graph by adding the actions for the current level and the state literals for the next level. The process continues until either a solution is found or it is learned that no solution exists.

```
function GRAPHPLAN(problem) returns solution or failure

  graph ← INITIAL-PLANNING-GRAPH(problem)
  goals ← GOALS[problem]
  loop do
    if goals all non-mutex in last level of graph then do
      solution ← EXTRACT-SOLUTION(graph, goals, LENGTH(graph))
      if solution ≠ failure then return solution
      else if NO-SOLUTION-POSSIBLE(graph) then return failure
    graph ← EXPAND-GRAPH(graph, problem)
```

Figure 11.13 The GRAPHPLAN algorithm. GRAPHPLAN alternates between a solution extraction step and a graph expansion step. EXTRACT-SOLUTION looks for whether a plan can be found, starting at the end and searching backwards. EXPAND-GRAPH adds the actions for the current level and the state literals for the next level.

Let us now trace the operation of GRAPHPLAN on the spare tire problem from Section 11.1. The entire graph is shown in Figure 11.14. The first line of GRAPHPLAN initializes the planning graph to a one-level (S_0) graph consisting of the five literals from the initial state. The goal literal $At(Spare, Axle)$ is not present in S_0, so we need not call EXTRACT-SOLUTION—we are certain that there is no solution yet. Instead, EXPAND-GRAPH adds the three actions whose preconditions exist at level S_0 (i.e., all the actions except $PutOn(Spare, Axle)$), along with persistence actions for all the literals in S_0. The effects of the actions are added at level S_1. EXPAND-GRAPH then looks for mutex relations and adds them to the graph.

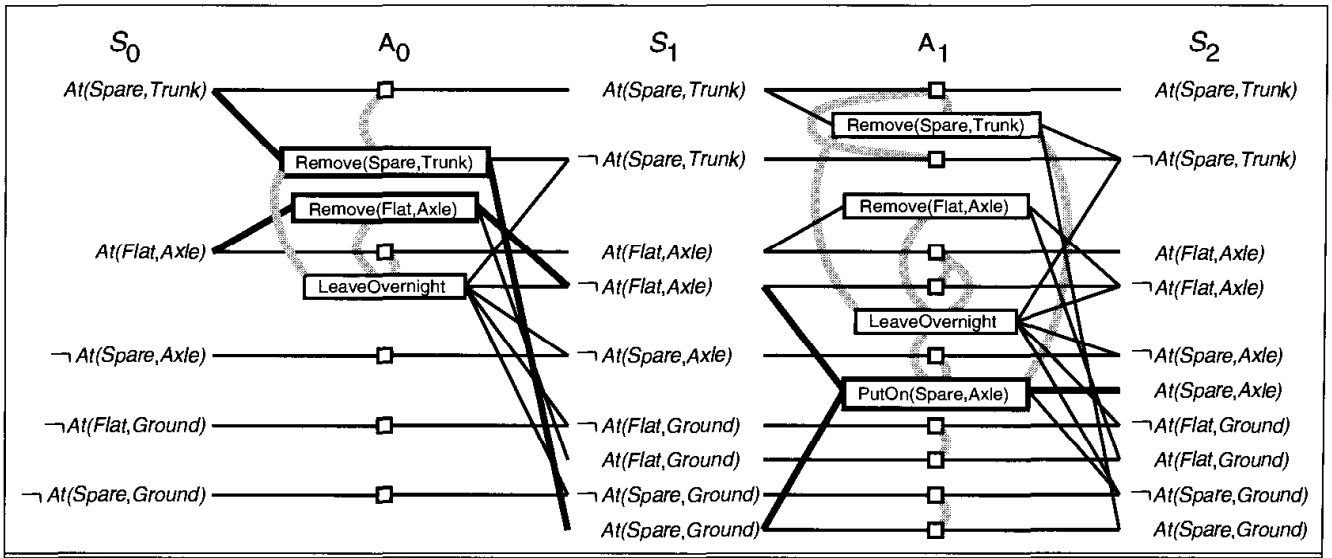


Figure 11.14 The planning graph for the spare tire problem after expansion to level S_2. Mutex links are shown as gray lines. Only some representative mutexes are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.

$At(Spare, Axle)$ is still not present in S_1, so again we do not call EXTRACT-SOLUTION. The call to EXPAND-GRAPH gives us the planning graph shown in Figure 11.14. Now that we have the full complement of actions, it is worthwhile to look at some of the examples of mutex relations and their causes:

- *Inconsistent effects:* $Remove(Spare, Trunk)$ is mutex with $LeaveOvernight$ because one has the effect $At(Spare, Ground)$ and the other has its negation.
- *Interference:* $Remove(Flat, Axle)$ is mutex with $LeaveOvernight$ because one has the precondition $At(Flat, Axle)$ and the other has its negation as an effect.
- *Competing needs:* $PutOn(Spare, Axle)$ is mutex with $Remove(Flat, Axle)$ because one has $At(Flat, Axle)$ as a precondition and the other has its negation.
- *Inconsistent support:* $At(Spare, Axle)$ is mutex with $At(Flat, Axle)$ in S_2 because the only way of achieving $At(Spare, Axle)$ is by $PutOn(Spare, Axle)$, and that is mutex with the persistence action that is the only way of achieving $At(Flat, Axle)$. Thus, the mutex relations detect the immediate conflict that arises from trying to put two objects in the same place at the same time.

This time, when we go back to the start of the loop, all the literals from the goal are present in S_2, and none of them is mutex with any other. That means that a solution might exist, and EXTRACT-SOLUTION will try to find it. In essence, EXTRACT-SOLUTION solves a Boolean CSP whose variables are the actions at each level, and the values for each variable are *in* or *out* of the plan. We can use standard CSP algorithms for this, or we can define EXTRACT-SOLUTION as a search problem, where each state in the search contains a pointer to a level in the planning graph and a set of unsatisfied goals. We define this search problem as follows:

- The initial state is the last level of the planning graph, S_n, along with the set of goals from the planning problem.
- The actions available in a state at level S_i are to select any conflict-free subset of the actions in A_{i-1} whose effects cover the goals in the state. The resulting state has level S_{i-1} and has as its set of goals the preconditions for the selected set of actions. By "conflict-free," we mean a set of actions such that no two of them are mutex, and no two of their preconditions are mutex.
- The goal is to reach a state at level S_0 such that all the goals are satisfied.
- The cost of each action is 1.

For this particular problem, we start at S_2 with the goal $At(Spare, Axle)$. The only choice we have for achieving the goal set is $PutOn(Spare, Axle)$. That brings us to a search state at S_1 with goals $At(Spare, Ground)$ and $\neg At(Flat, Axle)$. The former can be achieved only by $Remove(Spare, Trunk)$, and the latter by either $Remove(Flat, Axle)$ or $LeaveOvernight$. But $LeaveOvernight$ is mutex with $Remove(Spare, Trunk)$, so the only solution is to choose $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$. That brings us to a search state at S_0 with the goals $At(Spare, Trunk)$ and $At(Flat, Axle)$. Both of these are present in the state, so we have a solution: the actions $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$ in level A_0, followed by $PutOn(Spare, Axle)$ in A_1.

We know that planning is PSPACE-complete and that constructing the planning graph takes polynomial time, so it must be the case that solution extraction is intractable in the worst case. Therefore, we will need some heuristic guidance for choosing among actions during the backward search. One approach that works well in practice is a greedy algorithm based on the level cost of the literals. For any set of goals, we proceed in the following order:

1. Pick first the literal with the highest level cost.
2. To achieve that literal, choose the action with the easiest preconditions first. That is, choose an action such that the sum (or maximum) of the level costs of its preconditions is smallest.

Termination of GRAPHPLAN

So far, we have skated over the question of termination. If a problem has no solution, can we be sure that GRAPHPLAN will not loop forever, extending the planning graph at each iteration? The answer is yes, but the full proof is beyond the scope of this book. Here, we outline just the main ideas, particularly the ones that shed light on planning graphs in general.

The first step is to notice that certain properties of planning graphs are monotonically increasing or decreasing. "X increases monotonically" means that the set of Xs at level $i + 1$ is a superset (not necessarily proper) of the set at level i. The properties are as follows:

- *Literals increase monotonically:* Once a literal appears at a given level, it will appear at all subsequent levels. This is because of the persistence actions; once a literal shows up, persistence actions cause it to stay forever.
- *Actions increase monotonically:* Once an action appears at a given level, it will appear at all subsequent levels. This is a consequence of literals' increasing; if the preconditions of an action appear at one level, they will appear at subsequent levels, and thus so will the action.
- *Mutexes decrease monotonically:* If two actions are mutex at a given level A_i, then they will also be mutex for all *previous* levels at which they both appear. The same holds for mutexes between literals. It might not always appear that way in the figures, because the figures have a simplification: they display neither literals that cannot hold at level S_i nor actions that cannot be executed at level A_i. We can see that "mutexes decrease monotonically" is true if you consider that these invisible literals and actions are mutex with everything.

  The proof is a little complex, but can be handled by cases: if actions A and B are mutex at level A_i, it must be because of one of the three types of mutex. The first two, inconsistent effects and interference, are properties of the actions themselves, so if the actions are mutex at A_i, they will be mutex at every level. The third case, competing needs, depends on conditions at level S_i: that level must contain a precondition of A that is mutex with a precondition of B. Now, these two preconditions can be mutex if they are negations of each other (in which case they would be mutex in every level) or if all actions for achieving one are mutex with all actions for achieving the other. But we already know that the available actions are increasing monotonically, so by induction, the mutexes must be decreasing.

Because the actions and literals increase and the mutexes decrease, and because there are only a finite number of actions and literals, every planning graph will eventually level off—all subsequent levels will be identical. Once a graph has leveled off, if it is missing one of the goals of the problem, or if two of the goals are mutex, then the problem can never be solved, and we can stop the GRAPHPLAN algorithm and return failure. If the graph levels off with all goals present and nonmutex, but EXTRACT-SOLUTION fails to find a solution, then we might have to extend the graph again a finite number of times, but eventually we can stop. This aspect of termination is more complex and is not covered here.

11.5 PLANNING WITH PROPOSITIONAL LOGIC

We saw in Chapter 10 that planning can be done by proving a theorem in situation calculus. That theorem says that, given the initial state and the successor-state axioms that describe the effects of actions, the goal will be true in a situation that results from a certain action sequence. As early as 1969, this approach was thought to be too inefficient for finding interesting plans. Recent developments in efficient reasoning algorithms for propositional logic (see Chapter 7) have generated renewed interest in planning as logical reasoning.

The approach we take in this section is based on testing the **satisfiability** of a logical sentence rather than on proving a theorem. We will be finding models of propositional sentences that look like this:

$$\textit{initial state} \wedge \textit{all possible action descriptions} \wedge \textit{goal} \ .$$

The sentence will contain proposition symbols corresponding to every possible action occurrence; a model that satisfies the sentence will assign *true* to the actions that are part of a correct plan and *false* to the others. An assignment that corresponds to an incorrect plan will not be a model, because it will be inconsistent with the assertion that the goal is true. If the planning problem is unsolvable, then the sentence will be unsatisfiable.

Describing planning problems in propositional logic

The process we will follow to translate STRIPS problems into propositional logic is a textbook example (so to speak) of the knowledge representation cycle: We will begin with what seems to be a reasonable set of axioms, we will find that these axioms allow for spurious unintended models, and we will write more axioms.

Let us begin with a very simple air transport problem. In the initial state (time 0), plane P_1 is at SFO and plane P_2 is at JFK. The goal is to have P_1 at JFK and P_2 at SFO; that is, the planes are to change places. First, we will need distinct proposition symbols for assertions about each time step. We will use superscripts to denote the time step, as in Chapter 7. Thus, the initial state will be written as

$$At(P_1, SFO)^0 \wedge At(P_2, JFK)^0 \ .$$

(Remember that $At(P_1, SFO)^0$ is an atomic symbol.) Because propositional logic has no closed-world assumption, we must also specify the propositions that are *not* true in the initial

state. If some propositions are unknown in the initial state, then they can be left unspecified (the **open world assumption**). In this example we specify:

$$\neg At(P_1, JFK)^0 \wedge \neg At(P_2, SFO)^0 .$$

The goal itself must be associated with a particular time step. Since we do not know *a priori* how many steps it takes to achieve the goal, we can try asserting that the goal is true in the initial state, time $T = 0$. That is, we assert $At(P_1, JFK)^0 \wedge At(P_2, SFO)^0$. If that fails, we try again with $T = 1$, and so on until we reach the minimum feasible plan length. For each value of T, the knowledge base will include only sentences covering the time steps from 0 up to T. To ensure termination, an arbitrary upper limit, $T_{\max}$, is imposed. This algorithm is shown in Figure 11.15. An alternative approach that avoids multiple solution attempts is discussed in Exercise 11.17.

```
function SATPLAN(problem, T_max) returns solution or failure
  inputs: problem, a planning problem
          T_max, an upper limit for plan length

  for T = 0 to T_max do
    cnf, mapping ← TRANSLATE-TO-SAT(problem, T)
    assignment ← SAT-SOLVER(cnf)
    if assignment is not null then
      return EXTRACT-SOLUTION(assignment, mapping)
  return failure
```

Figure 11.15 The SATPLAN algorithm. The planning problem is translated into a CNF sentence in which the goal is asserted to hold at a fixed time step T and axioms are included for each time step up to T. (Details of the translation are given in the text.) If the satisfiability algorithm finds a model, then a plan is extracted by looking at those proposition symbols that refer to actions and are assigned *true* in the model. If no model exists, then the process is repeated with the goal moved one step later.

The next issue is how to encode action descriptions in propositional logic. The most straightforward approach is to have one proposition symbol for each action occurrence; for example, $Fly(P_1, SFO, JFK)^0$ is true if plane P_1 flies from SFO to JFK at time 0. As in Chapter 7, we write propositional versions of the successor-state axioms developed for the situation calculus in Chapter 10. For example, we have

$$\begin{aligned} At(P_1, JFK)^1 \Leftrightarrow\ & (At(P_1, JFK)^0 \wedge \neg(Fly(P_1, JFK, SFO)^0 \wedge At(P_1, JFK)^0)) \\ & \vee (Fly(P_1, SFO, JFK)^0 \wedge At(P_1, SFO)^0) . \end{aligned} \tag{11.1}$$

That is, plane P_1 will be at JFK at time 1 if it was at JFK at time 0 and didn't fly away, or it was at SFO at time 0 and flew to JFK. We need one such axiom for each plane, airport, and time step. Moreover, each additional airport adds another way to travel to or from a given airport and hence adds more disjuncts to the right-hand side of each axiom.

With these axioms in place, we can run the satisfiability algorithm to find a plan. There ought to be a plan that achieves the goal at time $T = 1$, namely, the plan in which the two

planes swap places. Now, suppose the KB is

$$\textit{initial state} \wedge \textit{successor-state axioms} \wedge \textit{goal}^1 \ , \tag{11.2}$$

which asserts that the goal is true at time $T = 1$. You can check that the assignment in which

$$Fly(P_1, SFO, JFK)^0 \text{ and } Fly(P_2, JFK, SFO)^0$$

are true and all other action symbols are false is a model of the KB. So far, so good. Are there other possible models that the satisfiability algorithm might return? Indeed, yes. Are all these other models satisfactory plans? Alas, no. Consider the rather silly plan specified by the action symbols

$$Fly(P_1, SFO, JFK)^0 \text{ and } Fly(P_1, JFK, SFO)^0 \text{ and } Fly(P_2, JFK, SFO)^0 \ .$$

This plan is silly because plane P_1 starts at SFO, so the action $Fly(P_1, JFK, SFO)^0$ is infeasible. Nonetheless, the plan *is* a model of the sentence in Equation (11.2)! That is, it is consistent with everything we have said so far about the problem. To understand why, we need to look more carefully at what the successor-state axioms (such as Equation (11.1)) say about actions whose preconditions are not satisfied. The axioms *do* predict correctly that nothing will happen when such an action is executed (see Exercise 11.15), but they do *not* say that the action cannot be executed! To avoid generating plans with illegal actions, we must add **precondition axioms** stating that an action occurrence requires the preconditions to be satisfied.[6] For example, we need

PRECONDITION AXIOMS

$$Fly(P_1, JFK, SFO)^0 \Rightarrow At(P_1, JFK)^0 \ .$$

Because $At(P_1, JFK)^0$ is stated to be false in the initial state, this axiom ensures that every model also has $Fly(P_1, JFK, SFO)^0$ set to false. With the addition of precondition axioms, there is exactly one model that satisfies all of the axioms when the goal is to be achieved at time 1, namely the model in which plane P_1 flies to JFK and plane P_2 flies to SFO. Notice that this solution has two parallel actions, just as with GRAPHPLAN or POP.

More surprises emerge when we add a third airport, LAX. Now, each plane has two actions that are legal in each state. When we run the satisfiability algorithm, we find that a model with $Fly(P_1, SFO, JFK)^0$ and $Fly(P_2, JFK, SFO)^0$ and $Fly(P_2, JFK, LAX)^0$ satisfies all the axioms. That is, the successor-state and precondition axioms allow a plane to fly to two destinations at once! The preconditions for the two flights by P_2 are satisfied in the initial state; the successor-state axioms say that P_2 will be at SFO *and* LAX at time 1; so the goal is satisfied. Clearly, we must add more axioms to eliminate these spurious solutions. One approach is to add **action exclusion axioms** that prevent simultaneous actions. For example, we can insist on complete exclusion by adding all possible axioms of the form

ACTION EXCLUSION AXIOMS

$$\neg(Fly(P_2, JFK, SFO)^0 \wedge Fly(P_2, JFK, LAX)^0) \ .$$

These axioms ensure that no two actions can occur at the same time. They eliminate all spurious plans, but also force every plan to be totally ordered. This loses the flexibility of partially ordered plans; also, by increasing the number of time steps in the plan, computation time may be lengthened.

[6] Notice that the addition of precondition axioms means that we need not include preconditions for actions in the successor-state axioms.

Instead of complete exclusion, we can require only partial exclusion—that is, rule out simultaneous actions only if they interfere with each other. The conditions are the same as those for mutex actions: two actions cannot occur simultaneously if one negates a precondition or effect of the other. For example, $Fly(P_2, JFK, SFO)^0$ and $Fly(P_2, JFK, LAX)^0$ cannot both occur, because each negates the precondition of the other; on the other hand, $Fly(P_1, SFO, JFK)^0$ and $Fly(P_2, JFK, SFO)^0$ can occur together because the two planes do not interfere. Partial exclusion eliminates spurious plans without forcing a total ordering.

Exclusion axioms sometimes seem a rather blunt instrument. Instead of saying that a plane cannot fly to two airports at the same time, we might simply insist that no object can be in two places at once:

$$\forall p, x, y, t \;\; x \neq y \;\Rightarrow\; \neg(At(p, x)^t \wedge At(p, y)^t) \; .$$

STATE CONSTRAINTS

This fact, combined with the successor-state axioms, *implies* that a plane cannot fly to two airports at the same time. Facts such as this are called **state constraints**. In propositional logic, of course, we have to write out all the ground instances of each state constraint. For the airport problem, the state constraint suffices to rule out all spurious plans. State constraints are often much more compact than action exclusion axioms, but they are not always easy to derive from the original STRIPS description of a problem.

To summarize, planning as satisfiability involves finding models for a sentence containing the initial state, the goal, the successor-state axioms, the precondition axioms, and either the action exclusion axioms or the state constraints. It can be shown that this collection of axioms is sufficient, in the sense that there are no longer any spurious "solutions." Any model satisfying the propositional sentence will be a valid plan for the original problem—that is, every linearization of the plan is a legal sequence of actions that reaches the goal.

Complexity of propositional encodings

The principal drawback of the propositional approach is the sheer size of the propositional knowledge base that is generated from the original planning problem. For example, the action schema $Fly(p, a_1, a_2)$ becomes $T \times |Planes| \times |Airports|^2$ different proposition symbols. In general, the total number of action symbols is bounded by $T \times |Act| \times |O|^P$, where $|Act|$ is the number of action schemata, $|O|$ is the number of objects in the domain, and P is the maximum arity (number of arguments) of any action schema. The number of clauses is larger still. For example, with 10 time steps, 12 planes, and 30 airports, the complete action exclusion axiom has 583 million clauses.

Because the number of action symbols is exponential in the arity of the action schema, one answer might be to try to reduce the arity. We can do this by borrowing an idea from semantic networks (Chapter 10). Semantic networks use only binary predicates; predicates with more arguments are reduced to a set of binary predicates that describe each argument separately. Applying this idea to an action symbol such as $Fly(P_1, SFO, JFK)^0$, we obtain three new symbols:

$Fly_1(P_1)^0$: plane P_1 flew at time 0
$Fly_2(SFO)^0$: the origin of the flight was SFO
$Fly_3(JFK)^0$: the destination of the flight was JFK .

SYMBOL SPLITTING

This process, called **symbol splitting**, eliminates the need for an exponential number of symbols. Now we only need $T \times |Act| \times P \times |O|$.

Symbol splitting by itself can reduce the number of symbols, but does not automatically reduce the number of axioms in the KB. That is, if each action symbol in each clause were simply replaced by a conjunction of three symbols, then the total size of the KB would remain roughly the same. Symbol splitting actually does reduce the size of the KB because some of the split symbols will be irrelevant to certain axioms and can be omitted. For example, consider the successor-state axiom in Equation (11.1), modified to include LAX and to omit action preconditions (which will be covered by separate precondition axioms):

$$At(P_1, JFK)^1 \Leftrightarrow (At(P_1, JFK)^0 \wedge \neg Fly(P_1, JFK, SFO)^0 \wedge \neg Fly(P_1, JFK, LAX)^0) \\ \vee Fly(P_1, SFO, JFK)^0 \vee Fly(P_1, LAX, JFK)^0 .$$

The first condition says that P_1 will be at JFK if it was there at time 0 and didn't fly from JFK to any other city, no matter which one; the second says it will be there if it flew to JFK from another city, no matter which one. Using the split symbols, we can simply omit the argument whose value does not matter:

$$At(P_1, JFK)^1 \Leftrightarrow (At(P_1, JFK)^0 \wedge \neg(Fly_1(P_1)^0 \wedge Fly_2(JFK)^0)) \\ \vee (Fly_1(P_1)^0 \wedge Fly_3(JFK)^0) .$$

Notice that SFO and LAX are no longer mentioned in the axiom. More generally, the split action symbols now allow the size of each successor-state axiom to be independent of the number of airports. Similar reductions occur with the precondition axioms and action exclusion axioms (see Exercise 11.16). For the case described earlier with 10 time steps, 12 planes, and 30 airports, the complete action exclusion axiom is reduced from 583 million clauses to 9,360 clauses.

There is one drawback: the split-symbol representation does not allow for parallel actions. Consider the two parallel actions $Fly(P_1, SFO, JFK)^0$ and $Fly(P_2, JFK, SFO)^0$. Converting to the split representation, we have

$$Fly_1(P_1)^0 \wedge Fly_2(SFO)^0 \wedge Fly_3(JFK)^0 \wedge \\ Fly_1(P_2)^0 \wedge Fly_2(JFK)^0 \wedge Fly_3(SFO)^0 .$$

It is no longer possible to determine what happened! We know that P_1 and P_2 flew, but we cannot identify the origin and destination of each flight. This means that a complete action exclusion axiom must be used, with the drawbacks noted previously.

Planners based on satisfiability can handle large planning problems—for example, finding optimal 30-step solutions to blocks-world planning problems with dozens of blocks. The size of the propositional encoding and the cost of solution are highly problem-dependent, but in most cases the memory required to store the propositional axioms is the bottleneck. One interesting finding from this work has been that backtracking algorithms such as DPLL are often better at solving planning problems than local search algorithms such as WALKSAT. This is because the majority of the propositional axioms are Horn clauses, which are handled efficiently by the unit propagation technique. This observation has led to the development of hybrid algorithms combining some random search with backtracking and unit propagation.

11.6 ANALYSIS OF PLANNING APPROACHES

Planning is an area of great current interest within AI. One reason for this is that it combines the two major areas of AI we have covered so far: *search* and *logic*. That is, a planner can be seen either as a program that searches for a solution or as one that (constructively) proves the existence of a solution. The cross-fertilization of ideas from the two areas has led to both improvements in performance amounting to several orders of magnitude in the last decade and an increased use of planners in industrial applications. Unfortunately, we do not yet have a clear understanding of which techniques work best on which kinds of problems. Quite possibly, new techniques will emerge that dominate existing methods.

Planning is foremost an exercise in controlling combinatorial explosion. If there are p primitive propositions in a domain, then there are 2^p states. For complex domains, p can grow quite large. Consider that objects in the domain have properties (*Location*, *Color*, etc.) and relations (*At*, *On*, *Between*, etc.). With d objects in a domain with ternary relations, we get 2^{d^3} states. We might conclude that, in the worst case, planning is hopeless.

Against such pessimism, the divide-and-conquer approach can be a powerful weapon. In the best case—full decomposability of the problem—divide-and-conquer offers an exponential speedup. Decomposability is destroyed, however, by negative interactions between actions. Partial-order planners deal with this with causal links, a powerful representational approach, but unfortunately each conflict must be resolved with a choice (put the conflicting action before or after the link), and the choices can multiply exponentially. GRAPHPLAN avoids these choices during the graph construction phase, using mutex links to record conflicts without actually making a choice as to how to resolve them. SATPLAN represents a similar range of mutex relations, but does so by using the general CNF form rather than a specific data structure. How well this works depends on the SAT solver used.

SERIALIZABLE SUBGOALS

Sometimes it is possible to solve a problem efficiently by recognizing that negative interactions can be ruled out. We say that a problem has **serializable subgoals** if there exists an order of subgoals such that the planner can achieve them in that order, without having to undo any of the previously achieved subgoals. For example, in the blocks world, if the goal is to build a tower (e.g., A on B, which in turn is on C, which in turn is on the *Table*), then the subgoals are serializable bottom to top: if we first achieve C on *Table*, we will never have to undo it while we are achieving the other subgoals. A planner that uses the bottom-to-top trick can solve any problem in the blocks world domain without backtracking (although it might not always find the shortest plan).

As a more complex example, for the Remote Agent planner which commanded NASA's Deep Space One spacecraft, it was determined that the propositions involved in commanding a spacecraft are serializable. This is perhaps not too surprising, because a spacecraft is designed by its engineers to be as easy as possible to control (subject to other constraints). Taking advantage of the serialized ordering of goals, the Remote Agent planner was able to eliminate most of the search. This meant that it was fast enough to control the spacecraft in real time, something previously considered impossible.

There is more than one way to control combinatorial explosions. We saw in Chapter 5 that there are many techniques for controlling backtracking in constraint satisfaction problems (CSPs), such as dependency-directed backtracking. All of these techniques can be applied to planning. For example, extracting a solution from a planning graph can be formulated as a Boolean CSP whose variables state whether a given action should occur at a given time. The CSP can be solved using any of the algorithms in Chapter 5, such as min-conflicts. A closely related method, used in the BLACKBOX system, is to convert the planning graph into a CNF expression and then extract a plan by using a SAT solver. This approach seems to work better than SATPLAN, presumably because the planning graph has already eliminated many of the impossible states and actions from the problem. It also works better than GRAPHPLAN, presumably because a satisfiability search such as WALKSAT has much greater flexibility than the strict backtracking search that GRAPHPLAN uses.

There is no doubt that planners such as GRAPHPLAN, SATPLAN, and BLACKBOX have moved the field of planning forward, both by raising the level of performance of planning systems and by clarifying the representational and combinatorial issues involved. These methods are, however, inherently propositional and thus are limited in the domains they can express. (For example, logistics problems with a few dozen objects and locations can require gigabytes of storage for the corresponding CNF expressions.) It seems likely that first-order representations and algorithms will be required if further progress is to occur, although structures such as planning graphs will continue to be useful as a source of heuristics.

11.7 SUMMARY

In this chapter, we defined the problem of planning in deterministic, fully observable environments. We described the principal representations used for planning problems and several algorithmic approaches for solving them. The points to remember are:

- Planning systems are problem-solving algorithms that operate on explicit propositional (or first-order) representations of states and actions. These representations make possible the derivation of effective heuristics and the development of powerful and flexible algorithms for solving problems.
- The STRIPS language describes actions in terms of their preconditions and effects and describes the initial and goal states as conjunctions of positive literals. The ADL language relaxes some of these constraints, allowing disjunction, negation, and quantifiers.
- State-space search can operate in the forward direction (**progression**) or the backward direction (**regression**). Effective heuristics can be derived by making a subgoal independence assumption and by various relaxations of the planning problem.
- Partial-order planning (POP) algorithms explore the space of plans without committing to a totally ordered sequence of actions. They work back from the goal, adding actions to the plan to achieve each subgoal. They are particularly effective on problems amenable to a divide-and-conquer approach.

- A **planning graph** can be constructed incrementally, starting from the initial state. Each layer contains a superset of all the literals or actions that could occur at that time step and encodes mutual exclusion, or **mutex**, relations among literals or actions that cannot co-occur. Planning graphs yield useful heuristics for state-space and partial-order planners and can be used directly in the GRAPHPLAN algorithm.
- The GRAPHPLAN algorithm processes the planning graph, using a backward search to extract a plan. It allows for some partial ordering among actions.
- The SATPLAN algorithm translates a planning problem into propositional axioms and applies a satisfiability algorithm to find a model that corresponds to a valid plan. Several different propositional representations have been developed, with varying degrees of compactness and efficiency.
- Each of the major approaches to planning has its adherents, and there is as yet no consensus on which is best. Competition and cross-fertilization among the approaches have resulted in significant gains in efficiency for planning systems.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

AI planning arose from investigations into state-space search, theorem proving, and control theory and from the practical needs of robotics, scheduling, and other domains. STRIPS (Fikes and Nilsson, 1971), the first major planning system, illustrates the interaction of these influences. STRIPS was designed as the planning component of the software for the Shakey robot project at SRI. Its overall control structure was modeled on that of GPS, the General Problem Solver (Newell and Simon, 1961), a state-space search system that used means–ends analysis. STRIPS used a version of the QA3 theorem proving system (Green, 1969b) as a subroutine for establishing the truth of preconditions for actions. Lifschitz (1986) offers precise definitions and an analysis of the STRIPS language. Bylander (1992) shows simple STRIPS planning to be PSPACE-complete. Fikes and Nilsson (1993) give a historical retrospective on the STRIPS project and a survey of its relationship to more recent planning efforts.

The action representation used by STRIPS has been far more influential than its algorithmic approach. Almost all planning systems since then have used one variant or another of the STRIPS language. Unfortunately, the proliferation of variants has made comparisons needlessly difficult. With time came a better understanding of the limitations and tradeoffs among formalisms. The Action Description Language, or ADL, (Pednault, 1986) relaxed some of the restrictions in the STRIPS language and made it possible to encode more realistic problems. Nebel (2000) explores schemes for compiling ADL into STRIPS. The Problem Domain Description Language or PDDL (Ghallab *et al.*, 1998) was introduced as a computer-parsable, standardized syntax for representing STRIPS, ADL, and other languages. PDDL has been used as the standard language for the planning competitions at the AIPS conference, beginning in 1998.

Planners in the early 1970s generally worked with totally ordered action sequences. Problem decomposition was achieved by computing a subplan for each subgoal and then

LINEAR PLANNING

stringing the subplans together in some order. This approach, called **linear planning** by Sacerdoti (1975), was soon discovered to be incomplete. It cannot solve some very simple problems, such as the Sussman anomaly (see Exercise 11.11), found by Allen Brown during experimentation with the HACKER system (Sussman, 1975). A complete planner must allow for **interleaving** of actions from different subplans within a single sequence. The notion of serializable subgoals (Korf, 1987) corresponds exactly to the set of problems for which noninterleaved planners are complete.

INTERLEAVING

One solution to the interleaving problem was goal regression planning, a technique in which steps in a totally ordered plan are reordered so as to avoid conflict between subgoals. This was introduced by Waldinger (1975) and also used by Warren's (1974) WARPLAN. WARPLAN is also notable in that it was the first planner to be written in a logic programming language (Prolog) and is one of the best examples of the remarkable economy that can sometimes be gained by using logic programming: WARPLAN is only 100 lines of code, a small fraction of the size of comparable planners of the time. INTERPLAN (Tate, 1975a, 1975b) also allowed arbitrary interleaving of plan steps to overcome the Sussman anomaly and related problems.

The ideas underlying partial-order planning include the detection of conflicts (Tate, 1975a) and the protection of achieved conditions from interference (Sussman, 1975). The construction of partially ordered plans (then called **task networks**) was pioneered by the NOAH planner (Sacerdoti, 1975, 1977) and by Tate's (1975b, 1977) NONLIN system.[7]

Partial-order planning dominated the next 20 years of research, yet for much of that time, the field was not widely understood. TWEAK (Chapman, 1987) was a logical reconstruction and simplification of planning work of this time; his formulation was clear enough to allow proofs of completeness and intractability (NP-hardness and undecidability) of various formulations of the planning problem. Chapman's work led to what was arguably the first simple and readable description of a complete partial-order planner (McAllester and Rosenblitt, 1991). An implementation of McAllester and Rosenblitt's algorithm called SNLP (Soderland and Weld, 1991) was widely distributed and allowed many researchers to understand and experiment with partial-order planning for the first time. The POP algorithm described in this chapter is based on SNLP.

Weld's group also developed UCPOP (Penberthy and Weld, 1992), the first planner for problems expressed in ADL. UCPOP incorporated the number-of-unsatisfied-goals heuristic. It ran somewhat faster than SNLP, but was seldom able to find plans with more than a dozen or so steps. Although improved heuristics were developed for UCPOP (Joslin and Pollack, 1994; Gerevini and Schubert, 1996), partial-order planning fell into disrepute in the 1990s as faster methods emerged. Nguyen and Kambhampati (2001) suggest that a rehabilitation is merited: with accurate heuristics derived from a planning graph, their REPOP planner scales up much better than GRAPHPLAN and is competitive with the fastest state-space planners.

Avrim Blum and Merrick Furst (1995, 1997) revitalized the field of planning with their GRAPHPLAN system, which was orders of magnitude faster than the partial-order planners of

[7] Some confusion exists over terminology. Many authors use the term **nonlinear** to mean partially ordered. This is slightly different from Sacerdoti's original usage referring to interleaved plans.

the time. Other graph planning systems, such as IPP (Koehler *et al.*, 1997), STAN (Fox and Long, 1998) and SGP (Weld *et al.*, 1998), soon followed. A data structure closely resembling the planning graph had been developed slightly earlier by Ghallab and Laruelle (1994), whose IXTET partial-order planner used it to derive accurate heuristics to guide searches. Nguyen *et al.* (2001) give a very thorough analysis of heuristics derived from planning graphs. Our discussion of planning graphs is based partly on this work and on lecture notes by Subbarao Kambhampati. As mentioned in the chapter, a planning graph can be used in many different ways to guide the search for a solution. The winner of the 2002 AIPS planning competition, LPG (Gerevini and Serina, 2002), searched planning graphs using a local search technique inspired by WALKSAT.

Planning as satisfiability and the SATPLAN algorithm were proposed by Kautz and Selman (1992), who were inspired by the surprising success of greedy local search for satisfiability problems. (See Chapter 7.) Kautz *et al.* (1996) also investigated various forms of propositional representations for STRIPS axioms, finding that the most compact forms did not necessarily lead to the fastest solution times. A systematic analysis was carried out by Ernst *et al.* (1997), who also developed an automatic "compiler" for generating propositional representations from PDDL problems. The BLACKBOX planner, which combines ideas from GRAPHPLAN and SATPLAN, was developed by Kautz and Selman (1998).

The resurgence of interest in state-space planning was pioneered by Drew McDermott's UNPOP program (1996), which was the first to suggest a distance heuristic based on a relaxed problem with delete lists ignored. The name UNPOP was a reaction to the overwhelming concentration on partial-order planning at the time; McDermott suspected that other approaches were not getting the attention they deserved. Bonet and Geffner's Heuristic Search Planner (HSP) and its later derivatives (Bonet and Geffner, 1999) were the first to make state-space search practical for large planning problems. The most successful state-space searcher to date is Hoffmann's (2000) FASTFORWARD or FF, winner of the AIPS 2000 planning competition. FF uses a simplified planning graph heuristic with a very fast search algorithm that combines forward and local search in a novel way.

BINARY DECISION DIAGRAMS

Most recently, there has been interest in the representation of plans as **binary decision diagrams**, a compact description of finite automata widely studied in the hardware verification community (Clarke and Grumberg, 1987; McMillan, 1993). There are techniques for proving properties of binary decision diagrams, including the property of being a solution to a planning problem. Cimatti *et al.* (1998) present a planner based on this approach. Other representations have also been used; for example, Vossen *et al.* (2001) survey the use of integer programming for planning.

The jury is still out, but there are now some interesting comparisons of the various approaches to planning. Helmert (2001) analyzes several classes of planning problems, and shows that constraint-based approaches, such as GRAPHPLAN and SATPLAN are best for NP-hard domains, while search-based approaches do better in domains where feasible solutions can be found without backtracking. GRAPHPLAN and SATPLAN have trouble in domains with many objects, because that means they must create many actions. In some cases the problem can be delayed or avoided by generating the propositionalized actions dynamically, only as needed, rather than instantiating them all before the search begins.

Weld (1994, 1999) provides two excellent surveys of modern planning algorithms. It is interesting to see the change in the five years between the two surveys: the first concentrates on partial-order planning, and the second introduces GRAPHPLAN and SATPLAN. *Readings in Planning* (Allen *et al.*, 1990) is a comprehensive anthology of many of the best earlier articles in the field, including several good surveys. Yang (1997) provides a book-length overview of partial-order planning techniques.

Planning research has been central to AI since its inception, and papers on planning are a staple of mainstream AI journals and conferences. There are also specialized conferences such as the International Conference on AI Planning Systems (AIPS), the International Workshop on Planning and Scheduling for Space, and the European Conference on Planning.

EXERCISES

11.1 Describe the differences and similarities between problem solving and planning.

11.2 Given the axioms from Figure 11.2, what are all the applicable concrete instances of $Fly(p, from, to)$ in the state described by

$$At(P_1, JFK) \wedge At(P_2, SFO) \wedge Plane(P_1) \wedge Plane(P_2) \\ \wedge Airport(JFK) \wedge Airport(SFO) \ ?$$

11.3 Let us consider how we might translate a set of STRIPS schemata into the successor-state axioms of situation calculus. (See Chapter 10.)

- Consider the schema for $Fly(p, from, to)$. Write a logical definition for the predicate $FlyPrecond(p, from, to, s)$, which is true if the preconditions for $Fly(p, from, to)$ are satisfied in situation s.
- Next, assuming that $Fly(p, from, to)$ is the only action schema available to the agent, write down a successor-state axiom for $At(p, x, s)$ that captures the same information as the action schema.
- Now suppose there is an additional method of travel: $Teleport(p, from, to)$. It has the additional precondition $\neg\, Warped(p)$ and the additional effect $Warped(p)$. Explain how the situation calculus knowledge base must be modified.
- Finally, develop a general and precisely specified procedure for carrying out the translation from a set of STRIPS schemata to a set of successor-state axioms.

11.4 The monkey-and-bananas problem is faced by a monkey in a laboratory with some bananas hanging out of reach from the ceiling. A box is available that will enable the monkey to reach the bananas if he climbs on it. Initially, the monkey is at A, the bananas at B, and the box at C. The monkey and box have height Low, but if the monkey climbs onto the box he will have height $High$, the same as the bananas. The actions available to the monkey include Go from one place to another, $Push$ an object from one place to another, $ClimbUp$ onto or

ClimbDown from an object, and *Grasp* or *Ungrasp* an object. Grasping results in holding the object if the monkey and object are in the same place at the same height.

a. Write down the initial state description.

b. Write down STRIPS-style definitions of the six actions.

c. Suppose the monkey wants to fool the scientists, who are off to tea, by grabbing the bananas, but leaving the box in its original place. Write this as a general goal (i.e., not assuming that the box is necessarily at C) in the language of situation calculus. Can this goal be solved by a STRIPS-style system?

d. Your axiom for pushing is probably incorrect, because if the object is too heavy, its position will remain the same when the *Push* operator is applied. Is this an example of the ramification problem or the qualification problem? Fix your problem description to account for heavy objects.

11.5 Explain why the process for generating predecessors in backward search does not need to add the literals that are negative effects of the action.

11.6 Explain why dropping negative effects from every action schema in a STRIPS problem results in a relaxed problem.

11.7 Examine the definition of **bidirectional search** in Chapter 3.

a. Would bidirectional state-space search be a good idea for planning?

b. What about bidirectional search in the space of partial-order plans?

c. Devise a version of partial-order planning in which an action can be added to a plan if its preconditions can be achieved by the effects of actions already in the plan. Explain how to deal with conflicts and ordering constraints. Is the algorithm essentially identical to forward state-space search?

d. Consider a partial-order planner that combines the method in part (c) with the standard method of adding actions to achieve open conditions. Would the resulting algorithm be the same as part (b)?

11.8 Construct levels 0, 1, and 2 of the planning graph for the problem in Figure 11.2.

11.9 Prove the following assertions about planning graphs:

- A literal that does not appear in the final level of the graph cannot be achieved.
- The level cost of a literal in a serial graph is no greater than the actual cost of an optimal plan for achieving it.

11.10 We contrasted forward and backward state-space search planners with partial-order planners, saying that the latter is a plan-space searcher. Explain how forward and backward state-space search can also be considered plan-space searchers, and say what the plan refinement operators are.

SUSSMAN ANOMALY

11.11 Figure 11.16 shows a blocks-world problem known as the **Sussman anomaly**. The problem was considered anomalous because the noninterleaved planners of the early 1970s could not solve it. Write a definition of the problem in STRIPS notation and solve it, either by hand or with a planning program. A noninterleaved planner is a planner that, when given two subgoals G_1 and G_2, produces either a plan for G_1 concatenated with a plan for G_2, or vice-versa. Explain why a noninterleaved planner cannot solve this problem.

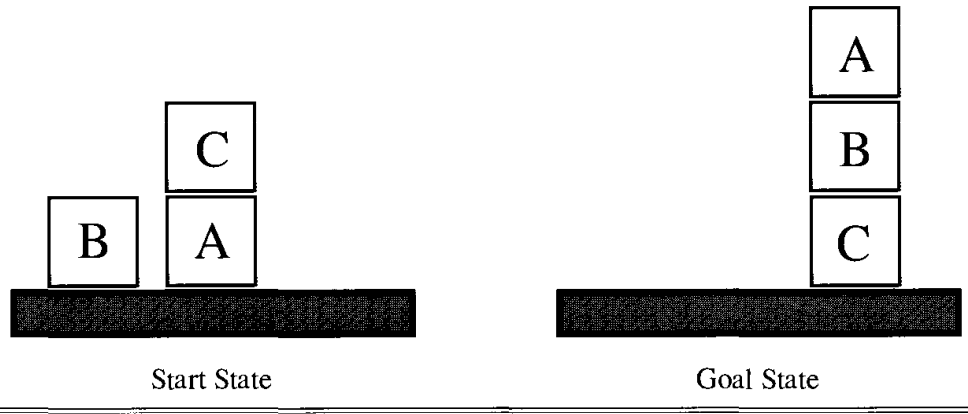


Figure 11.16 The "Sussman anomaly" blocks-world planning problem.

11.12 Consider the problem of putting on one's shoes and socks, as defined in Section 11.3. Apply GRAPHPLAN to this problem and show the solution obtained. Now add actions for putting on a coat and a hat. Show the partial order plan that is a solution, and show that there are 180 different linearizations of the partial-order plan. What is the minimum number of different planning graph solutions needed to represent all 180 linearizations?

11.13 The original STRIPS program was designed to control Shakey the robot. Figure 11.17 shows a version of Shakey's world consisting of four rooms lined up along a corridor, where each room has a door and a light switch.

The actions in Shakey's world include moving from place to place, pushing movable objects (such as boxes), climbing onto and down from rigid objects (such as boxes), and turning light switches on and off. The robot itself was never dexterous enough to climb on a box or toggle a switch, but the STRIPS planner was capable of finding and printing out plans that were beyond the robot's abilities. Shakey's six actions are the following:

- $Go(x, y)$, which requires that Shakey be at x and that x and y are locations in the same room. By convention a door between two rooms is in both of them.
- Push a box b from location x to location y within the same room: $Push(b, x, y)$. We will need the predicate Box and constants for the boxes.
- Climb onto a box: $ClimbUp(b)$; climb down from a box: $ClimbDown(b)$. We will need the predicate On and the constant $Floor$.
- Turn a light switch on: $TurnOn(s)$; turn it off: $TurnOff(s)$. To turn a light on or off, Shakey must be on top of a box at the light switch's location.

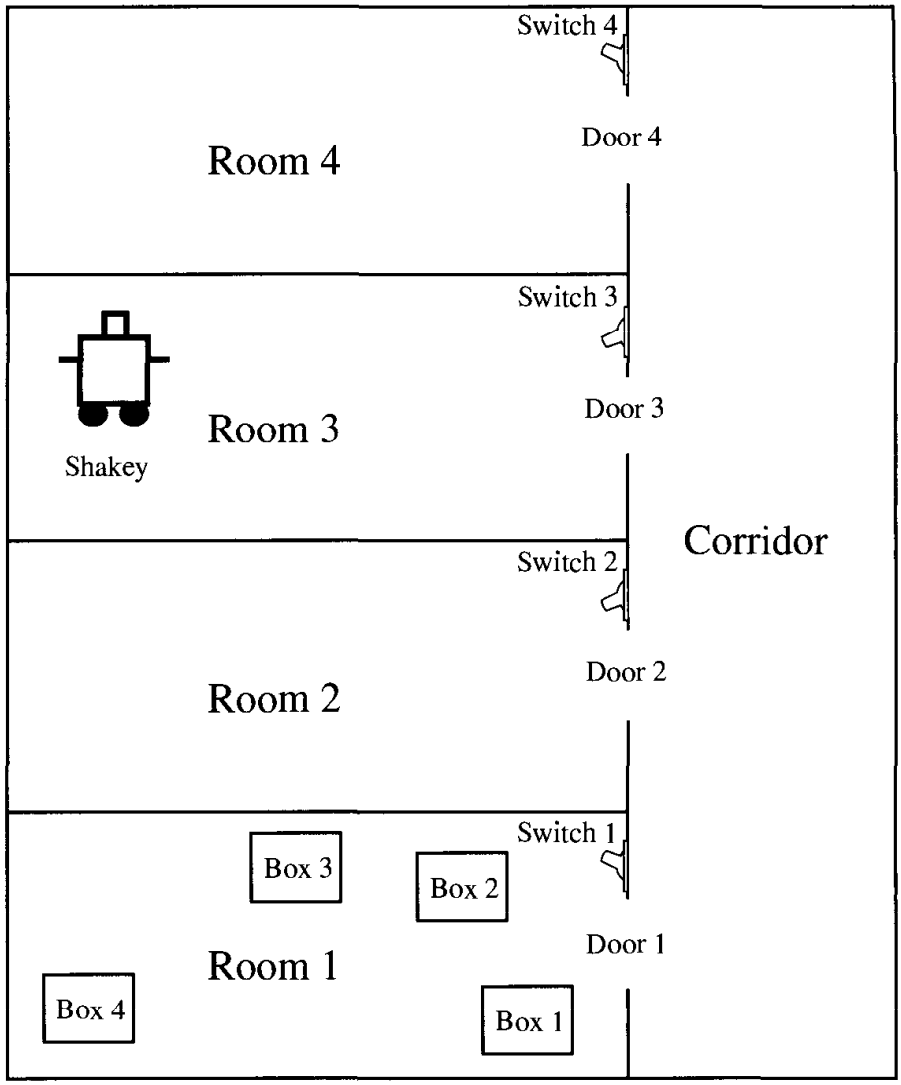


Figure 11.17 Shakey's world. Shakey can move between landmarks within a room, can pass through the door between rooms, can climb climbable objects and push pushable objects, and can flip light switches.

Describe Shakey's six actions and the initial state from Figure 11.17 in STRIPS notation. Construct a plan for Shakey to get Box_2 into $Room_2$.

11.14 We saw that planning graphs can handle only propositional actions. What if we want to use planning graphs for a problem with variables in the goal, such as $At(P_1, x) \wedge At(P_2, x)$, where x ranges over a finite domain of locations? How could you encode such a problem to work with planning graphs? (Hint: remember the *Finish* action from POP planning. What preconditions should it have?)

11.15 Up to now we have assumed that actions are only executed in the appropriate situations. Let us see what propositional successor-state axioms such as Equation (11.1) have to say about actions whose preconditions are not satisfied.

a. Show that the axioms predict that nothing will happen when an action is executed in a state where its preconditions are not satisifed.

b. Consider a plan p that contains the actions required to achieve a goal but also includes illegal actions. Is it the case that

$$\textit{initial state} \wedge \textit{successor-state axioms} \wedge p \models \textit{goal} \; ?$$

c. With first-order successor-state axioms in situation calculus (as in Chapter 10), is it possible to prove that a plan containing illegal actions will achieve the goal?

11.16 Giving examples from the airport domain, explain how symbol-splitting reduces the size of the precondition axioms and the action exclusion axioms. Derive a general formula for the size of each axiom set in terms of the number of time steps, the number of action schemata, their arities, and the number of objects.

11.17 In the SATPLAN algorithm in Figure 11.15, each call to the satisfiability algorithm asserts a goal g^T, where T ranges from 0 to $T_{\max}$. Suppose instead that the satisfiability algorithm is called only once, with the goal $g^0 \vee g^1 \vee \cdots \vee g^{T_{\max}}$.

a. Will this always return a plan if one exists with length less than or equal to $T_{\max}$?

b. Does this approach introduce any new spurious "solutions"?

c. Discuss how one might modify a satisfiability algorithm such as WALKSAT so that it finds short solutions (if they exist) when given a disjunctive goal of this form.

12 PLANNING AND ACTING IN THE REAL WORLD

In which we see how more expressive representations and more interactive agent architectures lead to planners that are useful in the real world.

The previous chapter introduced the most basic concepts, representations, and algorithms for planning. Planners that are are used in the real world for tasks such as scheduling Hubble Space Telescope observations, operating factories, and handling the logistics for military campaigns are more complex; they extend the basics in terms both of the representation language and of the way the planner interacts with the environment. This chapter shows how. Section 12.1 describes planning and scheduling with time and resource constraints, and Section 12.2 describes planning with predefined subplans. Sections 12.3 to 12.6 present a series of agent architectures designed to deal with uncertain environments. Section 12.7 shows how to plan when the environment contains other agents.

12.1 TIME, SCHEDULES, AND RESOURCES

The STRIPS representation talks about *what* actions do, but, because the representation is based on situation calculus, it cannot talk about *how long* an action takes or even about *when* an action occurs, except to say that it is before or after another action. For some domains, we would like to talk about when actions begin and end. For example, in the cargo delivery domain, we might like to know when the plane carrying some cargo will arrive, not just that it will arrive when it is done flying.

JOB SHOP SCHEDULING

Time is of the essence in the general family of applications called **job shop scheduling**. Such tasks require completing a set of jobs, each of which consists of a sequence of actions, where each action has a given duration and might require some resources. The problem is to determine a schedule that minimizes the total time required to complete all the jobs, while respecting the resource constraints.

An example of a job shop scheduling problem is given in Figure 12.1. This is a highly simplified automobile assembly problem. There are two jobs: assembling cars C_1 and C_2. Each job consists of three actions: adding the engine, adding the wheels, and inspecting the

$Init(Chassis(C_1) \wedge Chassis(C_2)$
$\wedge Engine(E_1, C_1, 30) \wedge Engine(E_2, C_2, 60)$
$\wedge Wheels(W_1, C_1, 30) \wedge Wheels(W_2, C_2, 15))$
$Goal(Done(C_1) \wedge Done(C_2))$

$Action(AddEngine(e, c),$
PRECOND:$Engine(e, c, d) \wedge Chassis(c) \wedge \neg EngineIn(c),$
EFFECT:$EngineIn(c) \wedge Duration(d))$
$Action(AddWheels(w, c),$
PRECOND:$Wheels(w, c, d) \wedge Chassis(c) \wedge EngineIn(c),$
EFFECT:$WheelsOn(c) \wedge Duration(d))$
$Action(Inspect(c),$ PRECOND:$EngineIn(c) \wedge WheelsOn(c) \wedge Chassis(c),$
EFFECT:$Done(c) \wedge Duration(10))$

Figure 12.1 A job shop scheduling problem for assembling two cars. The notation $Duration(d)$ means that an action takes d minutes to execute. $Engine(E_1, C_1, 60)$ means that E_1 is an engine that fits into chassis C_1 and takes 60 minutes to install.

results. The engine must be put in first (because having the front wheels on would inhibit access to the engine compartment) and of course the inspection must be done last.

The problem in Figure 12.1 can be solved by any of the planners we have already seen. Figure 12.2 (if you ignore the numbers) shows the solution that the partial-order planner POP would come up with. To make this a *scheduling* problem rather than a *planning* problem, we must now determine when each action should begin and end, based on the *durations* of actions as well as their ordering. The notation $Duration(d)$ in the effect of an action (where d must be bound to a number) means that the action takes d minutes to complete.

CRITICAL PATH METHOD

Given a partial ordering of actions with durations, as in Figure 12.2, we can apply the **critical path method** (CPM) to determine the possible start and end times of each action. A **path** through a partial-order plan is a linearly ordered sequence of actions beginning with *Start* and ending with *Finish*. (For example, there are two paths in the partial-order plan in Figure 12.2.)

CRITICAL PATH

The **critical path** is that path whose total duration is longest; the path is "critical" because it determines the duration of the entire plan—shortening other paths doesn't shorten the plan as a whole, but delaying the start of any action on the critical path slows down the whole plan. In the figure, the critical path is shown with bold lines. To complete the whole plan in the minimal total time, the actions on the critical path must be executed with no delay between them. Actions that are off the critical path have some leeway—a window of time in which they can be executed. The window is specified in terms of an earliest possible start time, ES, and a latest possible start time, LS. The quantity $LS - ES$ is known as the **slack** of an action. We can see in Figure 12.2 that the whole plan will take 80 minutes, that each action on the critical path has 0 slack (this will always be the case) and that each of the actions in the assembly of C_1 have a 10-minute window in which they can be started. Together the ES and LS times for all the actions constitute a **schedule** for the problem.

SLACK

SCHEDULE

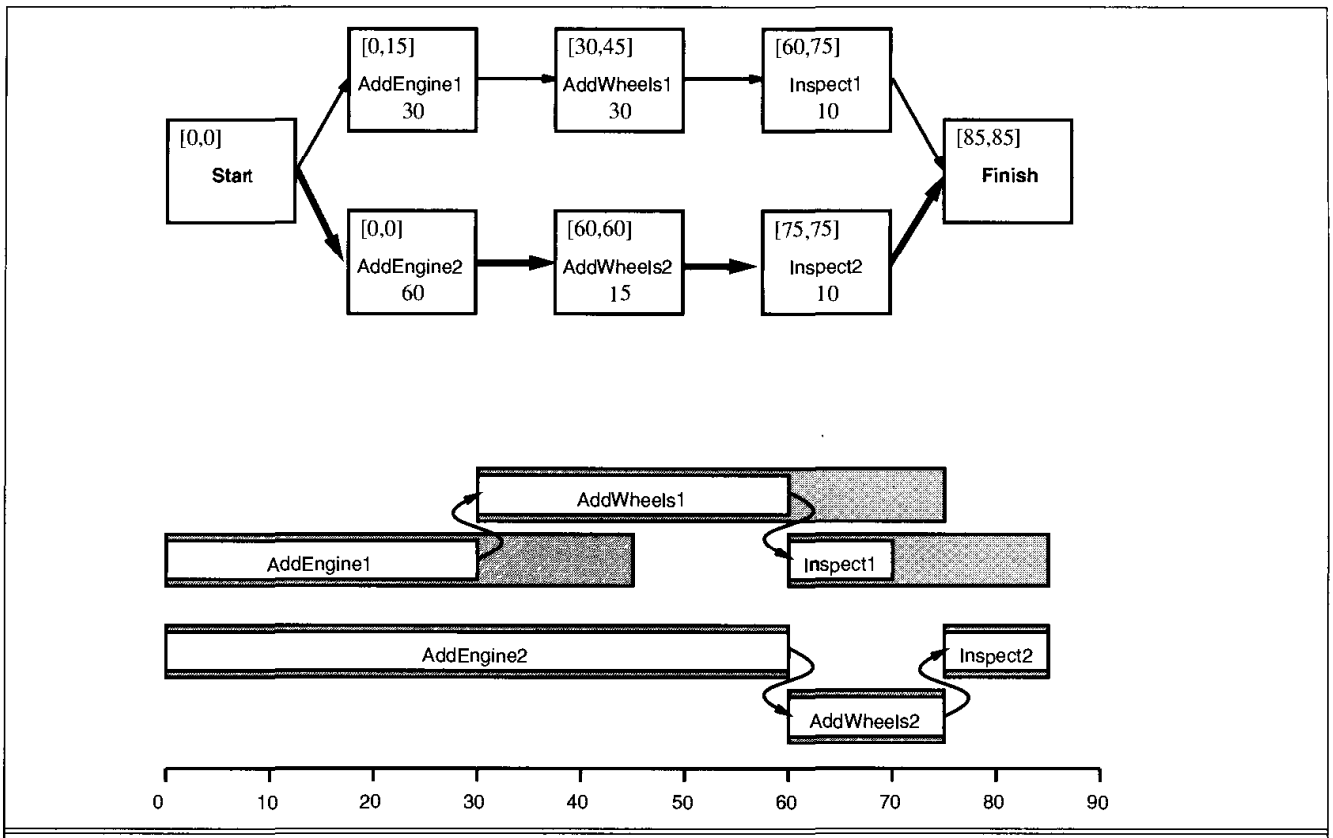


Figure 12.2 A solution to the job shop scheduling problem from Figure 12.1. At the top, the solution is given as a partial-order plan. The duration of each action is given at the bottom of each rectangle, with the earliest and latest start time listed as $[ES, LS]$ in the upper left. The difference between these two numbers is the slack of an action; actions with zero slack are on the critical path, shown with bold arrows. At the bottom of the figure, the same solution is shown as a timeline. Grey rectangles represent time intervals during which an action may be executed, provided that the ordering constraints are respected. The unoccupied portion of a gray rectangle indicates the slack.

The following formulas serve as a definition for ES and LS and also as the outline of a dynamic programming algorithm to compute them:

$$ES(Start) = 0 .$$
$$ES(B) = \max_{A \prec B} ES(A) + Duration(A) .$$
$$LS(Finish) = ES(Finish) .$$
$$LS(A) = \min_{A \prec B} LS(B) - Duration(A) .$$

The idea is that we start by assigning $ES(Start)$ to be 0. Then as soon as we get an action B such that all the actions that come immediately before B have ES values assigned, we set $ES(B)$ to be the maximum of the earliest finish times of those immediately preceding actions, where the earliest finish time of an action is defined as the earliest start time plus the duration. This process repeats until every action has been assigned an ES value. The LS values are computed in a similar manner, working backwards from the $Finish$ action. The details are left as an exercise.

The complexity of the critical path algorithm is just $O(Nb)$, where N is the number of actions and b is the maximum branching factor into or out of an action. (To see this,

note that the LS and ES computations are done once for each action, and each computation iterates over at most b other actions.) Therefore, the problem of finding a minimum-duration schedule, *given a partial ordering on the actions*, is quite easy to solve.

Scheduling with resource constraints

RESOURCES

Real scheduling problems are complicated by the presence of constraints on **resources**. For example, adding an engine to a car requires an engine hoist. If there is only one hoist, then we cannot simultaneously add engine E_1 to car C_1 and engine E_2 to car C_2; hence, the schedule shown in Figure 12.2 would be infeasible. The engine hoist is an example of a **reusable resource**—a resource that is "occupied" during the action but that becomes available again when the action is finished. Notice that reusable resources cannot be handled in our standard description of actions in terms of preconditions and effects, because the amount of resource available is unchanged after the action is completed.[1] For this reason, we augment our representation to include a field of the form RESOURCE:$R(k)$, which means that k units of resource R are required by the action. The resource requirement is both a prerequisite—the action cannot be performed if the resource is unavailable—and a *temporary* effect, in the sense that the availability of resource r is reduced by k for the duration of the action. Figure 12.3 shows how to extend the engine assembly problem to include three resources: an engine hoist for installing engines, a wheel station for putting on the wheels, and two inspectors. Figure 12.4 shows the solution with the fastest completion time, 115 minutes. This is longer than the 80 minutes required for a schedule without resource constraints. Notice that there is no time at which both inspectors are required, so we can immediately move one of our two inspectors to a more productive position.

REUSABLE RESOURCE

The representation of resources as numerical quantities, such as $Inspectors(2)$, rather than as named entities, such as $Inspector(I_1)$ and $Inspector(I_2)$, is an example of a very general technique called **aggregation**. The central idea of aggregation is to group individual objects into quantities when the objects are all indistinguishable with respect to the purpose at hand. In our assembly problem, it does not matter *which* inspector inspects the car, so there is no need to make the distinction. (The same idea works in the missionaries-and-cannibals problem in Exercise 3.9.) Aggregation is essential for reducing complexity. Consider what happens when a schedule is proposed that has 10 concurrent $Inspect$ actions but only 9 inspectors are available. With inspectors represented as quantities, a failure is detected immediately and the algorithm backtracks to try another schedule. With inspectors represented as individuals, the algorithm backtracks to try all 10! ways of assigning inspectors to $Inspect$ actions, to no avail.

AGGREGATION

Despite their advantages, resource constraints make scheduling problems more complicated by introducing additional interactions among actions. Whereas unconstrained scheduling using the critical-path method is easy, finding a resource-constrained schedule with the earliest possible completion time is NP-hard. This complexity is often seen in practice as well as in theory. A challenge problem posed in 1963—to find the optimal schedule for a

[1] In contrast, **consumable resources**, such as screws for assembling the engine, can be handled within the standard framework; see Exercise 12.2.

$$
\begin{aligned}
&Init(Chassis(C_1) \wedge Chassis(C_2)\\
&\quad \wedge Engine(E_1, C_1, 30) \wedge Engine(E_2, C_2, 60)\\
&\quad \wedge Wheels(W_1, C_1, 30) \wedge Wheels(W_2, C_2, 15)\\
&\quad \wedge EngineHoists(1) \wedge WheelStations(1) \wedge Inspectors(2))\\
&Goal(Done(C_1) \wedge Done(C_2))\\
&Action(AddEngine(e, c),\\
&\quad \text{PRECOND:} Engine(e, c, d) \wedge Chassis(c) \wedge \neg EngineIn(c),\\
&\quad \text{EFFECT:} EngineIn(c) \wedge Duration(d),\\
&\quad \text{RESOURCE:} EngineHoists(1))\\
&Action(AddWheels(w, c),\\
&\quad \text{PRECOND:} Wheels(w, c, d) \wedge Chassis(c) \wedge EngineIn(c),\\
&\quad \text{EFFECT:} WheelsOn(c) \wedge Duration(d),\\
&\quad \text{RESOURCE:} WheelStations(1))\\
&Action(Inspect(c),\\
&\quad \text{PRECOND:} EngineIn(c) \wedge WheelsOn(c),\\
&\quad \text{EFFECT:} Done(c) \wedge Duration(10),\\
&\quad \text{RESOURCE:} Inspectors(1))
\end{aligned}
$$

Figure 12.3 Job shop scheduling problem for assembling two cars, with resources. The available resources are one engine assembly station, one wheel assembly station, and two inspectors. The notation RESOURCE:r means that the resource r is used during execution of an action, but becomes free again when the action is complete.

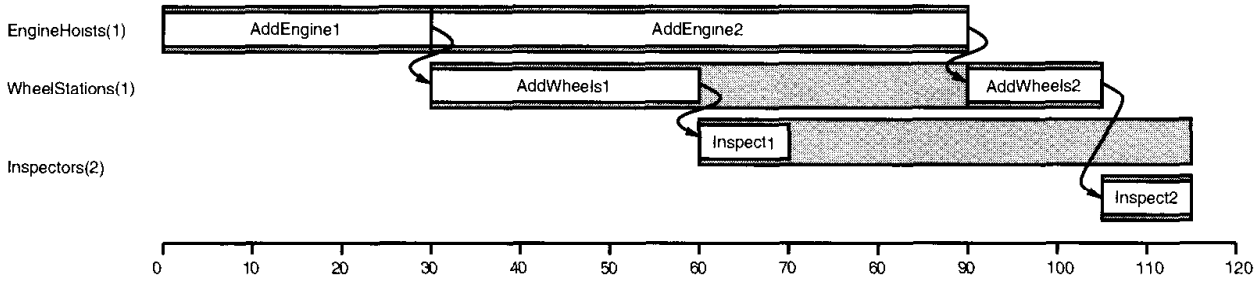


Figure 12.4 A solution to the job shop scheduling problem with resources from Figure 12.3. The left-hand margin lists the three resources, and actions are shown aligned horizontally with the resources they consume. There are two possible schedules, depending on which assembly uses the engine station first; we've shown the optimal solution, which takes 115 minutes.

problem involving just 10 machines and 10 jobs of 100 actions each—went unsolved for 23 years (Lawler *et al.*, 1993). Many approaches have been tried, including branch-and-bound, simulated annealing, tabu search, constraint satisfaction, and other techniques from Part II.

MINIMUM SLACK

One simple but popular heuristic is the **minimum slack** algorithm. It schedules actions in a greedy fashion. On each iteration, it considers the unscheduled actions that have had all their predecessors scheduled and schedules the one with the least slack for the earliest possible start. It then updates the ES and LS times for each affected action and repeats. The heuristic

is based on the same principle as the most-constrained-variable heuristic in constraint satisfaction. It often works well in practice, but for our assembly problem it yields a 130–minute solution, not the 115–minute solution of Figure 12.4.

The approach we have taken in this section is "plan first, schedule later": that is, we divided the overall problem into a *planning* phase in which actions are selected and partially ordered to meet the goals of the problem, and a later *scheduling* phase, in which temporal information is added to the plan to ensure that it meets resource and deadline constraints. This approach is common in real-world manufacturing and logistical settings, where the planning phase is often performed by human experts. When there are severe resource constraints, however, it could be that some legal plans will lead to much better schedules than others. In that case, it makes sense to *integrate* planning and scheduling by taking into account durations and overlaps during the construction of a partial-order plan. Several of the planning algorithms in Chapter 11 can be augmented to handle this information. For example, partial-order planners can detect resource constraint violations in much the same way that they detect conflicts with causal links. Heuristics can be modified to estimate the total completion time of a plan, rather than just the total cost of the actions. This is currently an active area of research.

12.2 HIERARCHICAL TASK NETWORK PLANNING

HIERARCHICAL DECOMPOSITION

One of the most pervasive ideas for dealing with complexity is **hierarchical decomposition**. Complex software is created from a hierarchy of subroutines or object classes, armies operate as a hierarchy of units, governments and corporations have hierarchies of departments, subsidiaries, and branch offices. The key benefit of hierarchical structure is that, at each level of the hierarchy, a computational task, military mission, or administrative function is reduced to a *small* number of activities at the next lower level, so that the computational cost of finding the correct way to arrange those activities for the current problem is small. Nonhierarchical methods, on the other hand, reduce a task to a *large* number of individual actions; for large-scale problems, this is completely impractical. In the best case—when high-level solutions always turn out to have satisfactory low-level implementations—hierarchical methods can result in linear-time instead of exponential-time planning algorithms.

HIERARCHICAL TASK NETWORK

This section describes a planning method based on **hierarchical task networks** or HTNs. The approach we take combines ideas from both partial-order planning (Section 11.3) and the area known as "HTN planning." In HTN planning, the initial plan, which describes the problem, is viewed as a very high-level description of what is to be done—for example, building a house. Plans are refined by applying **action decompositions**. Each action decomposition reduces a high-level action to a partially ordered set of lower-level actions. Action decompositions, therefore, embody knowledge about how to implement actions. For example, building a house might be reduced to obtaining a permit, hiring a contractor, doing the construction, and paying the contractor. (Figure 12.5 shows such a decomposition.) The process continues until only **primitive actions** remain in the plan. Typically, the primitive actions will be actions that the agent can execute automatically. For a general contractor,

ACTION DECOMPOSITION

PRIMITIVE ACTION

"install landscaping" might be primitive because it simply involves calling the landscaping contractor. For the landscaping contractor, actions such as "plant rhododendrons here" might be considered primitive.

In "pure" HTN planning, plans are generated *only* by successive action decompositions. The HTN therefore views planning as a process of making an activity description more *concrete*, rather than (as in the case of state-space and partial-order planning) a process of *constructing* an activity description, starting from the empty activity. It turns out that every STRIPS action description can be turned into an action decomposition (see Exercise 12.6), and that partial-order planning can be viewed as a special case of pure HTN planning. For certain tasks, however—especially "novel" conjunctive goals—the pure HTN viewpoint is rather unnatural, and we prefer to take a *hybrid* approach in which action decompositions are used as plan refinements in partial-order planning, in addition to the standard operations of establishing an open condition and resolving conflicts by adding ordering constraints. (Viewing HTN planning as an extension of partial-order planning has the additional advantage that we can use the same notational conventions instead of introducing a whole new set.) We begin by describing action decomposition in more detail. Then we explain how the partial-order planning algorithm must be modified to handle decompositions, and finally we discuss issues of completeness, complexity, and practicality.

Representing action decompositions

PLAN LIBRARY

General descriptions of action decomposition methods are stored in a **plan library**, from which they are extracted and instantiated to fit the needs of the plan being constructed. Each method is an expression of the form $Decompose(a, d)$. This says that an action a can be decomposed into the plan d, which is represented as a partial-order plan, as described in Section 11.3.

Building a house is a nice, concrete example, so we will use it to illustrate the concept of action decomposition. Figure 12.5 depicts one possible decomposition of the *BuildHouse* action into four lower-level actions. Figure 12.6 shows some of the action descriptions for the domain, as well as the decomposition for *BuildHouse* as it would appear in the plan library. There might be other possible decompositions in the library.

EXTERNAL PRECONDITIONS

EXTERNAL EFFECT

PRIMARY EFFECT

SECONDARY EFFECT

The *Start* action of the decomposition supplies all those preconditions of actions in the plan that are not supplied by other actions. We call these the **external preconditions**. In our example, the external preconditions of the decomposition are *Land* and *Money*. Similarly, the **external effects**, which are the preconditions of *Finish*, are all those effects of actions in the plan that are not negated by other actions. In our example, the external effects of *BuildHouse* are *House* and $\neg Money$. Some HTN planners also distinguish between **primary effects**, such as *House*, and **secondary effects**, such as $\neg Money$. Only primary effects may be used to achieve goals, whereas both kinds of effects might cause conflicts with other actions; this can greatly reduce the search space.[2]

[2] It could also prevent the discovery of unexpected plans. For example, a person facing bankruptcy proceedings can eliminate all liquid assets (i.e., achieve $\neg Money$) by buying or building a house. This plan is useful because current law precludes the seizure of a primary residence by creditors.

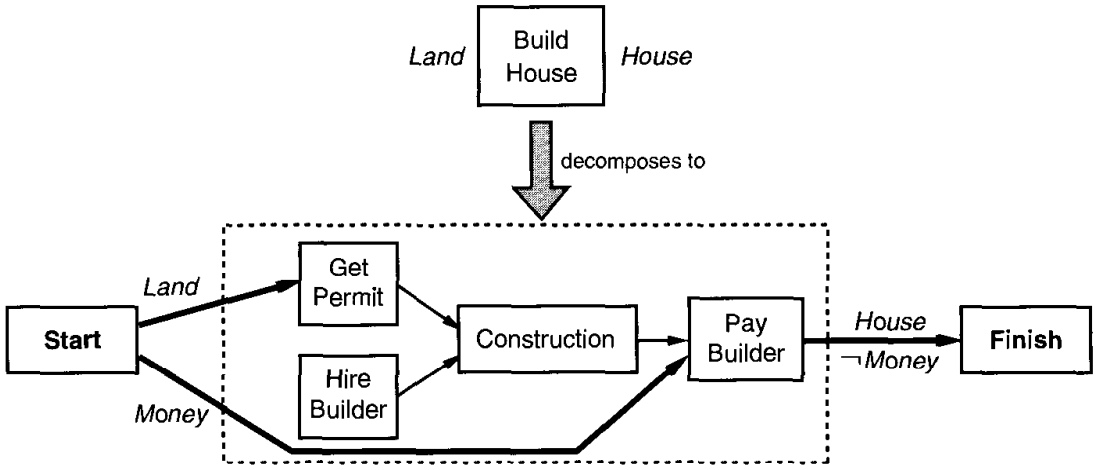


Figure 12.5 One possible decomposition for the *BuildHouse* action.

$Action(BuyLand,$ PRECOND:$Money,$ EFFECT:$Land \wedge \neg Money)$
$Action(GetLoan,$ PRECOND:$GoodCredit,$ EFFECT:$Money \wedge Mortgage)$
$Action(BuildHouse,$ PRECOND:$Land,$ EFFECT:$House)$

$Action(GetPermit,$ PRECOND:$Land,$ EFFECT:$Permit)$
$Action(HireBuilder,$ EFFECT:$Contract)$
$Action(Construction,$ PRECOND:$Permit \wedge Contract,$
EFFECT:$HouseBuilt \wedge \neg Permit)$
$Action(PayBuilder,$ PRECOND:$Money \wedge HouseBuilt,$
EFFECT:$\neg Money \wedge House \wedge \neg Contract)$

$Decompose(BuildHouse,$
$Plan($STEPS: $\{S_1 : GetPermit, S_2 : HireBuilder,$
$S_3 : Construction, S_4 : PayBuilder\}$
ORDERINGS: $\{Start \prec S_1 \prec S_3 \prec S_4 \prec Finish, \quad Start \prec S_2 \prec S_3\},$
LINKS: $\{Start \xrightarrow{Land} S_1, Start \xrightarrow{Money} S_4,$
$S_1 \xrightarrow{Permit} S_3, S_2 \xrightarrow{Contract} S_3, S_3 \xrightarrow{HouseBuilt} S_4,$
$S_4 \xrightarrow{House} Finish, S_4 \xrightarrow{\neg Money} Finish\}))$

Figure 12.6 Action descriptions for the house-building problem and a detailed decomposition for the *BuildHouse* action. The descriptions adopt a simplified view of money and an optimistic view of builders.

A decomposition should be a *correct* implementation of the action. A plan d implements an action a correctly if d is a complete and consistent partial-order plan for the problem of achieving the effects of a given the preconditions of a. Obviously, a decomposition will be correct if it is the result of running a sound partial-order planner.

A plan library could contain several decompositions for any given high-level action; for example, there might be another decomposition for *BuildHouse* that describes a process whereby the agent builds a house from rocks and turf with its own bare hands. Each de-

composition should be a correct plan, but it could have additional preconditions and effects beyond those stated in the high-level action description. For example, the decomposition for *BuildHouse* in Figure 12.5 requires *Money* in addition to *Land* and has the effect $\neg Money$. The self-build option, on the other hand, doesn't require money, but does require a ready supply of *Rocks* and *Turf*, and could result in a *BadBack*.

Given that a high-level action, such as *BuildHouse*, may have several possible decompositions, it is inevitable that its STRIPS action description will hide some of the preconditions and effects of those decompositions. The preconditions of the high-level action should be the *intersection* of the external preconditions of its decompositions, and the effects should be the intersection of the external effects of the decompositions. Put another way, the high-level preconditions and effects are guaranteed to be a subset of the true preconditions and effects of every primitive implementation.

INTERNAL EFFECT

Two other forms of information hiding should be noted. First, the high-level description completely ignores all **internal effects** of the decompositions. For example, our *BuildHouse* decomposition has temporary internal effects *Permit* and *Contract*.[3] Second, the high-level description does not specify the intervals "inside" the activity during which the high-level preconditions and effects must hold. For example, the *Land* precondition needs to be true (in our very approximate model) only until *GetPermit* is performed, and *House* is true only after *PayBuilder* is performed.

Information hiding of this kind is essential if hierarchical planning is to reduce complexity; we need to be able to reason about high-level actions without worrying about myriad details of the implementations. There is, however, a price to pay. For example, conflicts might exist between internal conditions of one high-level action and internal actions of another, but there is no way to detect this from the high-level descriptions. This issue has significant implications for HTN planning algorithms. In a nutshell, whereas primitive actions can be treated as point events by the planning algorithm, high-level actions have temporal extent within which all sorts of things can be going on.

Modifying the planner for decompositions

We now show how to modify POP to incorporate HTN planning. We do that by modifying the POP successor function (page 390) to allow decomposition methods to be applied to the current partial plan P. The new successor plans are formed by first selecting some nonprimitive action a' in P and then, for any $Decompose(a, d)$ method from the plan library such that a and a' unify with substitution θ, replacing a' with $d' = \text{SUBST}(\theta, d)$.

Figure 12.7 shows an example. At the top, there is a plan P for getting a house. The high-level action, $a' = BuildHouse$, is selected for decomposition. The decomposition d is selected from Figure 12.5, and *BuildHouse* is replaced by this decomposition. An additional step, *GetLoan*, is then introduced to resolve the new open condition, *Money*, that is created by the decomposition step. Replacing an action with its decomposition is a bit like transplant surgery: we have to take the new subplan out of its packaging (the *Start* and *Finish* steps),

[3] *Construction* negates the *Permit*, otherwise the same permit could be used to build many houses. Unfortunately, *Construction* does not terminate the *Contract* because we have to *PayBuilder* first.

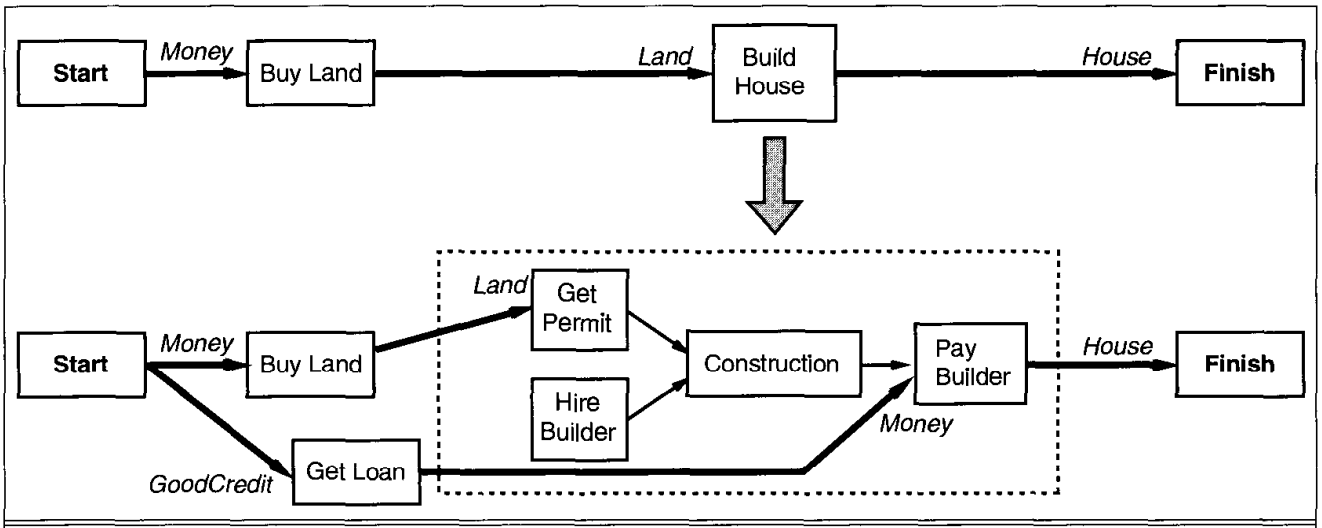


Figure 12.7 Decomposition of a high-level action within an existing plan. The *BuildHouse* action is replaced by the decomposition from Figure 12.5. The external precondition *Land* is supplied by the existing causal link from *BuyLand*. The external precondition *Money* remains open after the decomposition step, so we add a new action, *GetLoan*.

insert it, and hook everything up properly. There might be several ways to do this. To be more precise, we have for each possible decomposition d',

1. First, the action a' is removed from P. Then, for each step s in the decomposition d', we need to choose an action to fill the role of s and add it to the plan. It can be either a new instantiation of s *or an existing step* s' from P that unifies with s. For example, the decomposition of a *MakeWine* action might suggest that we *BuyLand*; possibly, we can use the same *BuyLand* action that we already have in the plan. We call this **subtask sharing**.

SUBTASK SHARING

   In Figure 12.7, there are no sharing opportunities, so new instances of the actions are created. Once the actions have been chosen, all the internal constraints from d' are copied over—for example, that *GetPermit* is ordered before *Construction* and that there is a causal link between these two steps supplying the *Permit* precondition of *Construction*. This completes the task of replacing a' with the instantiation of $d\theta$.

2. The next step is to hook up the ordering constraints for a' in the original plan to the steps in d'. First, consider an ordering constraint in P of the form $B \prec a'$. How should B be ordered with respect to the steps in d'? The most obvious solution is that B should come before every step in d', and that can be achieved by replacing every constraint of the form $Start \prec s$ in d' with a constraint $B \prec s$. On the other hand, this approach might be too strict! For example, *BuyLand* has to come before *BuildHouse*, but there is no need for *BuyLand* to come before *HireBuilder* in the expanded plan. Imposing an overly strict ordering might prevent some solutions from being found. Therefore, the best solution is for each ordering constraint to record the *reason* for the constraint; then, when a high-level action is expanded, the new ordering constraints can be as relaxed as possible, consistent with the reason for the original constraint. Exactly the same considerations apply when we are replacing constraints of the form $a' \prec C$.

3. The final step is to hook up causal links. If $B \xrightarrow{p} a'$ was a causal link in the original plan, replace it by a set of causal links from B to all the steps in d' with preconditions p that were supplied by the *Start* step in the decomposition d (i.e., to all the steps in d' for which p is an *external* precondition). In the example, the causal link $BuyLand \xrightarrow{Land} BuildHouse$ is replaced by the link $BuyLand \xrightarrow{Land} Permit$. (The *Money* precondition for *PayBuilder* in the decomposition becomes an open condition, because no action in the original plan supplies *Money* to *BuildHouse*.) Similarly, for each causal link $a' \xrightarrow{p} C$ in the plan, replace it with a set of causal links to C from whichever step in d' supplies p to the *Finish* step in the decomposition d (i.e., from the step in d' that has p as an *external* effect). In the example, we replace the link $BuildHouse \xrightarrow{House} Finish$ with the link $PayBuilder \xrightarrow{House} Finish$.

This completes the additions required for generating decompositions in the context of the POP planner.[4]

Additional modifications to the POP algorithm are required because of the fact that high-level actions *hide information* about their final primitive implementations. In particular, the original POP algorithm backtracks with failure if the current plan contains an irresolvable conflict—that is, if an action conflicts with a causal link but cannot be ordered either before or after the link. (Figure 11.9 shows an example.) With high-level actions, on the other hand, apparently irresolvable conflicts can sometimes be resolved by *decomposing* the conflicting actions and interleaving their steps. An example is given in Figure 12.8. Thus, it may be the case that a complete and consistent primitive plan can be obtained by decomposition *even when no complete and consistent high-level plan exists*. This possibility means that a complete HTN planner must forgo many pruning opportunities that are available to a standard POP planner. Alternatively, we can prune anyway and hope that no solution is missed.

Discussion

Let's begin with the bad news: pure HTN planning (where the only allowable plan refinement is decomposition) is undecidable, *even though the underlying state space is finite*! This might seem very depressing, since the point of HTN planning is to gain efficiency. The difficulty arises because action decompositions can be **recursive**—for example, going for a walk can be implemented by taking a step and then going for a walk—so HTN plans can be arbitrarily long. In particular, the shortest HTN solution could be arbitrarily long, so that there is no way to terminate the search after any fixed time. There are, however, at least three ways to look on the bright side:

1. We can rule out recursion, which very few domains require. In that case, all HTN plans are of finite length and can be enumerated.
2. We can bound the length of solutions we care about. Because the state space is finite, a plan that has more steps than there are states in the state space *must* include a loop that visits the same state twice. We would lose little by ruling out HTN solutions of this kind, and we would control the search.

[4] There are some additional minor modifications required for handling conflict resolution with high-level actions; the interested reader can consult the papers cited at the end of the chapter.

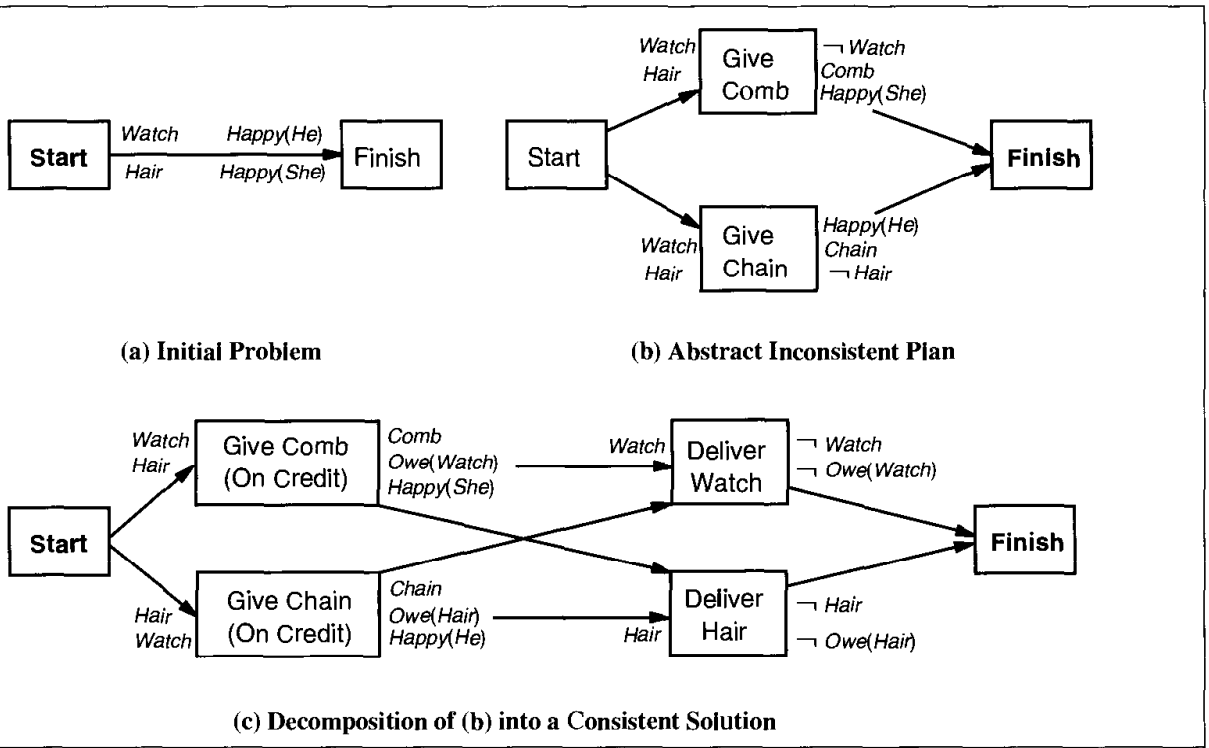


Figure 12.8 The *Gift of the Magi* problem, taken from the O. Henry story, shows an inconsistent abstract plan that nevertheless can be decomposed into a consistent solution. Part (a) shows the problem: A poor couple has only two prized possessions—he a gold watch and she her beautiful long hair. Each plans to buy a present to make the other happy. He decides to trade his watch to buy a silver comb for her hair, and she decides to sell her hair to get a gold chain for his watch. In (b) the partial plan is inconsistent, because there is no way to order the "Give Comb" and "Give Chain" abstract steps without a conflict. (We assume that the "Give Comb" action has the precondition *Hair*, because if the wife doesn't have her long hair, the action won't have the intended effect of making her happy, and similarly for the "Give Chain" action.) In (c) we decompose the "Give Comb" step with an "installment plan" method. In the first step of the decomposition, the husband takes possession of the comb and gives it to his wife, while agreeing to deliver the watch in payment at a later date. In the second step, the watch is handed over and the obligation is fulfilled. A similar method decomposes the "Give Chain" step. As long as both giving steps are ordered before the delivery steps, this decomposition solves the problem. (Note that it relies on the problem being defined so that the happiness of using the chain with the watch or the comb with the hair persists even after the possessions are surrendered.)

3. We can adopt the hybrid approach that combines POP and HTN planning. Partial-order planning by itself suffices to decide whether a plan exists, so the hybrid problem is clearly decidable.

We need to be a little bit careful with the third answer. POP can string together primitive actions in arbitrary ways, so we might find ourselves with solutions that are very hard to understand and do not have the nice, hierarchical organization of HTN plans. An appropriate compromise is to control the hybrid search so that action decompositions are preferred over adding new actions, although not to such an extent that arbitrarily long HTN plans are generated before any primitive actions can be added. One way to do this is to use a cost function

that gives a discount for actions introduced by decomposition; the larger the discount, the more the search will resemble pure HTN planning and the more hierarchical the solution will be. Hierarchical plans are usually much easier to execute in realistic settings, and are easier to fix when things go wrong.

Another important characteristic of HTN plans is the possibility of subtask sharing. Recall that subtask sharing means using the same action to implement two different steps in plan decompositions. If we disallow subtask sharing, then each instantiation of a decomposition d' can be done in only one way, rather than many, thereby greatly pruning the search space. Usually, this pruning saves some time and at worst leads to a solution that is slightly longer than optimal. In some cases, however, it can be more problematic. For example, consider the goal "enjoy a honeymoon and raise a family." The plan library might come up with "get married and go to Hawaii" for the first subgoal and "get married and have two children" for the second. Without subtask sharing, the plan will include two distinct marriage actions, often considered highly undesirable.

An interesting example of the costs and benefits of subtask sharing occurs in optimizing compilers. Consider the problem of compiling the expression $\tan(x) - \sin(x)$. Most compilers accomplish this by merging two separate subroutine calls in a trivial way: all the steps of tan come before any of the steps of sin. But consider the following Taylor series approximations for sin and tan:

$$\tan x \approx x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^7}{315}; \quad \sin x \approx x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} .$$

An HTN planner with subtask sharing could generate a more efficient solution, because it could choose to implement many steps of the sin computation with existing steps from tan. Most compilers do not do this kind of interprocedural sharing because it would take too much time to consider all the possible shared plans. Instead, most compilers generate each subplan independently, and then perhaps modify the result with a peephole optimizer.

Given all the additional complications caused by the introduction of action decompositions, why do we believe that HTN planning can be efficient? The actual sources of complexity are hard to analyze in practice, so we consider an idealized case. Suppose, for example, that we want to construct a plan with n actions. For a nonhierarchical, forward state-space planner with b allowable actions at each state, the cost is $O(b^n)$. For an HTN planner, let us suppose a very regular decomposition structure: each nonprimitive action has d possible decompositions, each into k actions at the next lower level. We want to know how many different decomposition trees there are with this structure. Now, if there are n actions at the primitive level, then the number of levels below the root is $\log_k n$, so the number of internal decomposition nodes is $1 + k + k^2 + \cdots + k^{\log_k n - 1} = (n-1)/(k-1)$. Each internal node has d possible decompositions, so there are $d^{(n-1)/(k-1)}$ possible regular decomposition trees that could be constructed. Examining this formula, we see that keeping d small and k large can result in huge savings: essentially we are taking the kth root of the nonhierarchical cost, if b and d are comparable. On the other hand, constructing a plan library that has a small number of long decompositions, but nonetheless allows us to solve any problem, is not always possible. Another way of saying this is that long macros that are usable across a wide range of problems are extremely precious.

Another, and perhaps better, reason for believing that HTN planning is efficient is that it works in practice. Almost all planners for large-scale applications are HTN planners, because HTN planning allows the human expert to provide the crucial knowledge about how to perform complex tasks so that large plans can be constructed with little computational effort. For example, O-PLAN (Bell and Tate, 1985), which combines HTN planning with scheduling, has been used to develop production plans for Hitachi. A typical problem involves a product line of 350 different products, 35 assembly machines, and over 2000 different operations. The planner generates a 30-day schedule with three 8-hour shifts a day, involving millions of steps.

The key to HTN planning, then, is the construction of a plan library containing known methods for implementing complex, high-level actions. One method of constructing the library is to *learn* the methods from problem-solving experience. After the excruciating experience of constructing a plan from scratch, the agent can save the plan in the library as a method for implementing the high-level action defined by the task. In this way, the agent can become more and more competent over time as new methods are built on top of old methods. One important aspect of this learning process is the ability to *generalize* the methods that are constructed, eliminating detail that is specific to the problem instance (e.g., the name of the builder or the address of the plot of land) and keeping just the key elements of the plan. Methods for achieving this kind of generalization are described in Chapter 19. It seems to us inconceivable that humans could be as competent as they are without some such mechanism.

12.3 PLANNING AND ACTING IN NONDETERMINISTIC DOMAINS

So far we have considered only **classical planning** domains that are fully observable, static, and deterministic. Furthermore, we have assumed that the action descriptions are correct and complete. In these circumstances, an agent can plan first and then execute the plan "with its eyes closed." In an uncertain environment, on the other hand, an agent must use its percepts to discover what is happening while the plan is being executed and possibly modify or replace the plan if something unexpected happens.

Agents have to deal with both *incomplete* and *incorrect* information. Incompleteness arises because the world is partially observable, nondeterministic, or both. For example, the door to the office supply cabinet might or might not be locked; one of my keys might or might not open the door if it is locked; and I might or might not be aware of these kinds of incompleteness in my knowledge. Thus, my model of the world is weak, but correct. On the other hand, incorrectness arises because the world does not necessarily match my model of the world; for example, I might *believe* that my key opens the supply cabinet, but I could be wrong if the locks have been changed. Without the ability to handle incorrect information, an agent can end up being as unintelligent as the dung beetle (page 37), which attempts to plug up its nest with a ball of dung even after the ball has been removed from its grasp.

BOUNDED INDETERMINACY

The possibility of having complete or correct knowledge depends on *how much* indeterminacy there is in the world. With **bounded indeterminacy**, actions can have unpredictable

effects, but the possible effects can be listed in the action description axioms. For example, when we flip a coin, it is reasonable to say that the outcome will be *Heads* or *Tails*. An agent can cope with bounded indeterminacy by making plans that work in all possible circumstances. With **unbounded indeterminacy**, on the other hand, the set of possible preconditions or effects either is unknown or is too large to be enumerated completely. This would be the case in very complex or dynamic domains such as driving, economic planning, and military strategy. An agent can cope with unbounded indeterminacy only if it is prepared to revise its plans and/or its knowledge base. Unbounded indeterminacy is closely related to the **qualification problem** discussed in Chapter 10—the impossibility of listing *all* the preconditions required for a real-world action to have its intended effect.

UNBOUNDED INDETERMINACY

There are four planning methods for handling indeterminacy. The first two are suitable for bounded indeterminacy, and the second two for unbounded indeterminacy:

SENSORLESS PLANNING

- **Sensorless planning**: Also called **conformant planning**, this method constructs standard, sequential plans that are to be executed without perception. The sensorless planning algorithm must ensure that the plan achieves the goal *in all possible circumstances*, regardless of the true initial state and the actual action outcomes. Sensorless planning relies on **coercion**—the idea that the world can be forced into a given state even when the agent has only partial information about the current state. Coercion is not always possible, so sensorless planning is often inapplicable. Sensorless problem solving, involving search in belief state space, was described in Chapter 3.

CONDITIONAL PLANNING

- **Conditional planning**: Also known as **contingency planning**, this approach deals with bounded indeterminacy by constructing a conditional plan with different branches for the different contingencies that could arise. Just as in classical planning, the agent plans first and then executes the plan that was produced. The agent finds out which part of the plan to execute by including **sensing actions** in the plan to test for the appropriate conditions. In the air transport domain, for example, we could have plans that say "check whether SFO airport is operational. If so, fly there; otherwise, fly to Oakland." Conditional planning is covered in Section 12.4.

SENSING ACTIONS

EXECUTION MONITORING AND REPLANNING

- **Execution monitoring and replanning**: In this approach, the agent can use any of the preceding planning techniques (classical, sensorless, or conditional) to construct a plan, but it also uses **execution monitoring** to judge whether the plan has a provision for the actual current situation or need to be revised. **Replanning** occurs when something goes wrong. In this way, the agent can handle unbounded indeterminacy. For example, even if a replanning agent did not envision the possibility of SFO's being closed, it can recognize that situation when it occurs and call the planner again to find a new path to the goal. Replanning agents are covered in Section 12.5.

CONTINUOUS PLANNING

- **Continuous planning**: All the planners we have seen so far are designed to achieve a goal and then stop. A continuous planner is designed to persist over a lifetime. It can handle unexpected circumstances in the environment, even if these occur while the agent is in the middle of constructing a plan. It can also handle the abandonment of goals and the creation of additional goals by **goal formulation**. Continuous planning is described in Section 12.6.

Let's consider an example to clarify the differences among the various kinds of agents. The problem is this: given an initial state with a chair, a table, and some cans of paint, with everything of unknown color, achieve the state where the chair and table have the same color.

A **classical planning** agent could not handle this problem, because the initial state is not fully specified—we don't know what color the furniture is.

A **sensorless planning** agent must find a plan that works without requiring any sensors during plan execution. The solution is to open any can of paint and apply it to both chair and table, thus **coercing** them to be the same color (even though the agent doesn't know what the color is). Coercion is most appropriate when propositions are expensive or impossible to perceive. For example, doctors often prescribe a broad-spectrum antibiotic rather than using the conditional plan of doing a blood test, then waiting for the results to come back, and then prescribing a more specific antibiotic. They do this because the delays and costs involved in performing the blood tests are usually too great.

A **conditional planning** agent can generate a better plan: first sense the color of the table and chair; if they are already the same then the plan is done. If not, sense the labels on the paint cans; if there is a can that is the same color as one piece of furniture, then apply the paint to the other piece. Otherwise paint both pieces with any color.

A **replanning** agent could generate the same plan as the conditional planner, or it could generate fewer branches at first and fill in the others at execution time as needed. It could also deal with incorrectness of its action descriptions. For example, suppose that the $Paint(obj, color)$ action is believed to have the deterministic effect $Color(obj, color)$. A conditional planner would just assume that the effect has occurred once the action has been executed, but a replanning agent would check for the effect, and if it were not true (perhaps because the agent was careless and missed a spot), it could then replan to repaint the spot. We will return to this example on page 441.

A **continuous planning** agent, in addition to handling unexpected events, can revise its plans appropriately if, say, we add the goal of having dinner on the table, so that the painting plan must be postponed.

In the real world, agents use a combination of approaches. Car manufacturers sell spare tires and air bags, which are physical embodiments of conditional plan branches designed to handle punctures or crashes; on the other hand, most car drivers never consider these possibilities, so they respond to punctures and crashes as replanning agents. In general, agents create conditional plans only for those contingencies that have important consequences and a nonnegligible chance of going wrong. Thus, a car driver contemplating a trip across the Sahara desert might do well to consider explicitly the possibility of breakdowns, whereas a trip to the supermarket requires less advance planning.

The agents we describe in this chapter are designed to handle indeterminacy, but are not capable of making tradeoffs between the probability of success and the cost of plan construction. Chapter 16 provides additional tools for dealing with these issues.

12.4 CONDITIONAL PLANNING

Conditional planning is a way to deal with uncertainty by checking what is actually happening in the environment at predetermined points in the plan. Conditional planning is simplest to explain for fully observable environments, so we will begin with that case. The partially observable case is more difficult, but more interesting.

Conditional planning in fully observable environments

Full observability means that the agent always knows the current state. If the environment is nondeterministic, however, the agent will not be able to predict the *outcome* of its actions. A conditional planning agent handles nondeterminism by building into the plan (at planning time) conditional steps that will check the state of the environment (at execution time) to decide what to do next. The problem, then, is how to construct these conditional plans.

We will use as our example domain the venerable **vacuum world**, whose state space, for the deterministic case, is laid out on page 65. Recall that the available actions are *Left*, *Right*, and *Suck*. We will need some propositions to define the states: let AtL (AtR) be true if the agent is in the left (right) state,[5] and let $CleanL$ ($CleanR$) be true if the left (right) state is clean. The first thing we need to do is augment the STRIPS language to allow for nondeterminism. To do this, we allow actions to have **disjunctive effects**, meaning that the action can have two or more different outcomes whenever it is executed. For example, suppose that moving *Left* sometimes fails. Then the normal action description

DISJUNCTIVE EFFECTS

$$Action(Left, \text{PRECOND:}AtR, \text{EFFECT:}AtL \wedge \neg AtR)$$

must be modified to include a disjunctive effect:

$$Action(Left, \text{PRECOND:}AtR, \text{EFFECT:}AtL \vee AtR)\ . \tag{12.1}$$

CONDITIONAL EFFECTS

We will also find it useful to allow actions to have **conditional effects**, wherein the effect of the action depends on the state in which it is executed. Conditional effects appear in the EFFECT slot of an action, and have the syntax "**when** $<condition>$: $<effect>$." For example, to model the *Suck* action, we would write

$$Action(Suck, \text{PRECOND:}, \text{EFFECT:}(\textbf{when}\ AtL\text{: } CleanL) \wedge (\textbf{when}\ AtR\text{: } CleanR))].$$

Conditional effects do not introduce indeterminacy, but they can help to model it. For example, suppose we have a devious vacuum cleaner that sometimes dumps dirt on the destination square when it moves, but only if that square is clean. This can be modeled with a description such as

$$Action(Left, \text{PRECOND:}AtR, \text{EFFECT:}AtL \vee (AtL \wedge \textbf{when}\ CleanL\text{: } \neg CleanL)),$$

which is both disjunctive and conditional.[6] To create conditional plans, we need **conditional steps**. We will write these using the syntax "**if** $<test>$ **then** $plan\_A$ **else** $plan\_B$," where

CONDITIONAL STEPS

[5] Obviously, AtR is true iff $\neg AtL$ is true, and vice versa. We use two propositions mainly to improve readability.

[6] The conditional effect **when** $CleanL$: $\neg CleanL$ may look a little odd. Remember, however, that here $CleanL$ refers to the situation *before* the action and $\neg CleanL$ refers to the situation *after* the action.

$<test>$ is a Boolean function of the state variables. For example, a conditional step for the vacuum world might be, "**if** $AtL \wedge CleanL$ **then** *Right* **else** *Suck*." The execution of such a step proceeds in the obvious way. By nesting conditional steps, plans become trees.

We want conditional plans that work *regardless of which action outcomes actually occur*. We have seen this problem before, in a different guise. In two-player games (Chapter 6), we want moves that will win *regardless of which moves the opponent makes*. For this reason, nondeterministic planning problems are often called **games against nature**.

GAMES AGAINST NATURE

Let us consider a specific example in the vacuum world. The initial state has the robot in the right square of a clean world; because the environment is fully observable, the agent knows the full state description, $AtR \wedge CleanL \wedge CleanR$. The goal state has the robot in the left square of a clean world. This would be quite trivial, were it not for the "double Murphy" vacuum cleaner that sometimes deposits dirt when it moves to a clean destination square and sometimes deposits dirt if *Suck* is applied to a clean square.

A "game tree" for this environment is shown in Figure 12.9. Actions are taken by the robot in the "state" nodes of the tree, and nature decides what the outcome will be at the "chance" nodes, shown as circles. A solution is a subtree that (1) has a goal node at every leaf, (2) specifies one action at each of its "state" nodes, and (3) includes every outcome branch at each of its "chance" nodes. The solution is shown in bold lines in the figure; it corresponds to the plan $[Left, \textbf{if } AtL \wedge CleanL \wedge CleanR \textbf{ then } [\,] \textbf{ else } Suck]$. (For now, because we are using a state-space planner, the tests in conditional steps will be complete state descriptions.)

For exact solutions of games, we use the **minimax algorithm** (Figure 6.3). For conditional planning, there are typically two modifications. First, MAX and MIN nodes can become

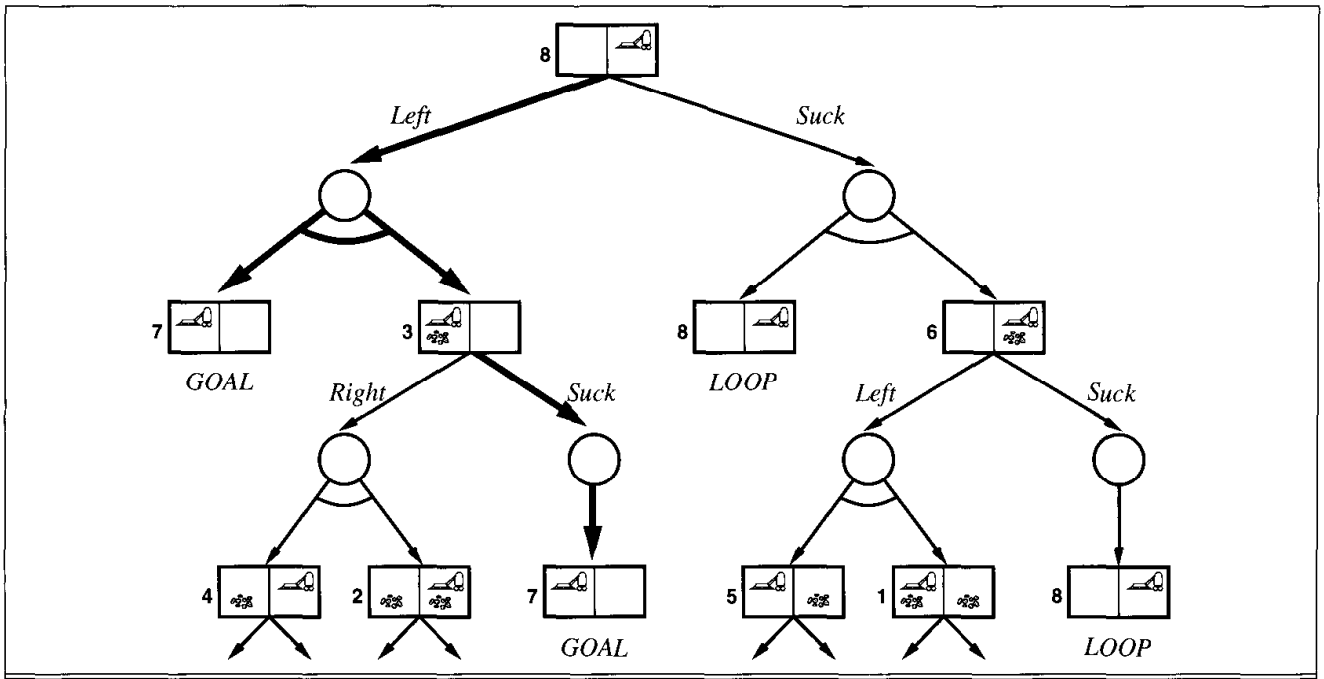


Figure 12.9 The first two levels of the search tree for the "double Murphy" vacuum world. State nodes are OR nodes where some action must be chosen. Chance nodes, shown as circles, are AND nodes where every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution is shown in bold lines.

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(INITIAL-STATE[problem], problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if GOAL-TEST[problem](state) then return the empty plan
  if state is on path then return failure
  for each action, state_set in SUCCESSORS[problem](state) do
      plan ← AND-SEARCH(state_set, problem, [state | path])
      if plan ≠ failure then return [action | plan]
  return failure

function AND-SEARCH(state_set, problem, path) returns a conditional plan, or failure
  for each s_i in state_set do
      plan_i ← OR-SEARCH(s_i, problem, path)
      if plan = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

Figure 12.10 An algorithm for searching AND–OR graphs generated by nondeterministic environments. We assume that SUCCESSORS returns a list of actions, each associated with a *set* of possible outcomes. The aim is to find a conditional plan that reaches a goal state in all circumstances.

OR and AND nodes. Intuitively, the plan needs to take *some* action at every state it reaches, but must handle *every* outcome for the action it takes. Second, the algorithm needs to return a conditional plan rather than just a single move. At an OR node, the plan is just the action selected, followed by whatever comes next. At an AND node, the plan is a nested series of if–then–else steps specifying subplans for each outcome; the tests in these steps are the complete state descriptions.[7]

Formally speaking, the search space we have defined is an AND–OR **graph**. In Chapter 7, AND–OR graphs showed up in propositional Horn clause inference. Here, the branches are actions rather than logical inference steps, but the algorithm is the same. Figure 12.10 gives a recursive, depth-first algorithm for AND–OR graph search.

One key aspect of the algorithm is the way in which it deals with cycles, which often arise in nondeterministic planning problems (e.g., if an action sometimes has no effect, or if an unintended effect can be corrected). If the current state is identical to a state on the path from the root, then it returns with failure. This doesn't mean that there is *no* solution from the current state; it simply means that if there *is* a noncyclic solution, it must be reachable from the earlier incarnation of the current state, so the new incarnation can be discarded. With this check, we ensure that the algorithm terminates in every finite state space, because every path must reach a goal, a dead end, or a repeated state. Notice that the algorithm does not check whether the current state is a repetition of a state on some *other* path from the root. Exercise 12.15 investigates this issue.

[7] Such plans could also be written using a **case** construct.

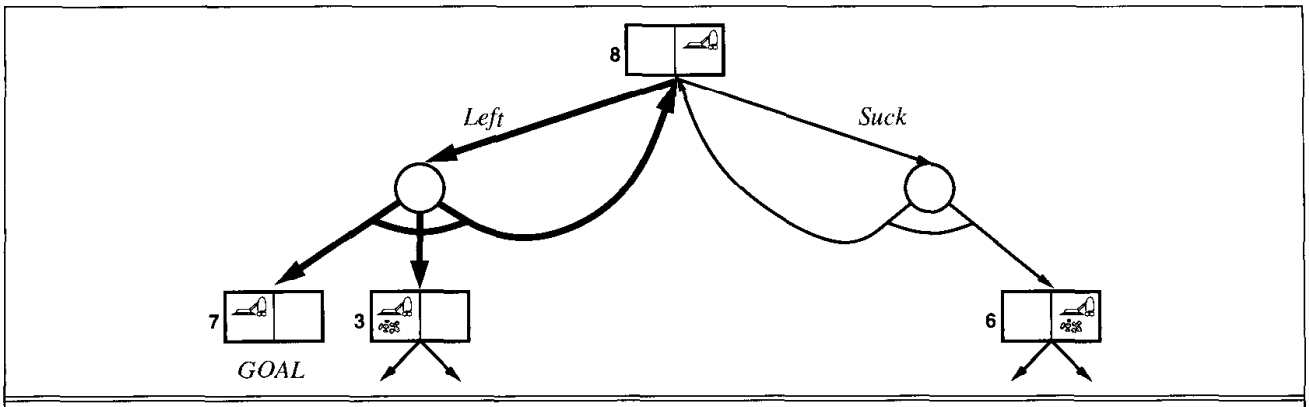


Figure 12.11 The first level of the search graph for the "triple Murphy" vacuum world, where we have shown cycles explicitly. All solutions for this problem are cyclic plans.

The plans returned by AND-OR-GRAPH-SEARCH contain conditional steps that test the entire state description to decide on a branch. In many cases, we can get away with less exhaustive tests. For example, the solution plan in Figure 12.9 could be written simply as $[Left, \textbf{if } CleanL \textbf{ then } [\,] \textbf{ else } Suck]$. This is because the single test, $CleanL$, suffices to divide the states at the AND-node into two singleton sets, so that after the test the agent knows exactly what state it is in. In fact, a series of if–then–else tests of single variables always suffices to divide a set of states into singletons, *provided* that the state is fully observable. We could, therefore, restrict the tests to be of single variables without loss of generality.

There is one final complication that often arises in nondeterministic domains: things don't always work the first time, and one has to try again. For example, consider the "triple Murphy" vacuum cleaner, which (in addition to its previously stated habits) sometimes fails to move when commanded—for example, $Left$ can have the disjunctive effect $AtL \vee AtR$, as in Equation (12.1). Now the plan $[Left, \textbf{if } CleanL \textbf{ then } [\,] \textbf{ else } Suck]$ is no longer guaranteed to work. Figure 12.11 shows part of the the search graph; clearly, there are no longer any acyclic solutions, and AND-OR-GRAPH-SEARCH would return with failure. There is, however, a

CYCLIC SOLUTION

cyclic solution, which is to keep trying $Left$ until it works. We can express this solution

LABEL

by adding a **label** to denote some portion of the plan and using that label later instead of repeating the plan itself. Thus, our cyclic solution is

$$[L_1 : Left, \textbf{if } AtR \textbf{ then } L_1 \textbf{ else if } CleanL \textbf{ then } [\,] \textbf{ else } Suck] .$$

(A better syntax for the looping part of this plan would be "**while** AtR **do** $Left$.") The modifications needed to AND-OR-GRAPH-SEARCH are covered in Exercise 12.16. The key realization is that a loop in the state space back to a state L translates to a loop in the plan back to the point where the subplan for state L is executed.

Now we have the ability to synthesize complex plans that look more like programs, with conditionals and loops. Unfortunately, these loops are, potentially, *infinite* loops. For example, nothing in the action representation for the triple Murphy world says that $Left$ will eventually succeed. Cyclic plans are therefore less desirable than acyclic plans, but they may be considered solutions, provided that every leaf is a goal state and a leaf is reachable from every point in the plan.

Conditional planning in partially observable environments

The preceding section dealt with fully observable environments, which have the advantage that conditional tests can ask any question at all and be sure of getting an answer. In the real world, partial observability is much more common. In the initial state of a partially observable planning problem, the agent knows only a certain amount about the actual state. The simplest way to model this situation is to say that the initial state belongs to a **state set**; the state set is a way of describing the agent's initial **belief state**.[8]

Suppose that a vacuum-world agent knows that it is in the right-hand square and that the square is clean, but it cannot sense the presence or absence of dirt in other squares. Then *as far as it knows* it could be in one of two states: the left-hand square might be either clean or dirty. This belief state is marked A in Figure 12.12. The figure shows part of the AND–OR graph for the "alternate double Murphy" vacuum world, in which dirt can sometimes be left behind when the agent leaves a clean square.[9] If the world were fully observable, the agent could construct a cyclic solution of the form "Keep moving left and right, sucking up dirt whenever it appears, until both squares are clean and I'm in the left square." (See Exercise 12.16.) Unfortunately, with local dirt sensing, this plan is unexecutable, because the truth value of the test "both squares are clean" cannot be determined.

Let us look at how the AND–OR graph is constructed. From belief state A, we show the outcome of moving *Left*. (The other actions make no sense.) Because the agent can leave dirt behind, the two possible initial worlds become four possible worlds, as shown in B and C. The worlds form two distinct belief states, classified by the available sensor information.[10] In B, the agent knows $CleanL$; in C it knows $\neg CleanL$. From C, cleaning up the dirt moves the agent to B. From B, moving *Right* might or might not leave dirt behind, so there are again four possible worlds, divided according to the agent's knowledge of $CleanR$ (back to A) or $\neg CleanR$ (belief state D).

In sum, nondeterministic, partially observable environments give us an AND–OR graph of belief states. Conditional plans can be found, therefore, using exactly the same algorithm as in the fully observable case, namely AND-OR-GRAPH-SEARCH. Another way to understand what's going on is to see that the agent's *belief* state is *always* fully observable—it always knows what it knows. "Standard" fully observable problem solving is just a special case in which every belief state is a singleton set containing exactly one physical state.

Are we done? Not quite! We still need to decide how belief states should be represented, how sensing works, and how action descriptions should be written in this new setting.

There are three basic choices for belief states:

1. Sets of full state descriptions. For example, the initial belief state in Figure 12.12 is

$$\{(AtR \wedge CleanR \wedge CleanL),\ (AtR \wedge CleanR \wedge \neg CleanL)\}\ .$$

This representation is simple to work with, but very expensive: if there are n Boolean

[8] These concepts are introduced in Section 3.6, which the reader might wish to consult before proceeding.

[9] Parents with young children will be familiar with this phenomenon. Apologies to others, as usual.

[10] Notice that they are *not* classified by whether there is dirt left behind when the agent moves. Branching in belief-state space is caused by alternative knowledge outcomes, not alternative physical outcomes.

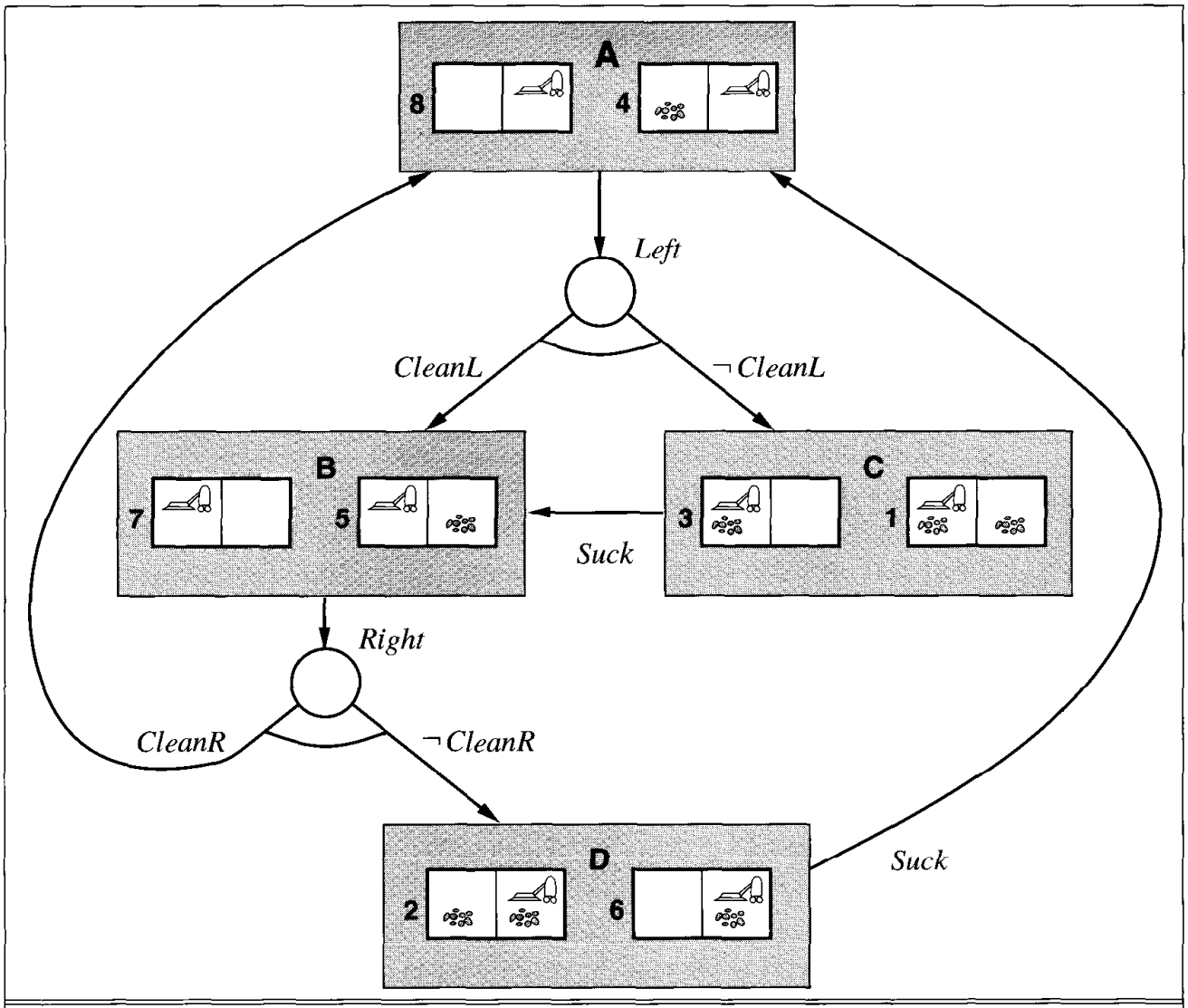


Figure 12.12 Part of the AND–OR graph for the "alternate double Murphy" vacuum world, in which dirt can sometimes be left behind when the agent leaves a clean square. The agent cannot sense dirt in other squares.

propositions defining the state, then a belief state can contain $O(2^n)$ physical state descriptions, each of size $O(n)$. Exponentially large belief states will occur whenever the agent knows only a fraction of the propositions—the less it knows, the more possible states it might be in.

2. Logical sentences that capture exactly the set of possible worlds in the belief state. For example, the initial state can be written as

$$AtR \wedge CleanR \ .$$

Clearly, every belief state can be captured exactly by a single logical sentence; if we have to, we can use the disjunction of all the conjunctive state descriptions, but our example shows that more compact sentences could exist.

One drawback with general logical sentences is that, because there are many different, logically equivalent sentences that describe the same belief state, repeated state checking in the graph search algorithm can require general theorem proving. For this

reason, we would like a *canonical* representation for sentences in which every belief state corresponds to exactly one sentence.[11] One such representation uses a conjunction of literals ordered by proposition name—$AtR \wedge CleanR$ is an example. This is just the standard state representation under the **open-world assumption** from Chapter 11. Not all logical sentences can be written in such form—for example, there is no way to represent $AtL \vee CleanR$—but many domains can be handled.

3. **Knowledge propositions** describing the agent's knowledge. (See also Section 7.7.) For the initial state, we have

$$K(AtR) \wedge K(CleanR) .$$

Here, K stands for "knows that" and $K(P)$ means that the agent knows that P is true.[12] With knowledge propositions, we use the **closed-world assumption**—if a knowledge proposition does not appear in the list, it is assumed false. For example, $\neg K(CleanL)$ and $\neg K(\neg CleanL)$ are implicit in the sentence above, so it captures the fact that the agent is ignorant of the truth value of $CleanL$.

It turns out that the second and third options are roughly equivalent, but we will use the third option, knowledge propositions, because it gives a more vivid description of sensing and because we already know how to write STRIPS descriptions with the closed-world assumption.

In both options, each proposition symbol can appear in one of three ways: positive, negative, or unknown. Therefore, there are exactly 3^n possible belief states that can be described this way. Now, the set of belief states is the powerset (set of all subsets) of the set of physical states. There are 2^n physical states, so there are 2^{2^n} belief states—far more than 3^n, so options 2 and 3 are quite restricted as representations of belief states. This currently is believed to be inevitable, because *any scheme capable of representing every possible belief state will require* $O(\log_2(2^{2^n})) = O(2^n)$ *bits to represent each one in the worst case.* Our simple schemes require only $O(n)$ bits to represent each belief state, trading expressiveness for compactness. In particular, if an action occurs, one of whose preconditions is unknown, then the resulting belief state will not be exactly representable and the action outcome becomes unknown.

Now we need to decide how sensing works. There are two choices here. We can have **automatic sensing**, which means that at every time step the agent gets all the available percepts. The example in Figure 12.12 assumes automatic sensing of location and local dirt. Alternatively, we can insist on **active sensing**, which means that percepts are obtained only by executing specific **sensory actions** such as $CheckDirt$ and $CheckLocation$. We will treat each kind of sensing in turn.

AUTOMATIC SENSING

ACTIVE SENSING

SENSORY ACTIONS

Let us now write an action description using knowledge propositions. Suppose the agent moves $Left$ in the alternate-double-Murphy world with automatic local dirt sensing; according to the rules for that world, the agent might or might not leave dirt behind if the square was clean. As a *physical* effect, this would be *disjunctive*; but as a *knowledge* effect, it

[11] The best-known canonical representation for a general propositional sentence is the **binary decision diagram**, or BDD (Bryant, 1992).

[12] This is the same notation used for circuit-based agents in Chapter 7. Some authors use it to mean "knows whether P is true." Translating between the two representations is straightforward.

simply deletes the agent's knowledge of $CleanR$. The agent will also know whether $CleanL$ is true, one way or the other, because of local dirt sensing, and it will know that it is AtL:

$$\begin{aligned} &Action(Left, \text{PRECOND:}AtR, \\ &\quad \text{EFFECT:}K(AtL) \wedge \neg K(AtR) \wedge \textbf{when } CleanR\text{: } \neg K(CleanR) \wedge \\ &\qquad \textbf{when } CleanL\text{: } K(CleanL) \wedge \\ &\qquad \textbf{when } \neg CleanL\text{: } K(\neg CleanL)) \ . \end{aligned} \tag{12.2}$$

Notice that the preconditions and **when** conditions are plain propositions, not knowledge propositions. This is as it should be, because the outcomes of actions do depend on the actual world, but how do we check the truth of those conditions when all we have is the belief state? If the agent *knows* a proposition, say $K(AtR)$, in the current belief state, then the proposition must be true in the current physical state, and indeed the action is applicable. If the agent doesn't know a proposition—for example, the **when** condition $CleanL$—then the belief state must include worlds in which $CleanL$ is true and worlds in which $CleanL$ is false. It is this that gives rise to multiple belief states resulting from the action. Thus, if the initial state is $(K(AtR) \wedge K(CleanR))$, then after the move $Left$, the two outcome belief states are $(K(AtL) \wedge K(CleanL))$ and $(K(AtL) \wedge K(\neg CleanL))$. In both cases, the truth value of $CleanL$ is known, so the $CleanL$ test can be used in the plan.

With active sensing (as opposed to automatic sensing), the agent gets new percepts only by asking for them. Thus, after moving $Left$, the agent will not know whether the left-hand square is dirty, so the last two conditional effects no longer appear in the action description in Equation (12.2). To find out whether the square is dirty, the agent can $CheckDirt$:

$$\begin{aligned} &Action(CheckDirt, \text{EFFECT:}\textbf{when } AtL \wedge CleanL\text{: } K(CleanL) \wedge \\ &\qquad \textbf{when } AtL \wedge \neg CleanL\text{: } K(\neg CleanL) \wedge \\ &\qquad \textbf{when } AtR \wedge CleanR\text{: } K(CleanR) \wedge \\ &\qquad \textbf{when } AtR \wedge \neg CleanR\text{: } K(\neg CleanR)) \ . \end{aligned} \tag{12.3}$$

It is easy to show that $Left$ followed by $CheckDirt$ in the active sensing setting results in the same two belief states as $Left$ did in the automatic sensing setting. With active sensing, it is always the case that physical actions map a belief state into a single successor belief state. Multiple belief states can be introduced only by sensory actions, which provide specific knowledge and hence allow conditional tests to be used in plans.

We have described a general approach to conditional planning based on state-space AND–OR search. The approach has proved to be quite effective on some test problems, but other problems are intractable. Theoretically, it can be shown that conditional planning belongs to a harder complexity class than classical planning. Recall that the definition of the class NP is that a candidate solution can be checked to see whether it really is a solution in polynomial time. This is true for classical plans (at least, for those of polynomial size) so the problem of classical planning is in *NP*. But in conditional planning a candidate must be checked to see whether, for *all* possible states, there exists *some* path through the plan that satisfies the goal. Checking the "all/some" combination cannot be done in polynomial time, so conditional planning is harder than *NP*. The only way out is to ignore some of the possible contingencies during the planning phase and to handle them only when they actually occur. This is the approach we pursue in the next section.

12.5 EXECUTION MONITORING AND REPLANNING

An **execution monitoring** agent checks its percepts to see whether everything is going according to plan. Murphy's law tells us that even the best-laid plans of mice, men, and conditional planning agents frequently fail. The problem is unbounded indeterminacy—some unanticipated circumstance will always arise for which the agent's actions descriptions are incorrect. Therefore, execution monitoring is a necessity in realistic environments. We will consider two kinds of execution monitoring: a simple, but weak form called **action monitoring**, whereby the agent checks the environment to verify that the next action will work, and a more complex but more effective form called **plan monitoring**, in which the agent verifies the entire remaining plan.

A **replanning** agent knows what to do when something unexpected happens: call a planner again to come up with a new plan to reach the goal. To avoid spending too much time planning, this is usually done by trying to repair the old plan—to find a way from the current unexpected state back onto the plan.

As an example, let us return to the double Murphy vacuum world in Figure 12.9. In this world, moving into a clean square sometimes deposits dirt in that square; but what if the agent doesn't know that or doesn't worry about it? Then it will come up with a very simple solution: $[Left]$. If no dirt is dumped on arrival when the plan is actually executed, then the agent will detect the achievement of the goal. Otherwise, because the $CleanL$ precondition of the implicit $Finish$ step is not satisfied, the agent will generate a new plan: $[Suck]$. Execution of this plan always succeeds.

Together, execution monitoring and replanning form a general strategy that can be applied to both fully and partially observable environments, and to a variety of planning representations including state-space, partial-order, and conditional plans. One simple approach to state-space planning is shown in Figure 12.13. The planning agent starts with a goal and creates an initial plan to achieve it. The agent then starts executing actions one by one. The replanning agent, unlike our other planning agents, keeps track of both the remaining unexecuted plan segment $plan$ and the complete original plan $whole\_plan$. It uses **action monitoring**: before carrying out the next action of $plan$, the agent examines its percepts to see whether any preconditions of the plan have unexpectedly become unsatisfied. If they have, the agent will try to get back on track by replanning a sequence of actions that should take it back to some point in the $whole\_plan$.

ACTION MONITORING

Figure 12.14 provides a schematic illustration of the process. The replanner notices that the preconditions of the first action in $plan$ are not satisfied by the current state. It then calls the planner to come up with a new subplan called $repair$ that will get from the current situation to some state s on $whole\_plan$. In this example, the state s happens to be one step back from the current remaining $plan$. (That is why we keep track of the whole plan, rather than just the remaining plan.) In general, we choose s to be as close as possible to the current state. The concatenation of $repair$ and the portion of $whole\_plan$ from s onward, which we call $continuation$, makes up the new $plan$, and the agent is ready to resume execution.

```
function REPLANNING-AGENT(percept) returns an action
  static: KB, a knowledge base (includes action descriptions)
          plan, a plan, initially []
          whole_plan, a plan, initially []
          goal, a goal

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  current ← STATE-DESCRIPTION(KB, t)
  if plan = [] then
     whole_plan ← plan ← PLANNER(current, goal, KB)
  if PRECONDITIONS(FIRST(plan)) not currently true in KB then
     candidates ← SORT(whole_plan, ordered by distance to current)
     find state s in candidates such that
          failure ≠ repair ← PLANNER(current, s, KB)
     continuation ← the tail of whole_plan starting at s
     whole_plan ← plan ← APPEND(repair, continuation)
  return POP(plan)
```

Figure 12.13 An agent that does action monitoring and replanning. It uses a complete state-space planning algorithm called PLANNER as a subroutine. If the preconditions of the next action are not met, the agent loops through the possible points p in *whole_plan*, trying to find one that PLANNER can plan a path to. This path is called *repair*. If PLANNER succeeds in finding a repair, the agent appends *repair* and the tail of the plan after p, to create the new plan. The agent then returns the first step in the plan.

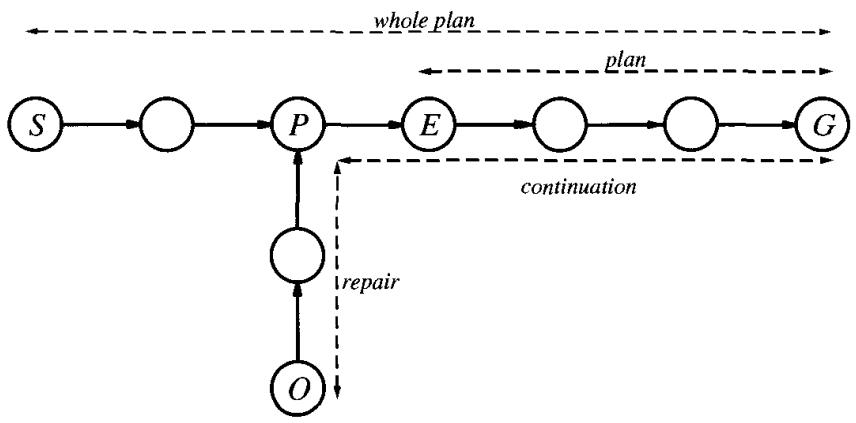


Figure 12.14 Before execution, the planner comes up with a plan, here called *whole_plan*, to get from S to G. The agent executes the plan until the point marked E. Before executing the remaining *plan*, it checks preconditions as usual and finds that it is actually in state O rather than state E. It then calls its planning algorithm to come up with *repair*, which is a plan to get from O to some point P on the original *whole_plan*. The new *plan* now becomes the concatenation of *repair* and *continuation* (the resumption of the original *whole_plan*).

Now let's return to the example problem of achieving a chair and table of matching color, this time via replanning. We'll assume a fully observable environment. In the initial state the chair is blue, the table is green, and there is a can of blue paint and a can of red paint. That gives us the following problem definition:

$$\begin{aligned}
&Init(Color(Chair, Blue) \wedge Color(Table, Green)\\
&\quad \wedge ContainsColor(BC, Blue) \wedge PaintCan(BC))\\
&\quad \wedge ContainsColor(RC, Red) \wedge PaintCan(RC)\\
&Goal(Color(Chair, x) \wedge Color(Table, x))\\
&Action(Paint(object, color),\\
&\quad \text{PRECOND:}HavePaint(color)\\
&\quad \text{EFFECT:}Color(object, color))\\
&Action(Open(can),\\
&\quad \text{PRECOND:}PaintCan(can) \wedge ContainsColor(can, color)\\
&\quad \text{EFFECT:}HavePaint(color)
\end{aligned}$$

The agent's PLANNER should come up with the following plan:

$$[Start; Open(BC); Paint(Table, Blue); Finish]$$

Now the agent is ready to execute the plan. Assume that all goes well as the agent opens the blue paint and applies it to the table. The agents from previous sections would declare victory at this point, having completed the steps in the plan. But the execution monitoring agent must first check the precondition of the *Finish* step, which says that the two pieces must have the same color. Suppose the agent perceives that they do not have the same color, because it missed a spot of green on the table. The agent then needs to figure out a position in *whole_plan* to aim for and a repair action sequence to get there. The agent notices that the current state is identical to the precondition before the *Paint* action, so the agent chooses the empty sequence for *repair* and makes its *plan* be the same $[Paint, Finish]$ sequence that it just attempted. With this new plan in place, execution monitoring resumes, and the *Paint* action is retried. This behavior will loop until the table is perceived to be completely painted. But notice that the loop is created by a process of plan–execute–replan, rather than by an explicit loop in a plan.

Action monitoring is a very simple method of execution monitoring but it can sometimes lead to less than intelligent behavior. For example, suppose that the agent constructs a plan to solve the painting problem by painting the chair and table red. Then it opens the can of red paint and finds that there is only enough paint for the chair. Action monitoring would not detect failure until *after* the chair has been painted, at which point $HavePaint(Red)$ becomes false. What we really need to do is detect failure whenever the state is such that the remaining plan no longer works. **Plan monitoring** achieves this by checking the preconditions for success of the entire remaining plan—that is, the preconditions of each step in the plan, except those preconditions that are achieved by another step in the remaining plan. Plan monitoring cuts off execution of a doomed plan as soon as possible, rather than continuing until the failure actually occurs.[13] In some cases, it can rescue the agent from disaster when the doomed plan would have led to a dead end from which the goal would be unachievable.

PLAN MONITORING

[13] Plan monitoring makes our agent smarter than a dung beetle. (See page 37.) Our agent would notice that the

It is relatively straightforward to modify a planning algorithm so that it annotates the plan at each point with the preconditions for success of the remaining plan. If we extend plan monitoring to check whether the current state satisfies the plan preconditions at any future point, rather than just the current point, then plan monitoring will also be able to take advantage of **serendipity**—that is, accidental success. If someone comes along and paints the table red at the same time that the agent is painting the chair red, then the final plan preconditions are satisfied (the goal has been achieved), and the agent can go home early.

So far, we have described monitoring and replanning in fully observable environments. Things can become much more complicated when the environment is partially observable. First, things can go wrong without the agent's being able to detect it. Second, "checking preconditions" could require the execution of sensing actions, which have to be planned for—either at planning time, which takes us back to conditional planning, or at execution time. In the worst case, the execution of a sensing action could require a complex plan that itself requires monitoring and hence further sensing actions, and so on. If the agent insists on checking every precondition, it might never get around to actually *doing* anything. The agent should prefer to check those variables that are important, have a good chance of going wrong, and are not too expensive to perceive. This allows the agent to respond appropriately to important threats, but not waste time checking to see whether the sky is falling.

Now that we have described a method for monitoring and replanning, we need to ask, "Does it work?" This is a surprisingly tricky question. If we mean, "Can we guarantee that the agent will always achieve the goal, even with unbounded indeterminacy?" then the answer is no, because the agent could inadvertently arrive at a dead end, as described for online search in Section 4.5. For example, the vacuum agent might not know that its batteries can run out. Let's rule out dead ends; that is, let's assume that the agent can construct a plan to reach the goal from *any* state in the environment. If we assume that the environment is really nondeterministic, in the sense that such a plan always has *some* chance of success on any given execution attempt, then the agent will eventually reach the goal. The replanning agent therefore has capabilities analogous to those of the conditional planning agent. In fact, we can modify a conditional planner so that it constructs only a partial solution plan that includes steps of the form "**if** $<test>$ **then** $plan\_A$ **else** $replan$." Under the assumptions we have made, such a plan can be a correct solution to the original problem; it might also be much cheaper to construct than a full conditional plan.

Trouble occurs when the agent's repeated attempts to reach the goal are futile—when they are blocked by some precondition or effect that it doesn't know about. For example, if the agent has the wrong card key to its hotel room, no amount of inserting and removing it is going to open the door.[14] One solution is to choose randomly from among the set of possible repair plans, rather than trying the same one each time. In this case, the repair plan of going to the front desk and getting a card key to the room would be a useful alternative. Given that the agent might not be able to distinguish between the truly nondeterministic case and the futile case, some variation in repairs is a good idea in general.

---

dung ball was missing from its grasp and would replan to get another ball and plug its hole.

[14] Futile repetition of a plan repair is exactly the behavior exhibited by the sphex wasp. (See page 37.)

Another solution to the problem of incorrect action descriptions is **learning**. After a few tries, a learning agent should be able to modify the action description that says that the key opens the door. At that point, the replanner will automatically come up with an alternative plan, such as getting a new key. This kind of learning is described in Chapter 21.

Even with all these potential improvements, the replanning agent still has a few shortcomings. It cannot perform in real-time environments, and there is no bound on the amount of time it will spend replanning and thus no bound on the time it takes to decide on an action. Also, it cannot formulate new goals of its own or accept new goals in addition to its current goals, so it cannot be a long-lived agent in a complex environment. These shortcomings will be addressed in the next section.

12.6 CONTINUOUS PLANNING

In this section, we design an agent that persists indefinitely in an environment. Thus it is not a "problem solver" that is given a single goal and then plans and acts until the goal is achieved; rather, it lives through a series of ever-changing goal formulation, planning, and acting phases. Rather than thinking of the planner and execution monitor as separate processes, one of which passes its results to the other, we can think of them as a single process in a **continuous planning agent**.

CONTINUOUS PLANNING AGENT

The agent is thought of as always being *part of the way through* executing a plan—the grand plan of living its life. Its activities include executing some steps of the plan that are ready to be executed, refining the plan to satisfy open preconditions or resolve conflicts, and modifying the plan in the light of additional information obtained during execution. Obviously, when it first formulates a new goal, the agent will have no actions ready to execute, so it will spend a while generating a partial plan. It is quite possible, however, for the agent to begin execution before the plan is complete, especially when it has independent subgoals to achieve. The continuous planning agent monitors the world continuously, updating its world model from new percepts even if its deliberations are still continuing.

We will first go through an example and then describe the agent program, which we will call CONTINUOUS-POP-AGENT because it uses partial-order plans to represent its intended activities. To simplify the presentation, we will assume a fully observable environment. The same techniques can be extended to the partially observable case.

The example we will use is a problem from the blocks world domain (Section 11.1). The start state is shown in Figure 12.15(a). The action we will need is $Move(x, y)$, which moves block x onto block y, provided that both are clear. Its action schema is

$$\begin{array}{l} Action(Move(x, y), \\ \quad \text{PRECOND:} Clear(x) \wedge Clear(y) \wedge On(x, z), \\ \quad \text{EFFECT:} On(x, y) \wedge Clear(z) \wedge \neg On(x, z) \wedge \neg Clear(y)) \ . \end{array}$$

The agent first needs to formulate a goal for itself. We won't discuss goal formulation here, but instead we will assume that somehow the agent was told (or decided on its own) to achieve the goal $On(C, D) \wedge On(D, B)$. The agent starts planning for this goal. Unlike

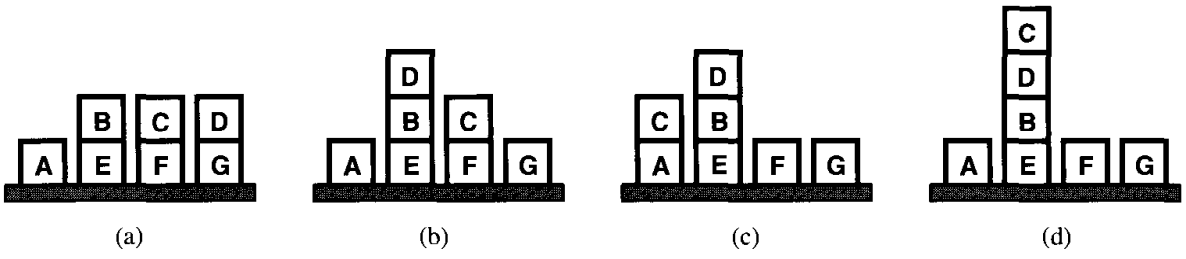


Figure 12.15 The sequence of states as the continuous planning agent tries to reach the goal state $On(C, D) \wedge On(D, B)$, as shown in (d). The start state is (a). At (b), another agent has interfered, putting D on B. At (c), the agent has executed $Move(C, D)$ but has failed, dropping C on A instead. It retries $Move(C, D)$, reaching the goal state (d).

all our other agents, which would shut off their percepts until the planner returns a complete solution to this problem, the continuous planning agent builds the plan incrementally, with each increment taking a bounded amount of time. After each increment, the agent returns $NoOp$ as its action and checks its percepts again. We assume that the percepts don't change and the agent quickly constructs the plan shown in Figure 12.16. Notice that although the preconditions of both actions are satisfied by $Start$, there is an ordering constraint putting $Move(D, B)$ before $Move(C, D)$. This is needed to ensure that $Clear(D)$ remains true until $Move(D, B)$ is completed. Throughout the continuous planning process, $Start$ is always used as the label for the current state. The agent updates the state after each action.

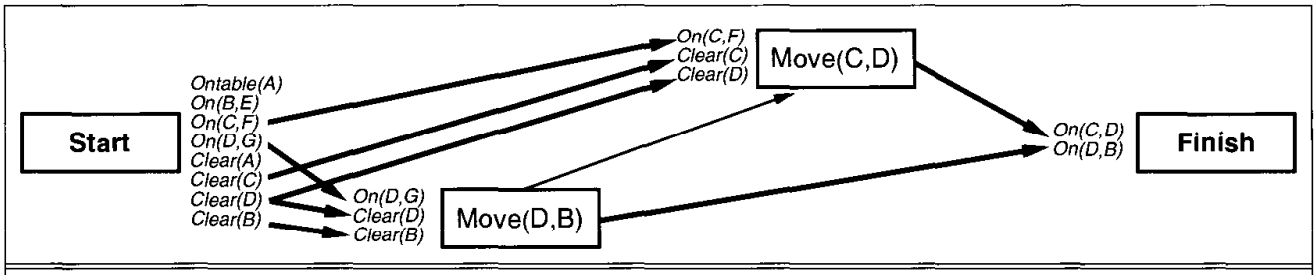


Figure 12.16 The initial plan constructed by the continuous planning agent. The plan is indistinguishable, so far, from that produced by a normal partial-order planner.

The plan is now ready to be executed, but before the agent can take action, nature intervenes. An external agent (perhaps the agent's teacher getting impatient) moves D onto B and the world is now in the state shown in Figure 12.15(b). The agent perceives this, recognizes that $Clear(B)$ and $On(D, G)$ are no longer true in the current state, and updates its model of the current state accordingly. The causal links that were supplying the preconditions $Clear(B)$ and $On(D, G)$ for the $Move(D, B)$ action become invalid and must be removed from the plan. The new plan is shown in Figure 12.17. At all times, $Start$ represents the current state, so this $Start$ is different from the one in the previous figure. Notice that the plan is now incomplete: two of the preconditions for $Move(D, B)$ are open, and its precondition $On(D, y)$ is now uninstantiated, because there is no longer any reason to assume the that move will be from G.

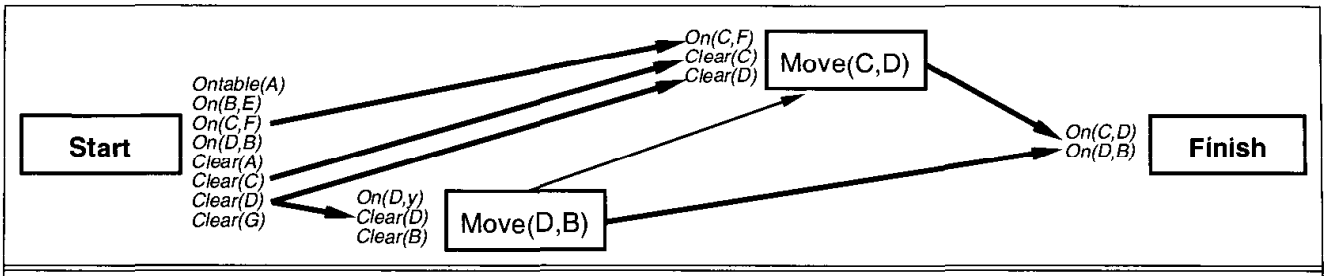


Figure 12.17 After someone else moves D onto B, the unsupported links supplying $Clear(B)$ and $On(D, G)$ are dropped, producing this plan.

Now the agent can take advantage of the "helpful" interference by noticing that the causal link $Move(D, B) \xrightarrow{On(D,B)} Finish$ can be replaced by a direct link from *Start* to *Finish*. This process is called **extending** a causal link and is done whenever a condition can be supplied by an earlier step instead of a later one without causing a new conflict.

EXTENSION

Once the old causal link from $Move(D, B)$ to *Finish* is removed, $Move(D, B)$ no longer supplies any causal links at all. It is now a **redundant step**. All redundant steps, and any links supplying them, are dropped from the plan. This gives the plan in Figure 12.18.

REDUNDANT STEP

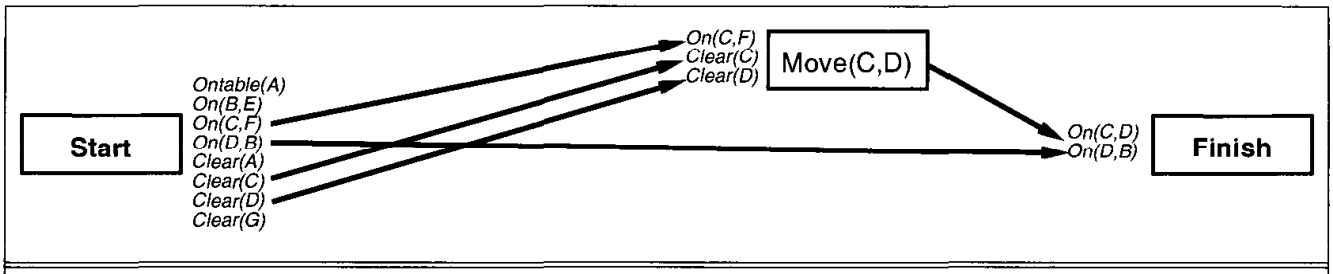


Figure 12.18 The link supplied by $Move(D, B)$ has been replaced by one from *Start*, and the now-redundant step $Move(D, B)$ has been dropped.

Now the step $Move(C, D)$ is ready to be executed, because all of its preconditions are satisfied by the *Start* step, no other steps are necessarily before it, and it does not conflict with any other link in the plan. The step is removed from the plan and executed. Unfortunately, the agent is clumsy and drops C onto A instead of D, giving the state shown in Figure 12.15(c). The new plan state is shown in Figure 12.19. Notice that although there are now no actions in the plan, there is still an open condition for the *Finish* step.

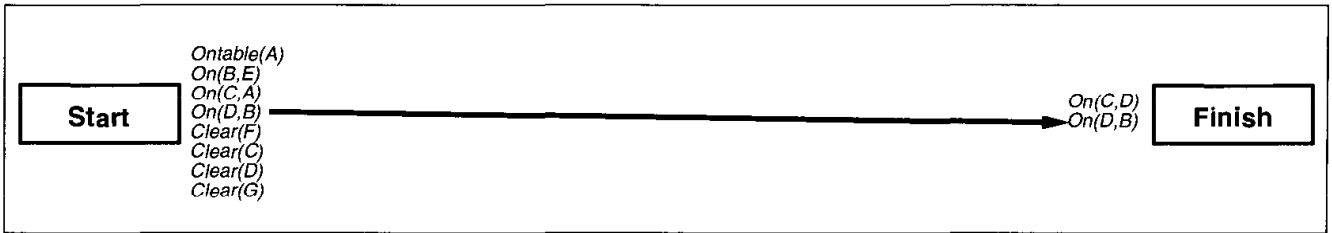


Figure 12.19 After $Move(C, D)$ is executed and removed from the plan, the effects of the *Start* step reflect the fact that C ended up on A instead of the intended D. The goal precondition $On(C, D)$ is still open.

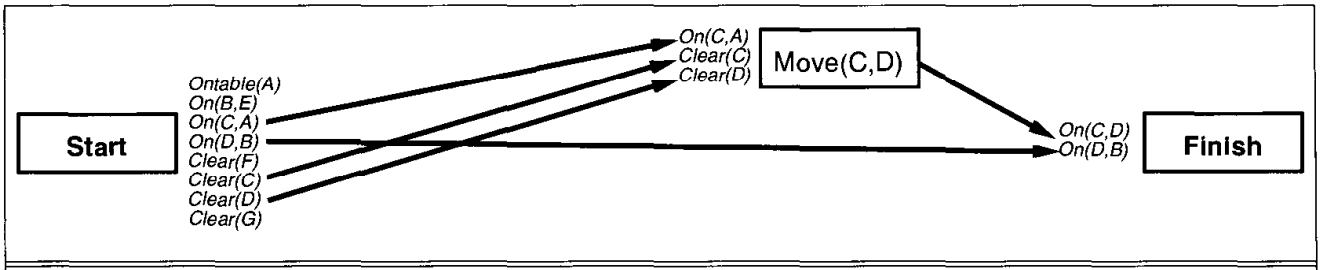


Figure 12.20 The open condition is resolved by adding $Move(C, D)$ back in. Notice the new bindings for the preconditions.

The agent decides to plan for the open condition. Once again, $Move(C, D)$ will satisfy the goal condition. Its preconditions are satisfied by new causal links from the *Start* step. The new plan appears in Figure 12.20.

Once again, $Move(C, D)$ is ready for execution. This time it works, resulting in the goal state shown in Figure 12.15(d). Once the step is dropped from the plan, the goal condition $On(C, D)$ becomes open again. Because the *Start* step is updated to reflect the new world state, however, the goal condition can be satisfied immediately by a link from the *Start* step. This is the normal course of events when an action is successful. The final plan state is shown in Figure 12.21. Because all the goal conditions are satisfied by the *Start* step and there are no remaining actions, the agent is now free to remove the goals from *Finish* and formulate a new goal.

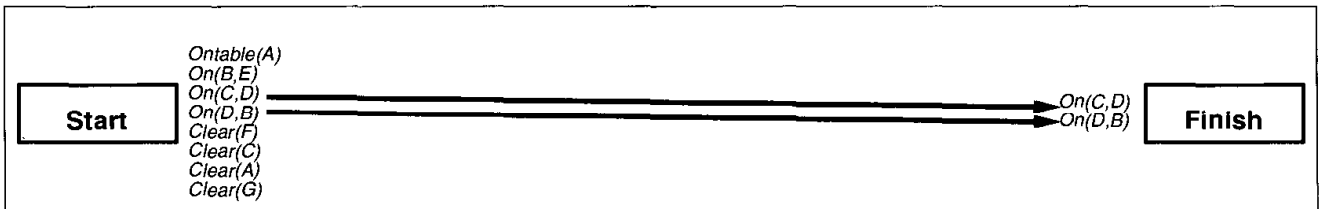


Figure 12.21 After $Move(C, D)$ is executed and dropped from the plan, the remaining open condition $On(C, D)$ is resolved by adding a causal link from the new *Start* step. The plan is now completed.

From this example, we can see that continuous planning is quite similar to partial-order planning. On each iteration, the algorithm finds something about the plan that needs fixing—
PLAN FLAW a so-called **plan flaw**—and fixes it. The POP algorithm can be seen as a flaw-removal algorithm where the two flaws are open preconditions and causal conflicts. The continuous planning agent, on the other hand, addresses a much broader range of flaws:

- *Missing goal:* The agent can decide to add a new goal or goals to the *Finish* state. (Under continuous planning, it might make more sense to change the name of *Finish* to *Infinity*, and of *Start* to *Current*, but we will stick with tradition.)
- *Open precondition:* Add a causal link to an open precondition, choosing either a new or an existing action (as in POP).
- *Causal Conflict:* Given a causal link $A \xrightarrow{p} B$ and an action C with effect $\neg p$, choose an ordering constraint or variable constraint to resolve the conflict (as in POP).

- *Unsupported link:* If there is a causal link $Start \xrightarrow{p} A$ where p is no longer true in *Start*, then remove the link. (This prevents us from executing an action whose preconditions are false.)
- *Redundant action:* If an action A supplies no causal links, remove it and its links. (This allows us to take advantage of serendipitous events.)
- *Unexecuted action:* If an action A (other than *Finish*) has its preconditions satisfied in *Start*, has no other actions (besides *Start*) ordered before it, and conflicts with no causal links, then remove A and its causal links and return it as the action to be executed.
- *Unnecessary historical goal:* If there are no open preconditions and no actions in the plan (so that all causal links go directly from *Start* to *Finish*), then we have achieved the current goal set. Remove the goals and the links to them to allow for new goals.

The CONTINUOUS-POP-AGENT is shown in Figure 12.22. It has a cycle of "perceive, remove flaw, act." It keeps a persistent plan in its knowledge base, and on each turn it removes one flaw from the plan. It then takes an action (although often the action will be *NoOp*) and repeats the loop. This agent can handle many of the problems listed in the discussion of the replanning agent on page 445. In particular, it can act in real time, it handles serendipity, it can formulate its own goals, and it can handle unexpected events that affect future plans.

```
function CONTINUOUS-POP-AGENT(percept) returns an action
  static: plan, a plan, initially with just Start, Finish

  action ← NoOp (the default)
  EFFECTS[Start] = UPDATE(EFFECTS[Start], percept)
  REMOVE-FLAW(plan) // possibly updating action
  return action
```

Figure 12.22 CONTINUOUS-POP-AGENT, a continuous partial-order planning agent. After receiving a percept, the agent removes a flaw from its constantly updated plan and then returns an action. Often it will take many steps of flaw-removal planning, during which it returns *NoOp*, before it is ready to take a real action.

12.7 MULTIAGENT PLANNING

So far we have dealt with **single-agent environments**, in which our agent is alone. When there are other agents in the environment, our agent could simply include them in its model of the environment, without changing its basic algorithms. In many cases, however, that would lead to poor performance because dealing with other agents is not the same as dealing with nature. In particular, nature is (one assumes) indifferent to the agent's intentions,[15] whereas other agents are not. This section introduces multiagent planning to handle these issues.

[15] Residents of the United Kingdom, where the mere act of planning a picnic guarantees rain, might disagree.

$Agents(A, B)$
$Init(At(A, [Left, Baseline]) \wedge At(B, [Right, Net]) \wedge$
$Approaching(Ball, [Right, Baseline])) \wedge Partner(A, B) \wedge Partner(B, A)$
$Goal(Returned(Ball) \wedge At(agent, [x, Net]))$
$Action(Hit(agent, Ball),$
PRECOND:$Approaching(Ball, [x, y]) \wedge At(agent, [x, y]) \wedge$
$Partner(agent, partner) \wedge \neg At(partner, [x, y])$
EFFECT:$Returned(Ball))$
$Action(Go(agent, [x, y]),$
PRECOND:$At(agent, [a, b])$,
EFFECT:$At(agent, [x, y]) \wedge \neg At(agent, [a, b]))$

Figure 12.23 The doubles tennis problem. Two agents are playing together and can be in one of four locations: $[Left, Baseline]$, $[Right, Baseline]$, $[Left, Net]$, and $[Right, Net]$. The ball can be returned if exactly one player is in the right place.

We saw in Chapter 2 that multiagent environments can be **cooperative** or **competitive**. We will begin with a simple cooperative example: team planning in doubles tennis. Plans can be constructed that specify actions for both players on the team; we will describe techniques for constructing such plans efficiently. Efficient plan construction is useful, but does not guarantee success; the agents have to agree to use the same plan! This requires some form of **coordination**, possibly achieved by **communication**.

Cooperation: Joint goals and plans

Two agents playing on a doubles tennis team have the joint goal of winning the match, which gives rise to various subgoals. Let's suppose that at one point in the game, they have the joint goal of returning the ball that has been hit to them and ensuring that at least one of them is covering the net. We can represent this notion as a **multiagent planning** problem, as shown in Figure 12.23.

MULTIAGENT PLANNING

This notation introduces two new features. First, $Agents(A, B)$ declares that there are two agents, A and B, who are participating in the plan. (For this problem the opposing players are not considered agents.) Second, each action explicitly mentions the agent as a parameter, because we need to keep track of which agent does what.

JOINT PLAN

A solution to a multiagent planning problem is a **joint plan** consisting of actions for each agent. A joint plan is a solution if the goal will be achieved when each agent performs its assigned actions. The following plan is a solution to the tennis problem:

PLAN 1:
$$A: [Go(A, [Right, Baseline]), Hit(A, Ball)]$$
$$B: [NoOp(B), NoOp(B)] .$$

If both agents have the same knowledge base, and if this is the only solution, then everything would be fine; the agents could each determine the solution and then jointly execute it. Unfortunately for the agents (and we will soon see why it's unfortunate), there is another plan

that satisfies the goal just as well as the first:

PLAN 2:
$$A: [Go(A, [Left, Net]), NoOp(A)]$$
$$B: [Go(B, [Right, baseline]), Hit(B, Ball)] .$$

If A chooses plan 2 and B chooses plan 1, then nobody will return the ball. Conversely, if A chooses 1 and B chooses 2, then they will probably collide with each other; no one returns the ball and the net may remain uncovered. Hence, the existence of correct joint plans does not mean that the goal will be achieved. The agents need a mechanism for **coordination** to reach the *same* joint plan; moreover, it must be common knowledge (see Chapter 10) among the agents that some particular joint plan will be executed.

COORDINATION

Multibody planning

This section concentrates on the construction of correct joint plans, deferring the coordination issue for the time being. We call this **multibody planning**; it is essentially the planning problem faced by a single centralized agent that can dictate actions to each of several physical entities. In the truly multiagent case, it enables each agent to figure out what the possible joint plans are that would succeed if executed jointly.

MULTIBODY PLANNING

Our approach to multibody planning will be based on partial-order planning, as described in Section 11.3. We will assume full observability, to keep things simple. There is one additional issue that doesn't arise in the single-agent case: the environment is no longer truly **static**, because other agents could act while any particular agent is deliberating. Therefore, we need to be concerned about **synchronization**. For simplicity, we will assume that each action takes the same amount of time and that actions at each point in the joint plan are simultaneous.

SYNCHRONIZATION

At any point in time, each agent is executing exactly one action (perhaps including *NoOp*). This set of concurrent actions is called a **joint action**. For example, a joint action in the tennis domain (page 450) with two agents A and B is $\langle NoOp(A), Hit(B, Ball)\rangle$. A joint plan consists of a partially ordered graph of joint actions. For example, Plan 2 for the tennis problem can be represented as this sequence of joint actions:

JOINT ACTION

$$\langle Go(A, [Left, Net]), Go(B, [Right, baseline])\rangle$$
$$\langle NoOp(A), Hit(B, Ball)\rangle$$

We *could* do planning using the regular POP algorithm, applied to the set of all possible joint actions. The only problem is the size of this set: with 10 actions and 5 agents we get 10^5 joint actions. It would be tedious to specify the preconditions and effects of each action correctly, and inefficient to do planning with such a large set.

An alternative is to define joint actions implicitly, by describing how each individual action interacts with other possible actions. This will be simpler, because most actions are independent of most others; we need list only the few actions that actually interact. We can do that by augmenting the usual STRIPS or ADL action descriptions with one new feature: a **concurrent action list**. This is similar to the precondition of an action description except that rather than describing state variables, it describes actions that must or must not be executed

CONCURRENT ACTION LIST

concurrently. For example, the *Hit* action could be described as follows:

$$\begin{aligned}&Action(Hit(A, Ball),\\ &\quad \text{CONCURRENT:}\neg Hit(B, Ball)\\ &\quad \text{PRECOND:}Approaching(Ball, [x, y]) \wedge At(A, [x, y])\\ &\quad \text{EFFECT:}Returned(Ball))\ .\end{aligned}$$

Here, we have the prohibited-concurrency constraint that, during the execution of the *Hit* action, there can be no other *Hit* action by another agent. We can also *require* concurrent action, for example when two agents are needed to carry a cooler full of beverages to the tennis court. The description for this action says that agent A cannot execute a *Carry* action unless there is another agent B who is simultaneously executing a *Carry* of the same cooler:

$$\begin{aligned}&Action(Carry(A, cooler, here, there),\\ &\quad \text{CONCURRENT:}Carry(B, cooler, here, there)\\ &\quad \text{PRECOND:}At(A, here) \wedge At(cooler, here) \wedge Cooler(cooler)\\ &\quad \text{EFFECT:}At(A, there) \wedge At(cooler, there) \wedge \neg At(A, here) \wedge \neg At(cooler, here)).\end{aligned}$$

With this representation, it is possible to create a planner that is very close to the POP partial-order planner. There are three differences:

1. In addition to the temporal ordering relation $A \prec B$, we allow $A = B$ and $A \preceq B$, meaning "concurrent" and "before or concurrent," respectively.
2. When a new action has required concurrent actions, we must instantiate those actions, using new or existing actions in the plan.
3. Prohibited concurrent actions are an additional source of constraints. Each constraint must be resolved by constraining conflicting actions to be before or after.

This representation gives us the equivalent of POP for multibody domains. We could extend this approach with the refinements of the last two chapters—HTNs, partial observability, conditionals, execution monitoring, and replanning—but that is beyond the scope of this book.

Coordination mechanisms

CONVENTION

The simplest method by which a group of agents can ensure agreement on a joint plan is to adopt a **convention** prior to engaging in joint activity. A convention is any constraint on the selection of joint plans, beyond the basic constraint that the joint plan must work if all agents adopt it. For example, the convention "stick to your side of the court" would cause the doubles partners to select plan 2, whereas the convention "one player always stays at the net" would lead them to plan 1. Some conventions, such as driving on the proper side of the road, are so widely adopted that they are considered **social laws**. Human languages can also be viewed as conventions.

SOCIAL LAWS

The conventions in the preceding paragraph are domain-specific and can be implemented by constraining the action descriptions to rule out violations of the convention. A more general approach is to use domain-independent conventions. For example, if each agent runs the same multibody planning algorithm with the same inputs, it can follow the convention of executing the first feasible joint plan found, confident that the other agents will come

to the same choice. A more robust but more expensive strategy would be to generate all joint plans and then pick the one, say, whose printed representation is alphabetically first.

Conventions can also arise through evolutionary processes. For example, colonies of social insects execute very elaborate joint plans, which are facilitated by the common genetic makeup of the individuals in the colony. Conformity can also be enforced by the fact that deviation from conventions reduces evolutionary fitness, so that any feasible joint plan can become a stable equilibrium. Similar considerations apply to the development of human language, where the important thing is not which language each individual should speak, but the fact that all individuals speak the same language. One final example appears in the flocking behavior of birds. We can obtain a reasonable simulation if each bird agent (sometimes called a birdoid or **boid**) executes the following three rules with some method of combination:

BOID

1. Separation: Steer away from neighbors when you start to get too close.
2. Cohesion: Steer towards the average position of the neighbors.
3. Alignment: Steer towards the average orientation (heading) of the neighbors.

EMERGENT BEHAVIOR

If all the birds execute the same policy, the flock exhibits the **emergent behavior** of flying as a pseudo-rigid body with roughly constant density that does not disperse over time. As with insects, there is no need for each agent to possess the joint plan that models the actions of other agents.

Typically, conventions are adopted to cover a universe of individual multiagent planning problems, rather than being developed anew for each problem. This can lead to inflexibility and breakdown, as can be seen sometimes in doubles tennis when the ball is roughly equidistant between the two partners. In the absence of an applicable convention, agents can use **communication** to achieve common knowledge of a feasible joint plan. For example, a doubles tennis player could shout "Mine!" or "Yours!" to indicate a preferred joint plan. We cover mechanisms for communication in more depth in Chapter 22, where we observe that communication does not necessarily involve a verbal exchange. For example, one player can communicate a preferred joint plan to the other simply by executing the first part of it. In our tennis problem, if agent A heads for the net, then agent B is obliged to go back to the baseline to hit the ball, because plan 2 is the only joint plan that begins with A's heading for the net. This approach to coordination, sometimes called **plan recognition**, works when a single action (or short sequence of actions) is enough to determine a joint plan unambiguously.

PLAN RECOGNITION

The burden for ensuring that the agents arrive at a successful joint plan can be placed either on the agent designers or on the agents themselves. In the former case, before the agents begin to plan, the agent designer should prove that the agents' policies and strategies will be successful. The agents themselves can be reactive if that works for the environment they exist in, and they need not have explicit models about the other agents. In the latter case, the agents are deliberative; they must prove or otherwise demonstrate that their plan will be effective, taking the other agents' reasoning into account. For example, in an environment with two logical agents A and B, they could both have the following definition:

$$\forall p, s \;\; Feasible(p, s) \Leftrightarrow CommonKnowledge(\{A, B\}, Achieves(p, s, Goal))$$

This says that in any situation s, the plan p is a feasible joint plan in that situation if it is common knowledge among the agents that p will achieve the goal. We need further axioms

JOINT INTENTION

to establish common knowledge of a **joint intention** to execute a *particular* joint plan; only then can agents begin to act.

Competition

COMPETITION

Not all multiagent environments involve cooperative agents. Agents with conflicting utility functions are in **competition** with each other. One example of this is two-player zero-sum games, such as chess. We saw in Chapter 6 that a chess-playing agent needs to consider the opponent's possible moves for several steps into the future. That is, an agent in a competitive environment must (a) recognize that there are other agents, (b) compute some of the other agent's possible plans, (c) compute how the other agent's plans interact with its own plans, and (d) decide on the best action in view of these interactions. So competition, like cooperation, requires a model of the other agent's plans. On the other hand, there is no commitment to a joint plan in a competitive environment.

Section 12.4 drew the analogy between games and conditional planning problems. The conditional planning algorithm in Figure 12.10 constructs plans that work under worst-case assumptions about the environment, so it can be applied in competitive situations where the agent is concerned only with success and failure. When the agent and its opponents are concerned about the *cost* of a plan, then minimax is appropriate. As yet, there has been little work on combining minimax with methods, such as POP and HTN planning, that go beyond the state-space search model used in Chapter 6. We will return to the question of competition in Section 17.6, which covers game theory.

12.8 SUMMARY

This chapter has addressed some of the complications of planning and acting in the real world. The main points are:

- Many actions consume **resources**, such as money, gas, or raw materials. It is convenient to treat these resources as numeric measures in a pool rather than try to reason about, say, each individual coin and bill in the world. Actions can generate and consume resources, and it is usually cheap and effective to check partial plans for satisfaction of resource constraints before attempting further refinements.
- Time is one of the most important resources. It can be handled by specialized scheduling algorithms, or scheduling can be integrated with planning.
- **Hierarchical task network** (HTN) planning allows the agent to take advice from the domain designer in the form of decomposition rules. This makes it feasible to create the very large plans required by many real-world applications.
- Standard planning algorithms assume complete and correct information and deterministic, fully observable environments. Many domains violate this assumption.
- Incomplete information can be dealt with by planning to use sensing actions to obtain the information needed. **Conditional plans** allow the agent to sense the world during

execution to decide what branch of the plan to follow. In some cases, **sensorless** or **conformant planning** can be used to construct a plan that works without the need for perception. Both sensorless and conditional plans can be constructed by search in the space of **belief states**.

- Incorrect information results in unsatisfied preconditions for actions and plans. **Execution monitoring** detects violations of the preconditions for successful completion of the plan.
- A **replanning agent** uses execution monitoring and splices in repairs as needed.
- A **continuous planning** agent creates new goals as it goes and reacts in real time.
- **Multiagent** planning is necessary when there are other agents in the environment with which to cooperate, compete, or coordinate. **Multibody** planning constructs joint plans, using an efficient decomposition of joint action descriptions, but must be augmented with some form of coordination if two cooperative agents are to agree on which joint plan to execute.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Planning with continuous time was first dealt with by DEVISER (Vere, 1983). The issue of systematic representation of time in plans was addressed by Dean *et al.* (1990) in the FORBIN system. NONLIN+ (Tate and Whiter, 1984) and SIPE (Wilkins, 1988, 1990) could reason about the allocation of limited resources to various plan steps. O-PLAN (Bell and Tate, 1985), an HTN planner, had a uniform, general representation for constraints on time and resources. In addition to the Hitachi application mentioned in the text, O-PLAN has been applied to software procurement planning at Price Waterhouse and back-axle assembly planning at Jaguar Cars. A number of hybrid planning-and-scheduling systems have been deployed: ISIS (Fox *et al.*, 1982; Fox, 1990) has been used for job shop scheduling at Westinghouse, GARI (Descotte and Latombe, 1985) planned the machining and construction of mechanical parts, FORBIN was used for factory control, and NONLIN+ was used for naval logistics planning.

After an initial flurry of theoretical work in the late 1980s, temporal planning made a comeback recently, when new algorithms and increased processing power made it feasible to attack practical applications. The two planners SAPA (Do and Kambhampati, 2001) and T4 (Haslum and Geffner, 2001) both used forward state-space search with sophisticated heuristics to handle actions with durations and resources. An alternative is to use very expressive action languages, but guide them by human-written domain-specific heuristics, as is done by ASPEN (Fukunaga *et al.*, 1997), HSTS (Jonsson *et al.*, 2000), and IxTeT (Ghallab and Laruelle, 1994).

There is a long history of scheduling in aerospace. T-SCHED (Drabble, 1990) was used to schedule mission-command sequences for the UOSAT-II satellite. OPTIMUM-AIV (Aarup *et al.*, 1994) and PLAN-ERS1 (Fuchs *et al.*, 1990), both based on O-PLAN, were used for spacecraft assembly and observation planning, respectively, at the European Space Agency.

SPIKE (Johnston and Adorf, 1992) was used for observation planning at NASA for the Hubble Space Telescope, while the Space Shuttle Ground Processing Scheduling System (Deale *et al.*, 1994) does job-shop scheduling of up to 16,000 worker-shifts. Remote Agent (Muscettola *et al.*, 1998) became the first autonomous planner-scheduler to control a spacecraft when it flew onboard the Deep Space One probe in 1999. The literature on job-shop scheduling in operations research is surveyed by Vaessens *et al.* (1996); theoretical results are presented by Martin and Shmoys (1996).

MACROPS

The facility in the STRIPS program for learning **macrops**—"macro-operators" consisting of a sequence of primitive steps—could be considered the first mechanism for hierarchical planning (Fikes *et al.*, 1972). Hierarchy was also used in the LAWALY system (Siklossy and Dreussi, 1973). The ABSTRIPS system (Sacerdoti, 1974) introduced the idea of an **abstraction hierarchy**, whereby planning at higher levels was permitted to ignore lower-level preconditions of actions in order to derive the general structure of a working plan. Austin Tate's Ph.D. thesis (1975b) and work by Earl Sacerdoti (1977) developed the basic ideas of HTN planning in its modern form. Many practical planners, including O-PLAN and SIPE, are HTN planners. Yang (1990) discusses properties of actions that make HTN planning efficient. Erol, Hendler, and Nau (1994, 1996) present a complete hierarchical decomposition planner as well as a range of complexity results for pure HTN planners. Other authors (Ambros-Ingerson and Steel, 1988; Young *et al.*, 1994; Barrett and Weld, 1994; Kambhampati *et al.*, 1998) have proposed the hybrid approach taken in this chapter, in which decompositions are just another form of refinement that can be used in partial-order planning.

ABSTRACTION HIERARCHY

Beginning with the work on macro-operators in STRIPS, one of the goals of hierarchical planning has been the reuse of previous planning experience in the form of generalized plans. The technique of **explanation-based learning**, described in depth in Chapter 19, has been applied in several systems as a means of generalizing previously computed plans, including SOAR (Laird *et al.*, 1986) and PRODIGY (Carbonell *et al.*, 1989). An alternative approach is to store previously computed plans in their original form and then reuse them to solve new, similar problems by analogy to the original problem. This is the approach taken by the field called **case-based planning** (Carbonell, 1983; Alterman, 1988; Hammond, 1989). Kambhampati (1994) argues that case-based planning should be analyzed as a form of refinement planning and provides a formal foundation for case-based partial-order planning.

CASE-BASED PLANNING

The unpredictability and partial observability of real environments was recognized early on in robotics projects that used planning techniques, including Shakey (Fikes *et al.*, 1972) and FREDDY (Michie, 1974). The problem received more attention after the publication of McDermott's (1978a) influential article, *Planning and Acting*.

Early planners, which lacked conditionals and loops, did not explicitly recognize the concept of conditional planning; but nevertheless they sometimes resorted to a coercive style in response to environmental uncertainty. Sacerdoti's NOAH used coercion in its solution to the "keys and boxes" problem, a planning challenge problem in which the planner knows little about the initial state. Mason (1993) argued that sensing often can and should be dispensed with in robotic planning, and described a sensorless plan that can move a tool into a specific position on a table by a sequence of tilting actions, *regardless* of the initial position. We describe this idea in the context of robotics. (See Figure 25.17.)

Goldman and Boddy (1996) introduced the term **conformant planning** for sensorless planners that handle uncertainty by coercing the world into known states, noting that sensorless plans are often effective even if the agent has sensors. The first moderately efficient conformant planner was Smith and Weld's (1998) Conformant Graphplan or CGP. Ferraris and Giunchiglia (2000) and Rintanen (1999) independently developed SATplan-based conformant planners. Bonet and Geffner (2000) describe a conformant planner based on heuristic search in the space of belief states, drawing on ideas first developed in the 1960s for partially observable Markov decision processes, or POMDPs (see Chapter 17). Currently, the fastest belief-state conformant planners, such as HSCP (Bertoli *et al.*, 2001a), use binary decision diagrams (BDDs) (Bryant, 1992) to represent belief states and are up to five orders of magnitude faster than CGP.

WARPLAN-C (Warren, 1976), a variant of WARPLAN, was one of the earliest planners to use conditional actions. Olawski and Gini (1990) lay out the major issues involved in conditional planning.

The conditional planning approach described in the chapter is based on the efficient search algorithms for cyclic AND–OR graphs developed by Jimenez and Torras (2000) and Hansen and Zilberstein (2001). Bertoli *et al.* (2001b) describe a BDD-based approach that constructs conditional plans with loops. C-BURIDAN (Draper *et al.*, 1994) handles conditional planning for actions with probabilistic outcomes, a problem also addressed under the heading of POMDPs (Chapter 17).

There is a close relation between conditional planning and automated program synthesis; a number of references appear in Chapter 9. The two fields have been pursued separately, because of the enormous difference in cost between execution of machine instructions and execution of actions by robot vehicles or manipulators. Linden (1991) attempts explicit cross-fertilization between the two fields.

In retrospect, it is now possible to see how the major classical planning algorithms led to extended versions for domains involving uncertainty. Search-based techniques led to search in belief space (Bonet and Geffner, 2000); SATPLAN led to stochastic SATPLAN (Majercik and Littman, 1999) and to planning using quantified Boolean logic (Rintanen, 1999); partial order planning led to UWL (Etzioni *et al.*, 1992), CNLP (Peot and Smith, 1992), and CASSANDRA (Pryor and Collins, 1996). GRAPHPLAN led to Sensory Graphplan or SGP (Weld *et al.*, 1998), but a full probabilistic GRAPHPLAN has yet to be developed.

The earliest major treatment of execution monitoring was PLANEX (Fikes *et al.*, 1972), which worked with the STRIPS planner to control the robot Shakey. PLANEX used triangle tables—essentially an efficient storage mechanism for the plan preconditions at each point in the plan—to allow recovery from partial execution failure without complete replanning. Shakey's model of execution is discussed further in Chapter 25. The NASL planner (McDermott, 1978a) treated a planning problem simply as a specification for carrying out a complex action, so that execution and planning were completely unified. It used theorem proving to reason about these complex actions.

SIPE (System for Interactive Planning and Execution monitoring) (Wilkins, 1988, 1990) was the first planner to deal systematically with the problem of replanning. It has been used in

demonstration projects in several domains, including planning operations on the flight deck of an aircraft carrier and job-shop scheduling for an Australian beer factory. Another study used SIPE to plan the construction of multistory buildings, one of the most complex domains ever tackled by a planner.

IPEM (Integrated Planning, Execution, and Monitoring) (Ambros-Ingerson and Steel, 1988) was the first system to integrate partial-order planning and execution to yield a continuous planning agent. Our CONTINUOUS-POP-AGENT combines ideas from IPEM, the PUCCINI planner (Golden, 1998), and the CYPRESS system (Wilkins *et al.*, 1995).

In the mid-1980s, it was believed by some that partial-order planning and related techniques could never run fast enough to generate effective behavior for an agent in the real world (Agre and Chapman, 1987). Instead, **reactive planning** systems were proposed; in their basic form, these are reflex agents, possibly with internal state, that can be implemented with any of a variety of representations for condition–action rules. Brooks's (1986) subsumption architecture (see Chapters 7 and 25) used layered finite-state machines in legged and wheeled robots to control their locomotion and avoid obstacles. Pengi (Agre and Chapman, 1987) was able to play a (fully observable) video game using Boolean circuits combined with a "visual" representation of current goals and the agent's internal state.

REACTIVE PLANNING

"Universal plans" (Schoppers, 1987) were developed as a lookup-table method for reactive planning, but turned out to be a rediscovery of the idea of **policies** that had long been used in Markov decision processes. A universal plan (or a policy) contains a mapping from any state to the action that should be taken in that state. Ginsberg (1989) made a spirited attack on universal plans, including intractability results for some formulations of the reactive planning problem. Schoppers (1989) made an equally spirited reply.

POLICIES

As is often the case, a hybrid approach resolves the controversy. Using well-designed hierarchies, HTN planners, such as PRS (Georgeff and Lansky, 1987) and RAP (Firby, 1996), as well as continuous planning agents, can achieve reactive response times and complex long-range planning behavior in many problem domains.

Multiagent planning has leaped in popularity in recent years, although it does have a long history. Konolige (1982) provided a formalization of multiagent planning in first-order logic, while Pednault (1986) gave a STRIPS-style description. The notion of joint intention, which is essential if agents are to execute a joint plan, comes from work on communicative acts (Cohen and Levesque, 1990; Cohen *et al.*, 1990). Our presentation of multibody partial-order planning is based on the work of Boutilier and Brafman (2001).

We have barely skimmed the surface of work on negotiation in multiagent planning. Durfee and Lesser (1989) discuss how tasks can be shared out among agents by negotiation. Kraus *et al.* (1991) describe a system for playing Diplomacy, a board game requiring negotiation, coalition formation and dissolution, and dishonesty. Stone (2000) shows how agents can cooperate as teammates in the competitive, dynamic, partially observable environment of robotic soccer. (Weiss, 1999) is a book-length overview of multiagent systems.

The boid model on page 453 is due to Reynolds (1987), who won an Academy Award for its application to flocks of bats and swarms of penguins in *Batman Returns*.

EXERCISES

12.1 Examine carefully the representation of time and resources in Section 12.1.

a. Why is it a good idea to have $Duration(d)$ be an effect of an action, rather than having a separate field in the action of the form DURATION: d? (*Hint:* Consider conditional effects and disjunctive effects.)

b. Why is RESOURCE:m a separate field in the action, rather than being an effect?

CONSUMABLE RESOURCE

12.2 A **consumable resource** is a resource that is (partially) used up by an action. For example, attaching engines to cars requires screws. The screws, once used, are not available for other attachments.

a. Explain how to modify the representation in Figure 12.3 so that there are 100 screws initially, engine E_1 requires 40 screws, and engine E_2 requires 50 screws. The $+$ and $-$ function symbols may be used in effect literals for resources.

b. Explain how the definition of **conflict** between causal links and actions in partial-order planning must be modified to handle consumable resources.

c. Some actions—for example, resupplying the factory with screws or refueling a car—can *increase* the availability of resources. A resource is monotonically non-increasing if no action increases it. Explain how to use this property to prune the search space.

12.3 Give decompositions for the *HireBuilder* and *GetPermit* steps in Figure 12.7, and show how the decomposed subplans connect into the overall plan.

12.4 Give an example in the house-building domain of two abstract subplans that cannot be merged into a consistent plan without sharing steps. (*Hint:* Places where two physical parts of the house come together are also places where two subplans tend to interact.)

12.5 Some people say an advantage of HTN planning is that it can solve problems like "take a round trip from Los Angeles to New York and back" that are hard to express in non-HTN notations because the start and goal states would be the same ($At(LA)$). Can you think of a way to represent and solve this problem without HTNs?

12.6 Show how a standard STRIPS action description can be rewritten as an HTN decomposition, using the notation $Achieve(p)$ to denote the *activity* of achieving the condition p.

12.7 Some of the operations in standard programming languages can be modeled as actions that change the state of the world. For example, the assignment operation copies the contents of a memory location, while the print operation changes the state of the output stream. A program consisting of these operations can also be considered as a plan, whose goal is given by the specification of the program. Therefore, planning algorithms can be used to construct programs that achieve a given specification.

a. Write an operator schema for the assignment operator (assigning the value of one variable to another). Remember that the original value will be overwritten!

b. Show how object creation can be used by a planner to produce a plan for exchanging the values of two variables using a temporary variable.

12.8 Consider the following argument: In a framework that allows uncertain initial states, **disjunctive effects** are just a notational convenience, not a source of additional representational power. For any action schema a with disjunctive effect $P \vee Q$, we could always replace it with the conditional effects **when** R: P $\wedge$ **when** $\neg R$: Q, which in turn can be reduced to two regular actions. The proposition R stands for a random proposition that is unknown in the initial state and for which there are no sensing actions. Is this argument correct? Consider separately two cases, one in which only one instance of action schema a is in the plan, the other in which more than one instance is.

12.9 Why can't conditional planning deal with unbounded indeterminacy?

12.10 In the blocks world we were forced to introduce two STRIPS actions, *Move* and *MoveToTable*, in order to maintain the *Clear* predicate properly. Show how conditional effects can be used to represent both of these cases with a single action.

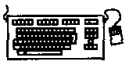
12.11 Conditional effects were illustrated for the *Suck* action in the vacuum world—which square becomes clean depends on which square the robot is in. Can you think of a new set of propositional variables to define states of the vacuum world, such that *Suck* has an *unconditional* description? Write out the descriptions of *Suck*, *Left*, and *Right*, using your propositions, and demonstrate that they suffice to describe all possible states of the world.

12.12 Write out the full description of *Suck* for the double Murphy vacuum cleaner that sometimes deposits dirt when it moves to a clean destination square and sometimes deposits dirt if *Suck* is applied to a clean square.

12.13 Find a suitably dirty carpet, free of obstacles, and vacuum it. Draw the path taken by the vacuum cleaner as accurately as you can. Explain it, with reference to the forms of planning discussed in this chapter.

12.14 The following quotes are from the backs of shampoo bottles. Identify each as an unconditional, conditional, or execution monitoring plan. (a) "Lather. Rinse. Repeat." (b) "Apply shampoo to scalp and let it remain for several minutes. Rinse and repeat if necessary." (c) "See a doctor if problems persist."

12.15 The AND-OR-GRAPH-SEARCH algorithm in Figure 12.10 checks for repeated states only on the path from the root to the current state. Suppose that, in addition, the algorithm were to store *every* visited state and check against that list. (See GRAPH-SEARCH in Figure 3.19 for an example.) Determine the information that should be stored and how the algorithm should use that information when a repeated state is found. (*Hint*: You will need to distinguish at least between states for which a successful subplan was constructed previously and states for which no subplan could be found.) Explain how to use **labels** to avoid having multiple copies of subplans.



12.16 Explain precisely how to modify the AND-OR-GRAPH-SEARCH algorithm to generate a cyclic plan if no acyclic plan exists. You will need to deal with three issues: labeling

the plan steps so that a cyclic plan can point back to an earlier part of the plan, modifying OR-SEARCH so that it continues to look for acyclic plans after finding a cyclic plan, and augmenting the plan representation to indicate whether a plan is cyclic. Show how your algorithm works on (a) the triple Murphy vacuum world, and (b) the alternate double Murphy vacuum world. You might wish to use a computer implementation to check your results. Can the plan for case (b) be written using standard loop syntax?

12.17 Specify in full the belief state update procedure for partially observable environments. That is, the method for computing the new belief state representation (as a list of knowledge propositions) from the current belief-state representation and an action description with conditional effects.

12.18 Write action descriptions, analogous to Equation (12.2), for the *Right* and *Suck* actions. Also write a description for *CheckLocation*, analogous to Equation (12.3). Repeat using the alternative set of propositions from Exercise 12.11.

12.19 Look at the list on page 445 of things that the replanning agent can't do. Sketch an algorithm that can handle one or more of them.

12.20 Consider the following problem: A patient arrives at the doctor's office with symptoms that could have been caused either by dehydration or by disease D (but not both). There are two possible actions: *Drink*, which unconditionally cures dehydration, and *Medicate*, which cures disease D, but has an undesirable side-effect if taken when the patient is dehydrated. Write the problem description in PDDL, and diagram a sensorless plan that solves the problem, enumerating all relevant possible worlds.

12.21 To the medication problem in the previous exercise, add a *Test* action that has the conditional effect *CultureGrowth* when *Disease* is true and in any case has the perceptual effect *Known*(*CultureGrowth*). Diagram a conditional plan that solves the problem and minimizes the use of the *Medicate* action.

13 UNCERTAINTY

In which we see what an agent should do when not all is crystal clear.

13.1 ACTING UNDER UNCERTAINTY

UNCERTAINTY

The logical agents described in Parts III and IV make the epistemological commitment that propositions are true, false, or unknown. When an agent knows enough facts about its environment, the logical approach enables it to derive plans that are guaranteed to work. This is a good thing. Unfortunately, *agents almost never have access to the whole truth about their environment.* Agents must, therefore, act under **uncertainty**. For example, an agent in the wumpus world of Chapter 7 has sensors that report only local information; most of the world is not immediately observable. A wumpus agent often will find itself unable to discover which of two squares contains a pit. If those squares are *en route* to the gold, then the agent might have to take a chance and enter one of the two squares.

The real world is far more complex than the wumpus world. For a logical agent, it might be impossible to construct a complete and correct description of how its actions will work. Suppose, for example, that the agent wants to drive someone to the airport to catch a flight and is considering a plan, A_{90}, that involves leaving home 90 minutes before the flight departs and driving at a reasonable speed. Even though the airport is only about 15 miles away, the agent will not be able to conclude with certainty that "Plan A_{90} will get us to the airport in time." Instead, it reaches the weaker conclusion "Plan A_{90} will get us to the airport in time, as long as my car doesn't break down or run out of gas, and I don't get into an accident, and there are no accidents on the bridge, and the plane doesn't leave early, and . . . ." None of these conditions can be deduced, so the plan's success cannot be inferred. This is an example of the **qualification problem** mentioned in Chapter 10.

If a logical agent cannot conclude that any particular course of action achieves its goal, then it will be unable to act. Conditional planning can overcome uncertainty to some extent, but only if the agent's sensing actions can obtain the required information and only if there are not too many different contingencies. Another possible solution would be to endow the agent with a simple but incorrect theory of the world that *does* enable it to derive a plan;

presumably, such plans will work *most* of the time, but problems arise when events contradict the agent's theory. Moreover, handling the tradeoff between the accuracy and usefulness of the agent's theory seems itself to require reasoning about uncertainty. In sum, no purely logical agent will be able to conclude that plan A_{90} is the right thing to do.

Nonetheless, let us suppose that A_{90} *is* in fact the right thing to do. What do we mean by saying this? As we discussed in Chapter 2, we mean that out of all the plans that could be executed, A_{90} is expected to maximize the agent's performance measure, given the information it has about the environment. The performance measure includes getting to the airport in time for the flight, avoiding a long, unproductive wait at the airport, and avoiding speeding tickets along the way. The information the agent has cannot guarantee any of these outcomes for A_{90}, but it can provide some degree of belief that they will be achieved. Other plans, such as A_{120}, might increase the agent's belief that it will get to the airport on time, but also increase the likelihood of a long wait. *The right thing to do—the* ***rational decision****—therefore depends on both the relative importance of various goals and the likelihood that, and degree to which, they will be achieved.* The remainder of this section hones these ideas, in preparation for the development of the general theories of uncertain reasoning and rational decisions that we present in this and subsequent chapters.

Handling uncertain knowledge

In this section, we look more closely at the nature of uncertain knowledge. We will use a simple diagnosis example to illustrate the concepts involved. Diagnosis—whether for medicine, automobile repair, or whatever—is a task that almost always involves uncertainty. Let us try to write rules for dental diagnosis using first-order logic, so that we can see how the logical approach breaks down. Consider the following rule:

$$\forall p \;\; Symptom(p, Toothache) \;\Rightarrow\; Disease(p, Cavity) \;.$$

The problem is that this rule is wrong. Not all patients with toothaches have cavities; some of them have gum disease, an abscess, or one of several other problems:

$$\forall p \;\; Symptom(p, Toothache) \;\Rightarrow\; Disease(p, Cavity) \vee Disease(p, GumDisease) \vee Disease(p, Abscess) \ldots$$

Unfortunately, in order to make the rule true, we have to add an almost unlimited list of possible causes. We could try turning the rule into a causal rule:

$$\forall p \;\; Disease(p, Cavity) \;\Rightarrow\; Symptom(p, Toothache) \;.$$

But this rule is not right either; not all cavities cause pain. The only way to fix the rule is to make it logically exhaustive: to augment the left-hand side with all the qualifications required for a cavity to cause a toothache. Even then, for the purposes of diagnosis, one must also take into account the possibility that the patient might have a toothache and a cavity that are unconnected.

Trying to use first-order logic to cope with a domain like medical diagnosis thus fails for three main reasons:

LAZINESS

- ◇ **Laziness**: It is too much work to list the complete set of antecedents or consequents needed to ensure an exceptionless rule and too hard to use such rules.

THEORETICAL IGNORANCE

◇ **Theoretical ignorance**: Medical science has no complete theory for the domain.

PRACTICAL IGNORANCE

◇ **Practical ignorance**: Even if we know all the rules, we might be uncertain about a particular patient because not all the necessary tests have been or can be run.

The connection between toothaches and cavities is just not a logical consequence in either direction. This is typical of the medical domain, as well as most other judgmental domains: law, business, design, automobile repair, gardening, dating, and so on. The agent's knowledge can at best provide only a **degree of belief** in the relevant sentences. Our main tool for dealing with degrees of belief will be **probability theory**, which assigns to each sentence a numerical degree of belief between 0 and 1. (Some alternative methods for uncertain reasoning are covered in Section 14.7.)

DEGREE OF BELIEF

PROBABILITY THEORY

Probability provides a way of **summarizing** *the uncertainty that comes from our laziness and ignorance.* We might not know for sure what afflicts a particular patient, but we believe that there is, say, an 80% chance—that is, a probability of 0.8—that the patient has a cavity if he or she has a toothache. That is, we expect that out of all the situations that are indistinguishable from the current situation as far as the agent's knowledge goes, the patient will have a cavity in 80% of them. This belief could be derived from statistical data—80% of the toothache patients seen so far have had cavities—or from some general rules, or from a combination of evidence sources. The 80% summarizes those cases in which all the factors needed for a cavity to cause a toothache are present and other cases in which the patient has both toothache and cavity but the two are unconnected. The missing 20% summarizes all the other possible causes of toothache that we are too lazy or ignorant to confirm or deny.

Assigning a probability of 0 to a given sentence corresponds to an unequivocal belief that the sentence is false, while assigning a probability of 1 corresponds to an unequivocal belief that the sentence is true. Probabilities between 0 and 1 correspond to intermediate degrees of belief in the truth of the sentence. The sentence itself is *in fact* either true or false. It is important to note that a degree of belief is different from a degree of truth. A probability of 0.8 does not mean "80% true" but rather an 80% degree of belief—that is, a fairly strong expectation. Thus, probability theory makes the same ontological commitment as logic—namely, that facts either do or do not hold in the world. Degree of truth, as opposed to degree of belief, is the subject of **fuzzy logic**, which is covered in Section 14.7.

In logic, a sentence such as "The patient has a cavity" is true or false depending on the interpretation and the world; it is true just when the fact it refers to is the case. In probability theory, a sentence such as "The probability that the patient has a cavity is 0.8" is about the agent's beliefs, not directly about the world. These beliefs depend on the percepts that the agent has received to date. These percepts constitute the **evidence** on which probability assertions are based. For example, suppose that the agent has drawn a card from a shuffled pack. Before looking at the card, the agent might assign a probability of $1/52$ to its being the ace of spades. After looking at the card, an appropriate probability for the same proposition would be 0 or 1. Thus, an assignment of probability to a proposition is analogous to saying whether a given logical sentence (or its negation) is entailed by the knowledge base, rather than whether or not it is true. Just as entailment status can change when more sentences are

EVIDENCE

added to the knowledge base, probabilities can change when more evidence is acquired.[1]

All probability statements must therefore indicate the evidence with respect to which the probability is being assessed. As the agent receives new percepts, its probability assessments are updated to reflect the new evidence. Before the evidence is obtained, we talk about **prior** or **unconditional** probability; after the evidence is obtained, we talk about **posterior** or **conditional** probability. In most cases, an agent will have some evidence from its percepts and will be interested in computing the posterior probabilities of the outcomes it cares about.

Uncertainty and rational decisions

The presence of uncertainty radically changes the way an agent makes decisions. A logical agent typically has a goal and executes any plan that is guaranteed to achieve it. An action can be selected or rejected on the basis of whether it achieves the goal, regardless of what other actions might achieve. When uncertainty enters the picture, this is no longer the case. Consider again the A_{90} plan for getting to the airport. Suppose it has a 95% chance of succeeding. Does this mean it is a rational choice? Not necessarily: There might be other plans, such as A_{120}, with higher probabilities of success. If it is vital not to miss the flight, then it is worth risking the longer wait at the airport. What about A_{1440}, a plan that involves leaving home 24 hours in advance? In most circumstances, this is not a good choice, because, although it almost guarantees getting there on time, it involves an intolerable wait.

PREFERENCES
OUTCOMES

To make such choices, an agent must first have **preferences** between the different possible **outcomes** of the various plans. A particular outcome is a completely specified state, including such factors as whether the agent arrives on time and the length of the wait at the airport. We will be using **utility theory** to represent and reason with preferences. (The term **utility** is used here in the sense of "the quality of being useful," not in the sense of the electric company or water works.) Utility theory says that every state has a degree of usefulness, or utility, to an agent and that the agent will prefer states with higher utility.

UTILITY THEORY

The utility of a state is relative to the agent whose preferences the utility function is supposed to represent. For example, the payoff functions for games in Chapter 6 are utility functions. The utility of a state in which White has won a game of chess is obviously high for the agent playing White, but low for the agent playing Black. Or again, some players (including the authors) might be happy with a draw against the world champion, whereas other players (including the former world champion) might not. There is no accounting for taste or preferences: you might think that an agent who prefers jalapeño bubble-gum ice cream to chocolate chocolate chip is odd or even misguided, but you could not say the agent is irrational. A utility function can even account for altruistic behavior, simply by including the welfare of others as one of the factors contributing to the agent's own utility.

Preferences, as expressed by utilities, are combined with probabilities in the general theory of rational decisions called **decision theory**:

DECISION THEORY

Decision theory = probability theory + utility theory .

[1] This is quite different from a sentence's becoming true or false as the world changes. Handling a changing world via probabilities requires the same kinds of mechanisms—situations, intervals, and events—that we used in Chapter 10 for logical representations. These mechanisms are discussed in Chapter 15.

The fundamental idea of decision theory is that *an agent is rational if and only if it chooses the action that yields the highest expected utility, averaged over all the possible outcomes of the action.* This is called the principle of **Maximum Expected Utility** (MEU). We saw this principle in action in Chapter 6 when we touched briefly on optimal decisions in backgammon. We will see that it is in fact a completely general principle.

Design for a decision-theoretic agent

Figure 13.1 sketches the structure of an agent that uses decision theory to select actions. The agent is identical, at an abstract level, to the logical agent described in Chapter 7. The primary difference is that the decision-theoretic agent's knowledge of the current state is uncertain;

BELIEF STATE

the agent's **belief state** is a representation of the probabilities of all possible actual states of the world. As time passes, the agent accumulates more evidence and its belief state changes. Given the belief state, the agent can make probabilistic predictions of action outcomes and hence select the action with highest expected utility. This chapter and the next concentrate on the task of representing and computing with probabilistic information in general. Chapter 15 deals with methods for the specific tasks of representing and updating the belief state and predicting the environment. Chapter 16 covers utility theory in more depth, and Chapter 17 develops algorithms for making complex decisions.

```
function DT-AGENT(percept) returns an action
  static: belief_state, probabilistic beliefs about the current state of the world
          action, the agent's action

  update belief_state based on action and percept
  calculate outcome probabilities for actions,
      given action descriptions and current belief_state
  select action with highest expected utility
      given probabilities of outcomes and utility information
  return action
```

Figure 13.1 A decision-theoretic agent that selects rational actions. The steps will be fleshed out in the next five chapters.

13.2 Basic Probability Notation

Now that we have set up the general framework for a rational agent, we will need a formal language for representing and reasoning with uncertain knowledge. Any notation for describing degrees of belief must be able to deal with two main issues: the nature of the sentences to which degrees of belief are assigned and the dependence of the degree of belief on the agent's experience. The version of probability theory we present uses an extension of propositional

logic for its sentences. The dependence on experience is reflected in the syntactic distinction between prior probability statements, which apply before any evidence is obtained, and conditional probability statements, which include the evidence explicitly.

Propositions

Degrees of belief are always applied to **propositions**—assertions that such-and-such is the case. So far we have seen two formal languages—propositional logic and first-order logic—for stating propositions. Probability theory typically uses a language that is slightly more expressive than propositional logic. This section describes that language. (Section 14.6 discusses ways to ascribe degrees of belief to assertions in first-order logic.)

RANDOM VARIABLE

The basic element of the language is the **random variable**, which can be thought of as referring to a "part" of the world whose "status" is initially unknown. For example, *Cavity* might refer to whether my lower left wisdom tooth has a cavity. Random variables play a role similar to that of CSP variables in constraint satisfaction problems and that of proposition symbols in propositional logic. We will always capitalize the names of random variables. (However, we still use lowercase, single-letter names to represent an unknown random variable, for example: $P(a) = 1 - P(\neg a)$.)

DOMAIN

Each random variable has a **domain** of values that it can take on. For example, the domain of *Cavity* might be $\langle true, false \rangle$.[2] (We will use lowercase for the names of values.) The simplest kind of proposition asserts that a random variable has a particular value drawn from its domain. For example, $Cavity = true$ might represent the proposition that I do in fact have a cavity in my lower left wisdom tooth.

As with CSP variables, random variables are typically divided into three kinds, depending on the type of the domain:

BOOLEAN RANDOM VARIABLES

- ♢ **Boolean random variables**, such as *Cavity*, have the domain $\langle true, false \rangle$. We will often abbreviate a proposition such as $Cavity = true$ simply by the lowercase name $cavity$. Similarly, $Cavity = false$ would be abbreviated by $\neg cavity$.

DISCRETE RANDOM VARIABLES

- ♢ **Discrete random variables**, which include Boolean random variables as a special case, take on values from a *countable* domain. For example, the domain of *Weather* might be $\langle sunny, rainy, cloudy, snow \rangle$. The values in the domain must be mutually exclusive and exhaustive. Where no confusion arises, we will use, for example, $snow$ as an abbreviation for $Weather = snow$.

CONTINUOUS RANDOM VARIABLES

- ♢ **Continuous random variables** take on values from the real numbers. The domain can be either the entire real line or some subset such as the interval [0,1]. For example, the proposition $X = 4.02$ asserts that the random variable X has the exact value 4.02. Propositions concerning continuous random variables can also be inequalities, such as $X \leq 4.02$

With some exceptions, we will be concentrating on the discrete case.

Elementary propositions, such as $Cavity = true$ and $Toothache = false$, can be combined to form complex propositions using all the standard logical connectives. For example,

[2] One might expect the domain to be written as a set: $\{true, false\}$. We write it as a tuple because it will be convenient later to impose an ordering on the values.

$Cavity = true \wedge Toothache = false$ is a proposition to which one may ascribe a degree of (dis)belief. As explained in the previous paragraph, this proposition may also be written as $cavity \wedge \neg toothache$.

Atomic events

ATOMIC EVENT

The notion of an **atomic event** is useful in understanding the foundations of probability theory. An atomic event is a *complete* specification of the state of the world about which the agent is uncertain. It can be thought of as an assignment of particular values to all the variables of which the world is composed. For example, if my world consists of only the Boolean variables $Cavity$ and $Toothache$, then there are just four distinct atomic events; the proposition $Cavity = false \wedge Toothache = true$ is one such event.[3]

Atomic events have some important properties:

- They are *mutually exclusive*—at most one can actually be the case. For example, $cavity \wedge toothache$ and $cavity \wedge \neg toothache$ cannot both be the case.
- The set of all possible atomic events is *exhaustive*—at least one must be the case. That is, the disjunction of all atomic events is logically equivalent to $true$.
- Any particular atomic event entails the truth or falsehood of every proposition, whether simple or complex. This can be seen by using the standard semantics for logical connectives (Chapter 7). For example, the atomic event $cavity \wedge \neg toothache$ entails the truth of $cavity$ and the falsehood of $cavity \Rightarrow toothache$.
- Any proposition is logically equivalent to the disjunction of all atomic events that entail the truth of the proposition. For example, the proposition $cavity$ is equivalent to disjunction of the atomic events $cavity \wedge toothache$ and $cavity \wedge \neg toothache$.

Exercise 13.4 asks you to prove some of these properties.

Prior probability

UNCONDITIONAL
PRIOR PROBABILITY

The **unconditional** or **prior probability** associated with a proposition a is the degree of belief accorded to it *in the absence of any other information*; it is written as $P(a)$. For example, if the prior probability that I have a cavity is 0.1, then we would write

$$P(Cavity = true) = 0.1 \quad \text{or} \quad P(cavity) = 0.1\,.$$

It is important to remember that $P(a)$ can be used only when there is no other information. As soon as some new information is known, we must reason with the *conditional* probability of a given that new information. Conditional probabilities are covered in the next section.

Sometimes, we will want to talk about the probabilities of all the possible values of a random variable. In that case, we will use an expression such as $\mathbf{P}(Weather)$, which denotes a *vector* of values for the probabilities of each individual state of the weather. Thus, instead

[3] Many standard formulations of probability theory take atomic events, also known as **sample points**, as primitive and define a random variable as a function taking an atomic event as input and returning a value from the appropriate domain. Such an approach is perhaps more general, but also less intuitive.

of writing the four equations

$$P(Weather = sunny) = 0.7$$
$$P(Weather = rain) = 0.2$$
$$P(Weather = cloudy) = 0.08$$
$$P(Weather = snow) = 0.02\ .$$

we may simply write

$$\mathbf{P}(Weather) = \langle 0.7, 0.2, 0.08, 0.02 \rangle\ .$$

PROBABILITY DISTRIBUTION

This statement defines a prior **probability distribution** for the random variable *Weather*.

We will also use expressions such as $\mathbf{P}(Weather, Cavity)$ to denote the probabilities of all combinations of the values of a set of random variables.[4] In that case, $\mathbf{P}(Weather, Cavity)$ can be represented by a 4×2 table of probabilities. This is called the **joint probability distribution** of *Weather* and *Cavity*.

JOINT PROBABILITY DISTRIBUTION

Sometimes it will be useful to think about the complete set of random variables used to describe the world. A joint probability distribution that covers this complete set is called the **full joint probability distribution**. For example, if the world consists of just the variables *Cavity*, *Toothache*, and *Weather*, then the full joint distribution is given by

FULL JOINT PROBABILITY DISTRIBUTION

$$\mathbf{P}(Cavity, Toothache, Weather).$$

This joint distribution can be represented as a $2 \times 2 \times 4$ table with 16 entries. A full joint distribution specifies the probability of every atomic event and is therefore a complete specification of one's uncertainty about the world in question. We will see in Section 13.4 that any probabilistic query can be answered from the full joint distribution.

For continuous variables, it is not possible to write out the entire distribution as a table, because there are infinitely many values. Instead, one usually defines the probability that a random variable takes on some value x as a parameterized function of x. For example, let the random variable X denote tomorrow's maximum temperature in Berkeley. Then the sentence

$$P(X = x) = U[18, 26](x)$$

expresses the belief that X is distributed uniformly between 18 and 26 degrees Celsius. (Several useful continuous distributions are defined in Appendix A.) Probability distributions for continuous variables are called **probability density functions**. Density functions differ in meaning from discrete distributions. For example, using the temperature distribution given earlier, we find that $P(X = 20.5) = U[18, 26](20.5) = 0.125/C$. This does *not* mean that there's a 12.5% chance that the maximum temperature will be *exactly* 20.5 degrees tomorrow; the probability that this will happen is of course zero. The technical meaning is that the probability that the temperature is in a small region around 20.5 degrees is equal, in the limit, to 0.125 divided by the width of the region in degrees Celsius:

PROBABILITY DENSITY FUNCTIONS

$$\lim_{dx \to 0} P(20.5 \leq X \leq 20.5 + dx)/dx = 0.125/C\ .$$

[4] The general notational rule is that the distribution covers all values of the variables that are capitalized. Thus, the expression $\mathbf{P}(Weather, cavity)$ is a four-element vector of probabilities for the conjunction of each weather type with $Cavity = true$.

Some authors use different symbols for discrete distributions and density functions; we use P in both cases, since confusion seldom arises and the equations are usually identical. Note that probabilities are unitless numbers, whereas density functions are measured with a unit, in this case reciprocal degrees.

Conditional probability

Once the agent has obtained some evidence concerning the previously unknown random variables making up the domain, prior probabilities are no longer applicable. Instead, we use **conditional** or **posterior** probabilities. The notation used is $P(a|b)$, where a and b are any propositions.[5] This is read as "the probability of a, given that *all we know* is b." For example,

CONDITIONAL PROBABILITY
POSTERIOR PROBABILITY

$$P(cavity|toothache) = 0.8$$

indicates that if a patient is observed to have a toothache and no other information is yet available, then the probability of the patient's having a cavity will be 0.8. A prior probability, such as $P(cavity)$, can be thought of as a special case of the conditional probability $P(cavity|\ )$, where the probability is conditioned on no evidence.

Conditional probabilities can be defined in terms of unconditional probabilities. The defining equation is

$$P(a|b) = \frac{P(a \wedge b)}{P(b)} \tag{13.1}$$

which holds whenever $P(b) > 0$. This equation can also be written as

$$P(a \wedge b) = P(a|b)P(b)$$

PRODUCT RULE

which is called the **product rule**. The product rule is perhaps easier to remember: it comes from the fact that, for a and b to be true, we need b to be true, and we also need a to be true given b. We can also have it the other way around:

$$P(a \wedge b) = P(b|a)P(a)\ .$$

In some cases, it is easier to reason in terms of prior probabilities of conjunctions, but for the most part, we will use conditional probabilities as our vehicle for probabilistic inference.

We can also use the **P** notation for conditional distributions. $\mathbf{P}(X|Y)$ gives the values of $P(X = x_i|Y = y_j)$ for each possible i, j. As an example of how this makes our notation more concise, consider applying the product rule to each case where the propositions a and b assert particular values of X and Y respectively. We obtain the following equations:

$$\begin{aligned}
P(X = x_1 \wedge Y = y_1) &= P(X = x_1|Y = y_1)P(Y = y_1)\ . \\
P(X = x_1 \wedge Y = y_2) &= P(X = x_1|Y = y_2)P(Y = y_2)\ . \\
&\vdots
\end{aligned}$$

We can combine all these into the single equation

$$\mathbf{P}(X, Y) = \mathbf{P}(X|Y)\mathbf{P}(Y)\ .$$

Remember that this denotes a set of equations relating the corresponding individual entries in the tables, *not* a matrix multiplication of the tables.

[5] The "|" operator has the lowest possible precedence, so $P(a \wedge b|c \vee d)$ means $P((a \wedge b)|(c \vee d))$.

It is tempting, but wrong, to view conditional probabilities as if they were logical implications with uncertainty added. For example, the sentence $P(a|b) = 0.8$ *cannot* be interpreted to mean "whenever b holds, conclude that $P(a)$ is 0.8." Such an interpretation would be wrong on two counts: first, $P(a)$ always denotes the prior probability of a, not the posterior probability given some evidence; second, the statement $P(a|b) = 0.8$ is immediately relevant just when b is the *only* available evidence. When additional information c is available, the degree of belief in a is $P(a|b \wedge c)$, which might have little relation to $P(a|b)$. For example, c might tell us directly whether a is true or false. If we examine a patient who complains of toothache, and discover a cavity, then we have additional evidence $cavity$, and we conclude (trivially) that $P(cavity|toothache \wedge cavity) = 1.0$.

13.3 THE AXIOMS OF PROBABILITY

So far, we have defined a syntax for propositions and for prior and conditional probability statements about those propositions. Now we must provide some sort of semantics for probability statements. We begin with the basic axioms that serve to define the probability scale and its endpoints:

1. All probabilities are between 0 and 1. For any proposition a,

$$0 \leq P(a) \leq 1 .$$

2. Necessarily true (i.e., valid) propositions have probability 1, and necessarily false (i.e., unsatisfiable) propositions have probability 0.

$$P(true) = 1 \qquad P(false) = 0 .$$

Next, we need an axiom that connects the probabilities of logically related propositions. The simplest way to do this is to define the probability of a disjunction as follows:

3. The probability of a disjunction is given by

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b) .$$

This rule is easily remembered by noting that the cases where a holds, together with the cases where b holds, certainly cover all the cases where $a \vee b$ holds; but summing the two sets of cases counts their intersection twice, so we need to subtract $P(a \wedge b)$.

KOLMOGOROV'S AXIOMS

These three axioms are often called **Kolmogorov's axioms** in honor of the Russian mathematician Andrei Kolmogorov, who showed how to build up the rest of probability theory from this simple foundation. Notice that the axioms deal only with prior probabilities rather than conditional probabilities; this is because we have already defined the latter in terms of the former via Equation (13.1).

WHERE DO PROBABILITIES COME FROM?

There has been endless debate over the source and status of probability numbers. The **frequentist** position is that the numbers can come only from *experiments*: if we test 100 people and find that 10 of them have a cavity, then we can say that the probability of a cavity is approximately 0.1. In this view, the assertion "the probability of a cavity is 0.1" means that 0.1 is the fraction that would be observed in the limit of infinitely many samples. From any finite sample, we can estimate the true fraction and also calculate how accurate our estimate is likely to be.

The **objectivist** view is that probabilities are real aspects of the universe—propensities of objects to behave in certain ways—rather than being just descriptions of an observer's degree of belief. For example, that a fair coin comes up heads with probability 0.5 is a propensity of the coin itself. In this view, frequentist measurements are attempts to observe these propensities. Most physicists agree that quantum phenomena are objectively probabilistic, but uncertainty at the macroscopic scale—e.g., in coin tossing—usually arises from ignorance of initial conditions and does not seem consistent with the propensity view.

The **subjectivist** view describes probabilities as a way of characterizing an agent's beliefs, rather than as having any external physical significance. This allows the doctor or analyst to make the numbers up—to say, "In my opinion, I expect the probability of a cavity to be about 0.1." Several more reliable techniques, such as the betting systems described on page 474, have also been developed for eliciting probability assessments from humans.

In the end, even a strict frequentist position involves subjective analysis, so the difference probably has little practical importance. The **reference class** problem illustrates the intrusion of subjectivity. Suppose that a frequentist doctor wants to know the chances that a patient has a particular disease. The doctor wants to consider other patients who are similar in important ways—age, symptoms, perhaps sex—and see what proportion of them had the disease. But if the doctor considered everything that is known about the patient—weight to the nearest gram, hair color, mother's maiden name, etc.—the result would be that there are no other patients who are exactly the same and thus no reference class from which to collect experimental data. This has been a vexing problem in the philosophy of science.

Laplace's **principle of indifference** (1816) states that propositions that are syntactically "symmetric" with respect to the evidence should be accorded equal probability. Various refinements have been proposed, culminating in the attempt by Carnap and others to develop a rigorous **inductive logic**, capable of computing the correct probability for any proposition from any collection of observations. Currently, it is believed that no unique inductive logic exists; rather, any such logic rests on a subjective prior probability distribution whose effect is diminished as more observations are collected.

Using the axioms of probability

We can derive a variety of useful facts from the basic axioms. For example, the familiar rule for negation follows by substituting $\neg a$ for b in axiom 3, giving us:

$$\begin{aligned} P(a \vee \neg a) &= P(a) + P(\neg a) - P(a \wedge \neg a) && \text{(by axiom 3 with } b = \neg a) \\ P(\mathit{true}) &- P(a) + P(\neg a) - P(\mathit{false}) && \text{(by logical equivalence)} \\ 1 &= P(a) + P(\neg a) && \text{(by axiom 2)} \\ P(\neg a) &= 1 - P(a) && \text{(by algebra).} \end{aligned}$$

The third line of this derivation is itself a useful fact and can be extended from the Boolean case to the general discrete case. Let the discrete variable D have the domain $\langle d_1, \ldots, d_n \rangle$. Then it is easy to show (Exercise 13.2) that

$$\sum_{i=1}^{n} P(D = d_i) = 1 \ .$$

That is, any probability distribution on a single variable must sum to 1.[6] It is also true that any *joint* probability distribution on any *set* of variables must sum to 1: this can be seen simply by creating a single megavariable whose domain is the cross product of the domains of the the original variables.

Recall that any proposition a is equivalent to the disjunction of all the atomic events in which a holds; call this set of events $\mathbf{e}(a)$. Recall also that atomic events are mutually exclusive, so the probability of any conjunction of atomic events is zero, by axiom 2. Hence, from axiom 3, we can derive the following simple relationship: *The probability of a proposition is equal to the sum of the probabilities of the atomic events in which it holds;* that is,

$$P(a) = \sum_{e_i \in \mathbf{e}(a)} P(e_i) \ . \tag{13.2}$$

This equation provides a simple method for computing the probability of any proposition, given a full joint distribution that specifies the probabilities of all atomic events. (See Section 13.4.) In subsequent sections we will derive additional rules for manipulating probabilities. First, however, we will examine the foundation for the axioms themselves.

Why the axioms of probability are reasonable

The axioms of probability can be seen as restricting the set of probabilistic beliefs that an agent can hold. This is somewhat analogous to the logical case, where a logical agent cannot simultaneously believe A, B, and $\neg(A \wedge B)$, for example. There is, however, an additional complication. In the logical case, the semantic definition of conjunction means that at least one of the three beliefs just mentioned *must be false in the world*, so it is unreasonable for an agent to believe all three. With probabilities, on the other hand, statements refer not to the world directly, but to the agent's own state of knowledge. Why, then, can an agent not hold the following set of beliefs, which clearly violates axiom 3?

$$\begin{aligned} P(a) &= 0.4 & \qquad P(a \wedge b) &= 0.0 \\ P(b) &= 0.3 & \qquad P(a \vee b) &= 0.8 \end{aligned} \tag{13.3}$$

[6] For continuous variables, the summation is replaced by an integral: $\int_{-\infty}^{\infty} P(X = x)\, dx = 1$.

This kind of question has been the subject of decades of intense debate between those who advocate the use of probabilities as the only legitimate form for degrees of belief and those who advocate alternative approaches. Here, we give one argument for the axioms of probability, first stated in 1931 by Bruno de Finetti.

The key to de Finetti's argument is the connection between degree of belief and actions. The idea is that if an agent has some degree of belief in a proposition a, then the agent should be able to state odds at which it is indifferent to a bet for or against a. Think of it as a game between two agents: Agent 1 states "my degree of belief in event a is 0.4." Agent 2 is then free to choose whether to bet for or against a, at stakes that are consistent with the stated degree of belief. That is, Agent 2 could choose to bet that a will occur, betting \$4 against Agent 1's \$6. Or Agent 2 could bet \$6 against \$4 that A will not occur.[7] If an agent's degrees of belief do not accurately reflect the world, then you would expect that it would tend to lose money over the long run to an opposing agent whose beliefs more accurately reflect the state of the world.

But de Finetti proved something much stronger: *If Agent 1 expresses a set of degrees of belief that violate the axioms of probability theory then there is a combination of bets by Agent 2 that* **guarantees** *that Agent 1 will lose money* **every** *time.* So if you accept the idea that an agent should be willing to "put its money where its probabilities are," then you should accept that it is irrational to have beliefs that violate the axioms of probability.

One might think that this betting game is rather contrived. For example, what if one refuses to bet? Does that end the argument? The answer is that the betting game is an abstract model for the decision-making situation in which every agent is *unavoidably* involved at every moment. Every action (including inaction) is a kind of bet, and every outcome can be seen as a payoff of the bet. Refusing to bet is like refusing to allow time to pass.

We will not provide the proof of de Finetti's theorem, but we will show an example. Suppose that Agent 1 has the set of degrees of belief from Equation (13.3). Figure 13.2 shows that if Agent 2 chooses to bet \$4 on a, \$3 on b, and \$2 on $\neg(a \vee b)$, then Agent 1 always loses money, regardless of the outcomes for a and b.

Agent 1		Agent 2		Outcome for Agent 1			
Proposition	Belief	Bet	Stakes	$a \wedge b$	$a \wedge \neg b$	$\neg a \wedge b$	$\neg a \wedge \neg b$
a	0.4	a	4 to 6	–6	–6	4	4
b	0.3	b	3 to 7	–7	3	–7	3
$a \vee b$	0.8	$\neg(a \vee b)$	2 to 8	2	2	2	–8
				–11	–1	–1	–1

Figure 13.2 Because Agent 1 has inconsistent beliefs, Agent 2 is able to devise a set of bets that guarantees a loss for Agent 1, no matter what the outcome of a and b.

[7] One might argue that the agent's preferences for different bank balances are such that the possibility of losing \$1 is not counterbalanced by an equal possibility of winning \$1. One possible response is to make the bet amounts small enough to avoid this problem. Savage's analysis (1954) circumvents the issue altogether.

Other strong philosophical arguments have been put forward for the use of probabilities, most notably those of Cox (1946) and Carnap (1950). The world being the way it is, however, practical demonstrations sometimes speak louder than proofs. The success of reasoning systems based on probability theory has been much more effective in making converts. We now look at how the axioms can be deployed to make inferences.

13.4 INFERENCE USING FULL JOINT DISTRIBUTIONS

PROBABILISTIC INFERENCE

In this section we will describe a simple method for **probabilistic inference**—that is, the computation from observed evidence of posterior probabilities for query propositions. We will use the full joint distribution as the "knowledge base" from which answers to all questions may be derived. Along the way we will also introduce several useful techniques for manipulating equations involving probabilities.

We begin with a very simple example: a domain consisting of just the three Boolean variables *Toothache*, *Cavity*, and *Catch* (the dentist's nasty steel probe catches in my tooth). The full joint distribution is a $2 \times 2 \times 2$ table as shown in Figure 13.3.

	toothache		$\neg$*toothache*	
	catch	$\neg$*catch*	*catch*	$\neg$*catch*
cavity	0.108	0.012	0.072	0.008
$\neg$*cavity*	0.016	0.064	0.144	0.576

Figure 13.3 A full joint distribution for the *Toothache*, *Cavity*, *Catch* world.

Notice that the probabilities in the joint distribution sum to 1, as required by the axioms of probability. Notice also that Equation (13.2) gives us a direct way to calculate the probability of any proposition, simple or complex: We simply identify those atomic events in which the proposition is true and add up their probabilities. For example, there are six atomic events in which $cavity \vee toothache$ holds:

$$P(cavity \vee toothache) = 0.108 + 0.012 + 0.072 + 0.008 + 0.016 + 0.064 = 0.28 .$$

One particularly common task is to extract the distribution over some subset of variables or a single variable. For example, adding the entries in the first row gives the unconditional or

MARGINAL PROBABILITY

marginal probability[8] of *cavity*:

$$P(cavity) = 0.108 + 0.012 + 0.072 + 0.008 = 0.2 .$$

MARGINALIZATION

This process is called **marginalization**, or **summing out**—because the variables other than *Cavity* are summed out. We can write the following general marginalization rule for any sets of variables $\mathbf{Y}$ and $\mathbf{Z}$:

$$\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z}} \mathbf{P}(\mathbf{Y}, \mathbf{z}) . \tag{13.4}$$

[8] So called because of a common practice among actuaries of writing the sums of observed frequencies in the margins of insurance tables.

That is, a distribution over $\mathbf{Y}$ can be obtained by summing out all the other variables from any joint distribution containing $\mathbf{Y}$. A variant of this rule involves conditional probabilities instead of joint probabilities, using the product rule:

$$\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z}} \mathbf{P}(\mathbf{Y}|\mathbf{z})P(\mathbf{z}) \ . \tag{13.5}$$

CONDITIONING

This rule is called **conditioning**. Marginalization and conditioning will turn out to be useful rules for all kinds of derivations involving probability expressions.

In most cases, we will be interested in computing *conditional* probabilities of some variables, given evidence about others. Conditional probabilities can be found by first using Equation (13.1) to obtain an expression in terms of unconditional probabilities and then evaluating the expression from the full joint distribution. For example, we can compute the probability of a cavity, given evidence of a toothache, as follows:

$$\begin{aligned} P(cavity|toothache) &= \frac{P(cavity \wedge toothache)}{P(toothache)} \\ &= \frac{0.108 + 0.012}{0.108 + 0.012 + 0.016 + 0.064} = 0.6 \ . \end{aligned}$$

Just to check, we can also compute the probability that there is no cavity, given a toothache:

$$\begin{aligned} P(\neg cavity|toothache) &= \frac{P(\neg cavity \wedge toothache)}{P(toothache)} \\ &= \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} = 0.4 \ . \end{aligned}$$

Notice that in these two calculations the term $1/P(toothache)$ remains constant, no matter

NORMALIZATION

which value of *Cavity* we calculate. In fact, it can be viewed as a **normalization** constant for the distribution $\mathbf{P}(Cavity|toothache)$, ensuring that it adds up to 1. Throughout the chapters dealing with probability, we will use α to denote such constants. With this notation, we can write the two preceding equations in one:

$$\begin{aligned} &\mathbf{P}(Cavity|toothache) = \alpha\,\mathbf{P}(Cavity, toothache) \\ &= \alpha\,[\mathbf{P}(Cavity, toothache, catch) + \mathbf{P}(Cavity, toothache, \neg catch)] \\ &= \alpha\,[\langle 0.108, 0.016\rangle + \langle 0.012, 0.064\rangle] = \alpha\,\langle 0.12, 0.08\rangle = \langle 0.6, 0.4\rangle \ . \end{aligned}$$

Normalization will turn out to be a useful shortcut in many probability calculations.

From the example, we can extract a general inference procedure. We will stick to the case in which the query involves a single variable. We will need some notation: let X be the query variable (*Cavity* in the example), let $\mathbf{E}$ be the set of evidence variables (just *Toothache* in the example), let $\mathbf{e}$ be the observed values for them, and let $\mathbf{Y}$ be the remaining unobserved variables (just *Catch* in the example). The query is $\mathbf{P}(X|\mathbf{e})$ and can be evaluated as

$$\mathbf{P}(X|\mathbf{e}) = \alpha\mathbf{P}(X, \mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}(X, \mathbf{e}, \mathbf{y}) \ , \tag{13.6}$$

where the summation is over all possible $\mathbf{y}$s (i.e., all possible combinations of values of the unobserved variables $\mathbf{Y}$). Notice that together the variables X, $\mathbf{E}$, and $\mathbf{Y}$ constitute the complete set of variables for the domain, so $\mathbf{P}(X, \mathbf{e}, \mathbf{y})$ is simply a subset of probabilities from the full joint distribution. The algorithm is shown in Figure 13.4. It loops over the values

```
function ENUMERATE-JOINT-ASK(X, e, P) returns a distribution over X
  inputs: X, the query variable
          e, observed values for variables E
          P, a joint distribution on variables {X} ∪ E ∪ Y    /* Y = hidden variables */

  Q(X) ← a distribution over X, initially empty
  for each value x_i of X do
      Q(x_i) ← ENUMERATE-JOINT(x_i, e, Y, [ ], P)
  return NORMALIZE(Q(X))

function ENUMERATE-JOINT(x, e, vars, values, P) returns a real number
  if EMPTY?(vars) then return P(x, e, values)
  Y ← FIRST(vars)
  return Σ_y ENUMERATE-JOINT(x, e, REST(vars), [y|values], P)
```

Figure 13.4 An algorithm for probabilistic inference by enumeration of the entries in a full joint distribution.

of X and the values of Y to enumerate all possible atomic events with **e** fixed, adds up their probabilities from the joint table, and normalizes the results.

Given the full joint distribution to work with, ENUMERATE-JOINT-ASK is a complete algorithm for answering probabilistic queries for discrete variables. It does not scale well, however: For a domain described by n Boolean variables, it requires an input table of size $O(2^n)$ and takes $O(2^n)$ time to process the table. In a realistic problem, there might be hundreds or thousands of random variables to consider, not just three. It quickly becomes completely impractical to define the vast numbers of probabilities required—the experience needed in order to estimate each of the table entries separately simply cannot exist.

For these reasons, the full joint distribution in tabular form is not a practical tool for building reasoning systems (although the historical notes at the end of the chapter includes one real-world application of this method). Instead, it should be viewed as the theoretical foundation on which more effective approaches may be built. The remainder of this chapter introduces some of the basic ideas required in preparation for the development of realistic systems in Chapter 14.

13.5 INDEPENDENCE

Let us expand the full joint distribution in Figure 13.3 by adding a fourth variable, *Weather*. The full joint distribution then becomes $\mathbf{P}(Toothache, Catch, Cavity, Weather)$, which has 32 entries (because *Weather* has four values). It contains four "editions" of the table shown in Figure 13.3, one for each kind of weather. It seems natural to ask what relationship these editions have to each other and to the original three-variable table. For example, how are $P(toothache, catch, cavity, Weather = cloudy)$ and $P(toothache, catch, cavity)$ related?

One way to answer this question is to use the product rule:

$$P(toothache, catch, cavity, Weather = cloudy)$$
$$= P(Weather = cloudy | toothache, catch, cavity) P(toothache, catch, cavity) .$$

Now, unless one is in the deity business, one should not imagine that one's dental problems influence the weather. Therefore, the following assertion seems reasonable:

$$P(Weather = cloudy | toothache, catch, cavity) = P(Weather = cloudy) . \quad (13.7)$$

From this, we can deduce

$$P(toothache, catch, cavity, Weather = cloudy)$$
$$= P(Weather = cloudy) P(toothache, catch, cavity) .$$

A similar equation exists for *every entry* in $\mathbf{P}(Toothache, Catch, Cavity, Weather)$. In fact, we can write the general equation

$$\mathbf{P}(Toothache, Catch, Cavity, Weather) = \mathbf{P}(Toothache, Catch, Cavity)\mathbf{P}(Weather) .$$

Thus, the 32-element table for four variables can be constructed from one 8-element table and one four-element table. This decomposition is illustrated schematically in Figure 13.5(a).

INDEPENDENCE

The property we used in writing Equation (13.7) is called **independence** (also **marginal independence** and **absolute independence**). In particular, the weather is independent of one's dental problems. Independence between propositions a and b can be written as

$$P(a|b) = P(a) \quad \text{or} \quad P(b|a) = P(b) \quad \text{or} \quad P(a \wedge b) = P(a)P(b) . \quad (13.8)$$

All these forms are equivalent (Exercise 13.7). Independence between variables X and Y can be written as follows (again, these are all equivalent):

$$\mathbf{P}(X|Y) = \mathbf{P}(X) \quad \text{or} \quad \mathbf{P}(Y|X) = \mathbf{P}(Y) \quad \text{or} \quad \mathbf{P}(X, Y) = \mathbf{P}(X)\mathbf{P}(Y) .$$

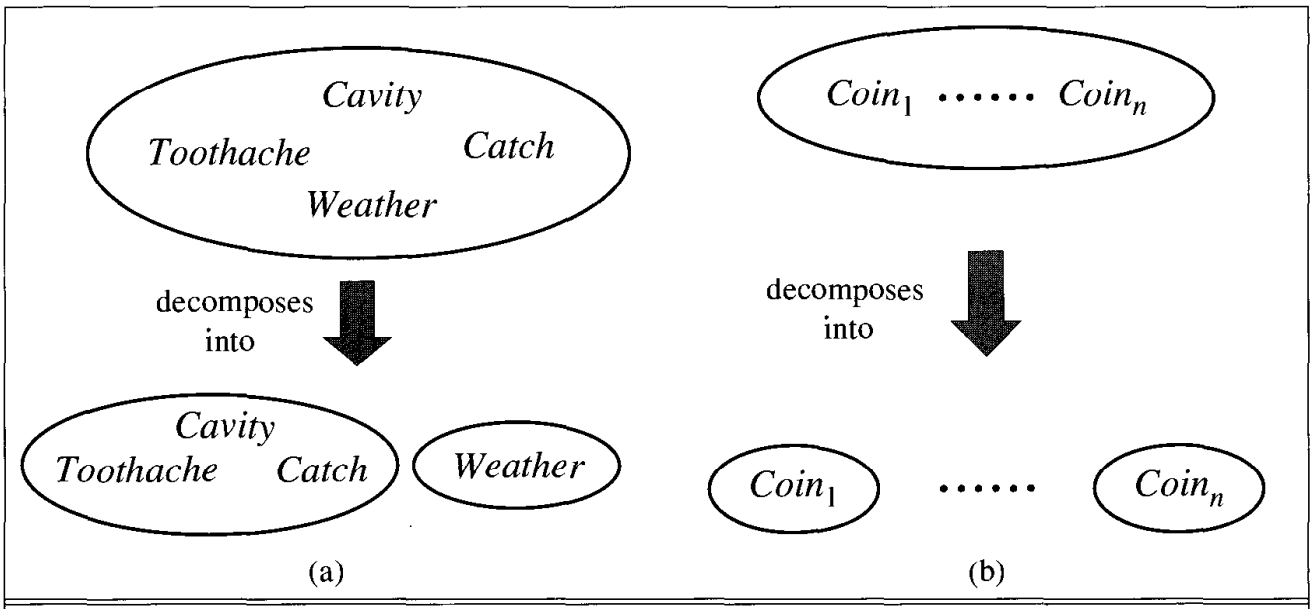


Figure 13.5 Two examples of factoring a large joint distribution into smaller distributions, using absolute independence. (a) Weather and dental problems are independent. (b) Coin flips are independent.

Independence assertions are usually based on knowledge of the domain. As we have seen, they can dramatically reduce the amount of information necessary to specify the full joint distribution. If the complete set of variables can be divided into independent subsets, then the full joint can be *factored* into separate joint distributions on those subsets. For example, the joint distribution on the outcome of n independent coin flips, $\mathbf{P}(C_1, \ldots, C_n)$, can be represented as the product of n single-variable distributions $\mathbf{P}(C_i)$. In a more practical vein, the independence of dentistry and meteorology is a good thing, because otherwise the practice of dentistry might require intimate knowledge of meteorology and *vice versa*.

When they are available, then, independence assertions can help in reducing the size of the domain representation and the complexity of the inference problem. Unfortunately, clean separation of entire sets of variables by independence is quite rare. Whenever a connection, however indirect, exists between two variables, independence will fail to hold. Moreover, even independent subsets can be quite large—for example, dentistry might involve dozens of diseases and hundreds of symptoms, all of which are interrelated. To handle such problems, we will need more subtle methods than the straightforward concept of independence.

13.6 Bayes' Rule and Its Use

On page 470, we defined the **product rule** and pointed out that it can be written in two forms because of the commutativity of conjunction:

$$P(a \wedge b) = P(a|b)P(b)$$
$$P(a \wedge b) = P(b|a)P(a) .$$

Equating the two right-hand sides and dividing by $P(a)$, we get

$$P(b|a) = \frac{P(a|b)P(b)}{P(a)} . \tag{13.9}$$

BAYES' RULE

This equation is known as **Bayes' rule** (also Bayes' law or Bayes' theorem).[9] This simple equation underlies all modern AI systems for probabilistic inference. The more general case of multivalued variables can be written in the **P** notation as

$$\mathbf{P}(Y|X) = \frac{\mathbf{P}(X|Y)\mathbf{P}(Y)}{\mathbf{P}(X)} .$$

where again this is to be taken as representing a set of equations, each dealing with specific values of the variables. We will also have occasion to use a more general version conditionalized on some background evidence **e**:

$$\mathbf{P}(Y|X, \mathbf{e}) = \frac{\mathbf{P}(X|Y, \mathbf{e})\mathbf{P}(Y|\mathbf{e})}{\mathbf{P}(X|\mathbf{e})} . \tag{13.10}$$

[9] According to rule 1 on page 1 of Strunk and White's *The Elements of Style*, it should be Bayes's rather than Bayes'. The latter is, however, more commonly used.

Applying Bayes' rule: The simple case

On the surface, Bayes' rule does not seem very useful. It requires three terms—a conditional probability and two unconditional probabilities—just to compute one conditional probability.

Bayes' rule is useful in practice because there are many cases where we do have good probability estimates for these three numbers and need to compute the fourth. In a task such as medical diagnosis, we often have conditional probabilities on causal relationships and want to derive a diagnosis. A doctor knows that the disease meningitis causes the patient to have a stiff neck, say, 50% of the time. The doctor also knows some unconditional facts: the prior probability that a patient has meningitis is 1/50,000, and the prior probability that any patient has a stiff neck is 1/20. Letting s be the proposition that the patient has a stiff neck and m be the proposition that the patient has meningitis, we have

$$\begin{aligned} P(s|m) &= 0.5 \\ P(m) &= 1/50000 \\ P(s) &= 1/20 \\ P(m|s) &= \frac{P(s|m)P(m)}{P(s)} = \frac{0.5 \times 1/50000}{1/20} = 0.0002 . \end{aligned}$$

That is, we expect only 1 in 5000 patients with a stiff neck to have meningitis. Notice that, even though a stiff neck is quite strongly indicated by meningitis (with probability 0.5), the probability of meningitis in the patient remains small. This is because the prior probability on stiff necks is much higher than that on meningitis.

Section 13.4 illustrated a process by which one can avoid assessing the probability of the evidence (here, $P(s)$) by instead computing a posterior probability for each value of the query variable (here, m and $\neg m$) and then normalizing the results. The same process can be applied when using Bayes' rule. We have

$$\mathbf{P}(M|s) = \alpha \langle P(s|m)P(m), P(s|\neg m)P(\neg m) \rangle .$$

Thus, in order to use this approach we need to estimate $P(s|\neg m)$ instead of $P(s)$. There is no free lunch—sometimes this is easier, sometimes it is harder. The general form of Bayes' rule with normalization is

$$\mathbf{P}(Y|X) = \alpha \mathbf{P}(X|Y)\mathbf{P}(Y) , \tag{13.11}$$

where α is the normalization constant needed to make the entries in $\mathbf{P}(Y|X)$ sum to 1.

One obvious question to ask about Bayes' rule is why one might have available the conditional probability in one direction, but not the other. In the meningitis domain, perhaps the doctor knows that a stiff neck implies meningitis in 1 out of 5000 cases; that is, the doctor has quantitative information in the **diagnostic** direction from symptoms to causes. Such a doctor has no need to use Bayes' rule. Unfortunately, *diagnostic knowledge is often more fragile than causal knowledge.* If there is a sudden epidemic of meningitis, the unconditional probability of meningitis, $P(m)$, will go up. The doctor who derived the diagnostic probability $P(m|s)$ directly from statistical observation of patients before the epidemic will have no idea how to update the value, but the doctor who computes $P(m|s)$ from the other three values will see that $P(m|s)$ should go up proportionately with $P(m)$. Most importantly, the

causal information $P(s|m)$ is *unaffected* by the epidemic, because it simply reflects the way meningitis works. The use of this kind of direct causal or model-based knowledge provides the crucial robustness needed to make probabilistic systems feasible in the real world.

Using Bayes' rule: Combining evidence

We have seen that Bayes' rule can be useful for answering probabilistic queries conditioned on one piece of evidence—for example, the stiff neck. In particular, we have argued that probabilistic information is often available in the form $P(\mathit{effect}|\mathit{cause})$. What happens when we have two or more pieces of evidence? For example, what can a dentist conclude if her nasty steel probe catches in the aching tooth of a patient? If we know the full joint distribution (Figure 13.3), one can read off the answer:

$$\mathbf{P}(\mathit{Cavity}|\mathit{toothache} \wedge \mathit{catch}) = \alpha \langle 0.108, 0.016 \rangle \approx \langle 0.871, 0.129 \rangle .$$

We know, however, that such an approach will not scale up to larger numbers of variables.

We can try using Bayes' rule to reformulate the problem:

$$\mathbf{P}(\mathit{Cavity}|\mathit{toothache} \wedge \mathit{catch}) = \alpha \mathbf{P}(\mathit{toothache} \wedge \mathit{catch}|\mathit{Cavity})\mathbf{P}(\mathit{Cavity}) . \quad (13.12)$$

For this reformulation to work, we need to know the conditional probabilities of the conjunction $\mathit{toothache} \wedge \mathit{catch}$ for each value of Cavity. That might be feasible for just two evidence variables, but again it will not scale up. If there are n possible evidence variables (X rays, diet, oral hygiene, etc.), then there are 2^n possible combinations of observed values for which we would need to know conditional probabilities. We might as well go back to using the full joint distribution. This is what first led researchers away from probability theory toward approximate methods for evidence combination that, while giving incorrect answers, require fewer numbers to give any answer at all.

Rather than taking this route, we need to find some additional assertions about the domain that will enable us to simplify the expressions. The notion of **independence** in Section 13.5 provides a clue, but needs refining. It would be nice if $\mathit{Toothache}$ and Catch were independent, but they are not: if the probe catches in the tooth, it probably has a cavity and that probably causes a toothache. These variables *are* independent, however, *given the presence or the absence of a cavity*. Each is directly caused by the cavity, but neither has a direct effect on the other: toothache depends on the state of the nerves in the tooth, whereas the probe's accuracy depends on the dentist's skill, to which the toothache is irrelevant.[10] Mathematically, this property is written as

$$\mathbf{P}(\mathit{toothache} \wedge \mathit{catch}|\mathit{Cavity}) = \mathbf{P}(\mathit{toothache}|\mathit{Cavity})\mathbf{P}(\mathit{catch}|\mathit{Cavity}) . \quad (13.13)$$

CONDITIONAL INDEPENDENCE

This equation expresses the **conditional independence** of $\mathit{toothache}$ and catch given Cavity. We can plug it into Equation (13.12) to obtain the probability of a cavity:

$$\mathbf{P}(\mathit{Cavity}|\mathit{toothache} \wedge \mathit{catch}) = \alpha \mathbf{P}(\mathit{toothache}|\mathit{Cavity})\mathbf{P}(\mathit{catch}|\mathit{Cavity})\mathbf{P}(\mathit{Cavity}).$$

Now the information requirements are the same as for inference using each piece of evidence separately: the prior probability $\mathbf{P}(\mathit{Cavity})$ for the query variable and the conditional probability of each effect, given its cause.

[10] We assume that the patient and dentist are distinct individuals.

The general definition of conditional independence of two variables X and Y, given a third variable Z is

$$\mathbf{P}(X, Y|Z) = \mathbf{P}(X|Z)\mathbf{P}(Y|Z) .$$

In the dentist domain, for example, it seems reasonable to assert conditional independence of the variables *Toothache* and *Catch*, given *Cavity*:

$$\mathbf{P}(Toothache, Catch|Cavity) = \mathbf{P}(Toothache|Cavity)\mathbf{P}(Catch|Cavity) . \quad (13.14)$$

Notice that this assertion is somewhat stronger than Equation (13.13), which asserts independence only for specific values of *Toothache* and *Catch*. As with absolute independence in Equation (13.8), the equivalent forms

$$\mathbf{P}(X|Y, Z) = \mathbf{P}(X|Z) \quad \text{and} \quad \mathbf{P}(Y|X, Z) = \mathbf{P}(Y|Z)$$

can also be used.

Section 13.5 showed that absolute independence assertions allow a decomposition of the full joint distribution into much smaller pieces. It turns out that the same is true for conditional independence assertions. For example, given the assertion in Equation (13.14), we can derive a decomposition as follows:

$$\begin{aligned}
&\mathbf{P}(Toothache, Catch, Cavity) \\
&= \mathbf{P}(Toothache, Catch|Cavity)\mathbf{P}(Cavity) \quad \text{(product rule)} \\
&= \mathbf{P}(Toothache|Cavity)\mathbf{P}(Catch|Cavity)\mathbf{P}(Cavity) \quad \text{[using (13.14)]}.
\end{aligned}$$

In this way, the original large table is decomposed into three smaller tables. The original table has seven independent numbers ($2^3 - 1$, because the numbers must sum to 1). The smaller tables contain five independent numbers ($2 \times (2^1 - 1)$ for each conditional probability distribution and $2^1 - 1$ for the prior on *Cavity*). This might not seem to be a major triumph, but the point is that, for n symptoms that are all conditionally independent given *Cavity*, the size of the representation grows as $O(n)$ instead of $O(2^n)$. Thus, *conditional independence assertions can allow probabilistic systems to scale up; moreover, they are much more commonly available than absolute independence assertions.* Conceptually, *Cavity* **separates** *Toothache* and *Catch* because it is a direct cause of both of them. The decomposition of large probabilistic domains into weakly connected subsets via conditional independence is one of the most important developments in the recent history of AI.

SEPARATION

The dentistry example illustrates a commonly occurring pattern in which a single cause directly influences a number of effects, all of which are conditionally independent, given the cause. The full joint distribution can be written as

$$\mathbf{P}(Cause, Effect_1, \ldots, Effect_n) = \mathbf{P}(Cause)\prod_i \mathbf{P}(Effect_i|Cause) .$$

NAIVE BAYES

Such a probability distribution is called a **naive Bayes** model—"naive" because it is often used (as a simplifying assumption) in cases where the "effect" variables are *not* conditionally independent given the cause variable. (The naive Bayes model is sometimes called a **Bayesian classifier**, a somewhat careless usage that has prompted true Bayesians to call it the **idiot Bayes** model.) In practice, naive Bayes systems can work surprisingly well, even when the independence assumption is not true. Chapter 20 describes methods for learning naive Bayes distributions from observations.

IDIOT BAYES

13.7 THE WUMPUS WORLD REVISITED

We can combine many of the ideas in this chapter to solve probabilistic reasoning problems in the wumpus world. (See Chapter 7 for a complete description of the wumpus world.) Uncertainty arises in the wumpus world because the agent's sensors give only partial, local information about the world. For example, Figure 13.6 shows a situation in which each of the three reachable squares—[1,3], [2,2], and [3,1]—might contain a pit. Pure logical inference can conclude nothing about which square is most likely to be safe, so a logical agent might be forced to choose randomly. We will see that a probabilistic agent can do much better than the logical agent.

Our aim will be to calculate the probability that each of the three squares contains a pit. (For the purposes of this example, we will ignore the wumpus and the gold.) The relevant properties of the wumpus world are that (1) a pit causes breezes in all neighboring squares, and (2) each square other than [1,1] contains a pit with probability 0.2. The first step is to identify the set of random variables we need:

- As in the propositional logic case, we want one Boolean variable P_{ij} for each square, which is true iff square $[i, j]$ actually contains a pit.
- We also have Boolean variables B_{ij} that are true iff square $[i, j]$ is breezy; we include these variables only for the observed squares—in this case, [1,1], [1,2], and [2,1].

The next step is to specify the full joint distribution, $\mathbf{P}(P_{1,1}, \ldots, P_{4,4}, B_{1,1}, B_{1,2}, B_{2,1})$. Applying the product rule, we have

$$\mathbf{P}(P_{1,1}, \ldots, P_{4,4}, B_{1,1}, B_{1,2}, B_{2,1}) = \mathbf{P}(B_{1,1}, B_{1,2}, B_{2,1} \mid P_{1,1}, \ldots, P_{4,4})\mathbf{P}(P_{1,1}, \ldots, P_{4,4}) .$$

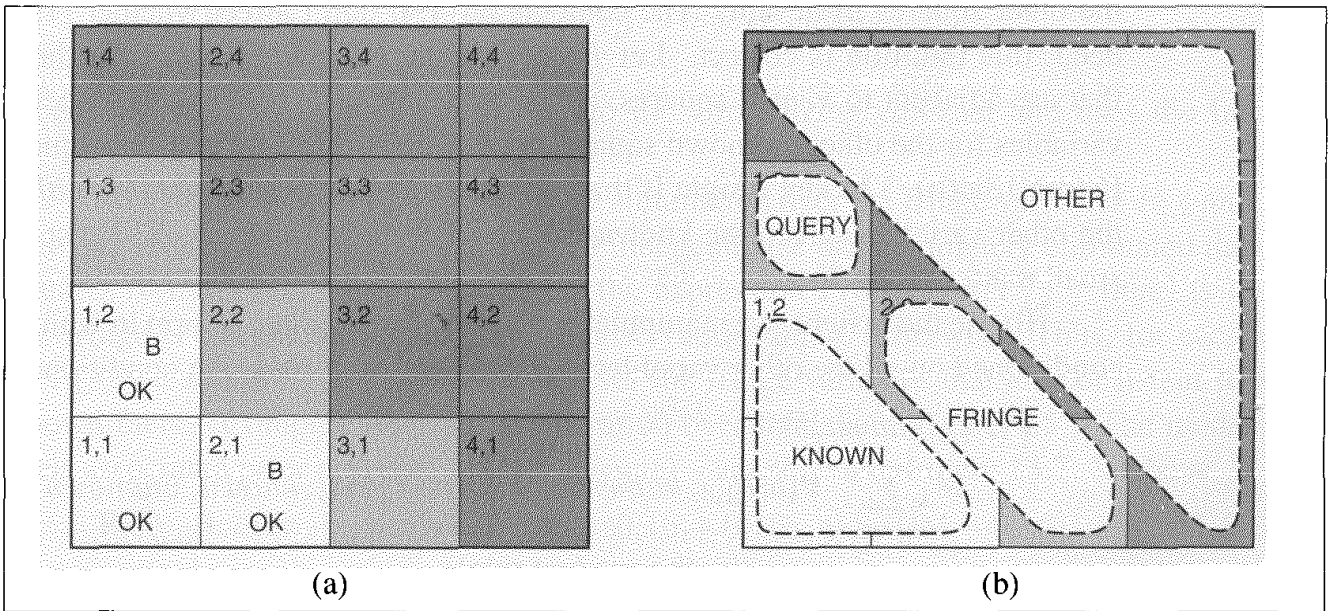


Figure 13.6 (a) After finding a breeze in both [1,2] and [2,1], the agent is stuck—there is no safe place to explore. (b) Division of the squares into *Known*, *Fringe*, and *Other*, for a query about [1,3].

This decomposition makes it very easy to see what the joint probability values should be. The first term is the conditional probability of a breeze configuration, given a pit configuration; this is 1 if the breezes are adjacent to the pits and 0 otherwise. The second term is the prior probability of a pit configuration. Each square contains a pit with probability 0.2, independently of the other squares; hence,

$$\mathbf{P}(P_{1,1},\ldots,P_{4,4}) = \prod_{i,j=1,1}^{4,4} \mathbf{P}(P_{i,j}) \; . \tag{13.15}$$

For a configuration with n pits, this is just $0.2^n \times 0.8^{16-n}$.

In the situation in Figure 13.6(a), the evidence consists of the observed breeze (or its absence) in each square that is visited, combined with the fact that each such square contains no pit. We'll abbreviate these facts as $b = \neg b_{1,1} \wedge b_{1,2} \wedge b_{2,1}$ and $known = \neg p_{1,1} \wedge \neg p_{1,2} \wedge \neg p_{2,1}$. We are interested in answering queries such as $\mathbf{P}(P_{1,3}|known, b)$: how likely is it that [1,3] contains a pit, given the observations so far?

To answer this query, we can follow the standard approach suggested by Equation (13.6) and implemented in the ENUMERATE-JOINT-ASK, namely, summing over entries from the full joint distribution. Let *Unknown* be a composite variable consisting of the $P_{i,j}$ variables for squares other than the *Known* squares and the query square [1,3]. Then, by Equation (13.6), we have

$$\mathbf{P}(P_{1,3}|known, b) = \alpha \sum_{unknown} \mathbf{P}(P_{1,3}, unknown, known, b) \; .$$

The full joint probabilities have already been specified, so we are done—that is, unless we care about computation. There are 12 unknown squares; hence the summation contains $2^{12} = 4096$ terms. In general, the summation grows exponentially with the number of squares.

Intuition suggests that we are missing something here. Surely, one might ask, aren't the other squares irrelevant? The contents of [4,4] don't affect whether [1,3] has a pit! Indeed, this intuition is correct. Let *Fringe* be the variables (other than the query variable) that are adjacent to visited squares, in this case just [2,2] and [3,1]. Also, let *Other* be the variables for the other unknown squares; in this case, there are 10 other squares, as shown in Figure 13.6(b). The key insight is that the observed breezes are *conditionally independent* of the other variables, given the known, fringe, and query variables. The rest is, as they say, a small matter of algebra.

To use the insight, we manipulate the query formula into a form in which the breezes are conditioned on all the other variables, and then we simplify using conditional independence:

$$\begin{aligned}
&\mathbf{P}(P_{1,3}|known, b) \\
&= \alpha \sum_{unknown} \mathbf{P}(b|P_{1,3}, known, unknown)\mathbf{P}(P_{1,3}, known, unknown) \\
&\qquad\qquad \text{(by the product rule)} \\
&= \alpha \sum_{fringe} \sum_{other} \mathbf{P}(b|known, P_{1,3}, fringe, other)\mathbf{P}(P_{1,3}, known, fringe, other) \\
&= \alpha \sum_{fringe} \sum_{other} \mathbf{P}(b|known, P_{1,3}, fringe)\mathbf{P}(P_{1,3}, known, fringe, other) \; ,
\end{aligned}$$

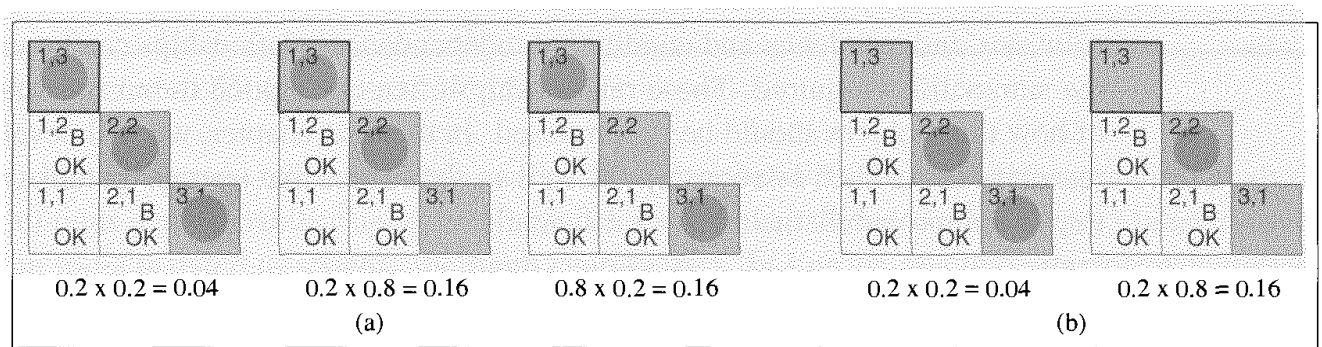


Figure 13.7 Consistent models for the fringe variables $P_{2,2}$ and $P_{3,1}$, showing $P(fringe)$ for each model: (a) three models with $P_{1,3} = true$ showing two or three pits, and (b) two models with $P_{1,3} = false$ showing one or two pits.

where the final step uses conditional independence. Now, the first term in this expression does not depend on the other variables, so we can move the summation inwards:

$$\begin{aligned}&\mathbf{P}(P_{1,3}|known, b)\\ &= \alpha \sum_{fringe} \mathbf{P}(b|known, P_{1,3}, fringe) \sum_{other} \mathbf{P}(P_{1,3}, known, fringe, other) \ .\end{aligned}$$

By independence, as in Equation (13.15), the prior term can be factored, and then the terms can be reordered:

$$\begin{aligned}&\mathbf{P}(P_{1,3}|known, b)\\ &= \alpha \sum_{fringe} \mathbf{P}(b|known, P_{1,3}, fringe) \sum_{other} \mathbf{P}(P_{1,3})P(known)P(fringe)P(other)\\ &= \alpha\, P(known)\mathbf{P}(P_{1,3}) \sum_{fringe} \mathbf{P}(b|known, P_{1,3}, fringe)P(fringe) \sum_{other} P(other)\\ &= \alpha' \mathbf{P}(P_{1,3}) \sum_{fringe} \mathbf{P}(b|known, P_{1,3}, fringe)P(fringe) \ ,\end{aligned}$$

where the last step folds $P(known)$ into the normalizing constant and uses the fact that $\sum_{other} P(other)$ equals 1.

Now, there are just four terms in the summation over the fringe variables $P_{2,2}$ and $P_{3,1}$. The use of independence and conditional independence has completely eliminated the other squares from consideration. Notice that the expression $\mathbf{P}(b|known, P_{1,3}, fringe)$ is 1 when the fringe is consistent with the breeze observations and 0 otherwise. Thus, for each value of $P_{1,3}$, we sum over the *logical models* for the fringe variables that are consistent with the known facts. (Compare with the enumeration over models in Figure 7.5.) The models and their associated prior probabilities—$P(fringe)$—are shown in Figure 13.7. We have

$$\mathbf{P}(P_{1,3}|known, b) = \alpha' \langle 0.2(0.04 + 0.16 + 0.16),\ 0.8(0.04 + 0.16)\rangle \approx \langle 0.31, 0.69\rangle \ .$$

That is, [1,3] (and [3,1] by symmetry) contains a pit with roughly 31% probability. A similar calculation, which the reader might wish to perform, shows that [2,2] contains a pit with roughly 86% probability. The wumpus agent should definitely avoid [2,2]!

What this section has shown is that even seemingly complicated problems can be formulated precisely in probability theory and solved using simple algorithms. To get *efficient*

solutions, independence and conditional independence relationships can be used to simplify the summations required. These relationships often correspond to our natural understanding of how the problem should be decomposed. In the next chapter, we will develop formal representations for such relationships as well as algorithms that operate on those representations to perform probabilistic inference efficiently.

13.8 SUMMARY

This chapter has argued that probability is the right way to reason about uncertainty.

- Uncertainty arises because of both laziness and ignorance. It is inescapable in complex, dynamic, or inaccessible worlds.
- Uncertainty means that many of the simplifications that are possible with deductive inference are no longer valid.
- Probabilities express the agent's inability to reach a definite decision regarding the truth of a sentence. Probabilities summarize the agent's beliefs.
- Basic probability statements include **prior probabilities** and **conditional probabilities** over simple and complex propositions.
- The **full joint probability distribution** specifies the probability of each complete assignment of values to random variables. It is usually too large to create or use in its explicit form.
- The axioms of probability constrain the possible assignments of probabilities to propositions. An agent that violates the axioms will behave irrationally in some circumstances.
- When the full joint distribution is available, it can be used to answer queries simply by adding up entries for the atomic events corresponding to the query propositions.
- **Absolute independence** between subsets of random variables might allow the full joint distribution to be factored into smaller joint distributions. This could greatly reduce complexity, but seldom occurs in practice.
- **Bayes' rule** allows unknown probabilities to be computed from known conditional probabilities, usually in the causal direction. Applying Bayes' rule with many pieces of evidence will in general run into the same scaling problems as does the full joint distribution.
- **Conditional independence** brought about by direct causal relationships in the domain might allow the full joint distribution to be factored into smaller, conditional distributions. The **naive Bayes** model assumes the conditional independence of all effect variables, given a single cause variable, and grows linearly with the number of effects.
- A wumpus-world agent can calculate probabilities for unobserved aspects of the world and use them to make better decisions than a purely logical agent makes.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Although games of chance date back at least to around 300 B.C., the mathematical analysis of odds and probability appears to be much more recent. Some work done by Mahaviracarya in India is dated to roughly the ninth century A.D. In Europe, the first attempts date only to the Italian Renaissance, beginning around 1500 A.D. The first significant systematic analyses were produced by Girolamo Cardano around 1565, but they remained unpublished until 1663. By that time, the discovery by Blaise Pascal (in correspondence with Pierre Fermat in 1654) of a systematic way of calculating probabilities had for the first time established probability as a mathematical discipline. The first published textbook on probability was *De Ratiociniis in Ludo Aleae* (Huygens, 1657). Pascal also introduced conditional probability, which is covered in Huygens's textbook. The Rev. Thomas Bayes (1702–1761) introduced the rule for reasoning about conditional probabilities that was named after him. It was published posthumously (Bayes, 1763). Kolmogorov (1950, first published in German in 1933) presented probability theory in a rigorously axiomatic framework for the first time. Rényi (1970) later gave an axiomatic presentation that took conditional probability, rather than absolute probability, as primitive.

Pascal used probability in ways that required both the objective interpretation, as a property of the world based on symmetry or relative frequency, and the subjective interpretation, based on degree of belief—the former in his analyses of probabilities in games of chance, the latter in the famous "Pascal's wager" argument about the possible existence of God. However, Pascal did not clearly realize the distinction between these two interpretations. The distinction was first drawn clearly by James Bernoulli (1654–1705).

Leibniz introduced the "classical" notion of probability as a proportion of enumerated, equally probable cases, which was also used by Bernoulli, although it was brought to prominence by Laplace (1749–1827). This notion is ambiguous between the frequency interpretation and the subjective interpretation. The cases can be thought to be equally probable either because of a natural, physical symmetry between them, or simply because we do not have any knowledge that would lead us to consider one more probable than another. The use of this latter, subjective consideration to justify assigning equal probabilities is known as the *principle of indifference* (Keynes, 1921).

The debate between objectivists and subjectivists became sharper in the 20th century. Kolmogorov (1963), R. A. Fisher (1922), and Richard von Mises (1928) were advocates of the relative frequency interpretation. Karl Popper's (1959, first published in German in 1934) "propensity" interpretation traces relative frequencies to an underlying physical symmetry. Frank Ramsey (1931), Bruno de Finetti (1937), R. T. Cox (1946), Leonard Savage (1954), and Richard Jeffrey (1983) interpreted probabilities as the degrees of belief of specific individuals. Their analyses of degree of belief were closely tied to utilities and to behavior—specifically, to the willingness to place bets. Rudolf Carnap, following Leibniz and Laplace, offered a different kind of subjective interpretation of probability—not as any actual individual's degree of belief, but as the degree of belief that an idealized individual *should* have in a particular proposition a, given a particular body of evidence $\mathbf{e}$. Carnap attempted to go further

CONFIRMATION

INDUCTIVE LOGIC

than Leibniz or Laplace by making this notion of degree of **confirmation** mathematically precise, as a logical relation between a and $\mathbf{e}$. The study of this relation was intended to constitute a mathematical discipline called **inductive logic**, analogous to ordinary deductive logic (Carnap, 1948, 1950). Carnap was not able to extend his inductive logic much beyond the propositional case, and Putnam (1963) showed that some fundamental difficulties would prevent a strict extension to languages capable of expressing arithmetic.

The question of reference classes is closely tied to the attempt to find an inductive logic. The approach of choosing the "most specific" reference class of sufficient size was formally proposed by Reichenbach (1949). Various attempts have been made, notably by Henry Kyburg (1977, 1983), to formulate more sophisticated policies in order to avoid some obvious fallacies that arise with Reichenbach's rule, but such approaches remain somewhat *ad hoc*. More recent work by Bacchus, Grove, Halpern, and Koller (1992) extends Carnap's methods to first-order theories, thereby avoiding many of the difficulties associated with the straightfoward reference-class method.

Bayesian probabilistic reasoning has been used in AI since the 1960s, especially in medical diagnosis. It was used not only to make a diagnosis from available evidence, but also to select further questions and tests using the theory of information value (Section 16.6) when available evidence was inconclusive (Gorry, 1968; Gorry *et al.*, 1973). One system outperformed human experts in the diagnosis of acute abdominal illnesses (de Dombal *et al.*, 1974). These early Bayesian systems suffered from a number of problems, however. Because they lacked any theoretical model of the conditions they were diagnosing, they were vulnerable to unrepresentative data occurring in situations for which only a small sample was available (de Dombal *et al.*, 1981). Even more fundamentally, because they lacked a concise formalism (such as the one to be described in Chapter 14) for representing and using conditional independence information, they depended on the acquisition, storage, and processing of enormous tables of probabilistic data. Because of these difficulties, probabilistic methods for coping with uncertainty fell out of favor in AI from the 1970s to the mid-1980s. Developments since the late 1980s are described in the next chapter.

The naive Bayes representation for joint distributions has been studied extensively in the pattern recognition literature since the 1950s (Duda and Hart, 1973). It has also been used, often unwittingly, in text retrieval, beginning with the work of Maron (1961). The probabilistic foundations of this technique, described further in Exercise 13.18, were elucidated by Robertson and Sparck Jones (1976). Domingos and Pazzani (1997) provide an explanation for the surprising success of naive Bayesian reasoning even in domains where the independence assumptions are clearly violated.

There are many good introductory textbooks on probability theory, including those by Chung (1979) and Ross (1988). Morris DeGroot (1989) offers a combined introduction to probability and statistics from a Bayesian standpoint, as well as a more advanced text (1970). Richard Hamming's (1991) textbook gives a mathematically sophisticated introduction to probability theory from the standpoint of a propensity interpretation based on physical symmetry. Hacking (1975) and Hald (1990) cover the early history of the concept of probability. Bernstein (1996) gives an entertaining popular account of the story of risk.

EXERCISES

13.1 Show from first principles that $P(a|b \wedge a) = 1$.

13.2 Using the axioms of probability, prove that any probability distribution on a discrete random variable must sum to 1.

13.3 Would it be rational for an agent to hold the three beliefs $P(A) = 0.4$, $P(B) = 0.3$, and $P(A \vee B) = 0.5$? If so, what range of probabilities would be rational for the agent to hold for $A \wedge B$? Make up a table like the one in Figure 13.2, and show how it supports your argument about rationality. Then draw another version of the table where $P(A \vee B) = 0.7$. Explain why it is rational to have this probability, even though the table shows one case that is a loss and three that just break even. (*Hint:* what is Agent 1 committed to about the probability of each of the four cases, especially the case that is a loss?)

13.4 This question deals with the properties of atomic events, as discussed on page 468.

a. Prove that the disjunction of all possible atomic events is logically equivalent to *true*. [*Hint:* Use a proof by induction on the number of random variables.]

b. Prove that any proposition is logically equivalent to the disjunction of the atomic events that entail its truth.

13.5 Consider the domain of dealing 5-card poker hands from a standard deck of 52 cards, under the assumption that the dealer is fair.

a. How many atomic events are there in the joint probability distribution (i.e., how many 5-card hands are there)?

b. What is the probability of each atomic event?

c. What is the probability of being dealt a royal straight flush? Four of a kind?

13.6 Given the full joint distribution shown in Figure 13.3, calculate the following:

a. $P(toothache)$

b. $\mathbf{P}(Cavity)$

c. $\mathbf{P}(Toothache|cavity)$

d. $\mathbf{P}(Cavity|toothache \vee catch)$.

13.7 Show that the three forms of independence in Equation (13.8) are equivalent.

13.8 After your yearly checkup, the doctor has bad news and good news. The bad news is that you tested positive for a serious disease and that the test is 99% accurate (i.e., the probability of testing positive when you do have the disease is 0.99, as is the probability of testing negative when you don't have the disease). The good news is that this is a rare disease, striking only 1 in 10,000 people of your age. Why is it good news that the disease is rare? What are the chances that you actually have the disease?

13.9 It is quite often useful to consider the effect of some specific propositions in the context of some general background evidence that remains fixed, rather than in the complete absence of information. The following questions ask you to prove more general versions of the product rule and Bayes' rule, with respect to some background evidence $\mathbf{e}$:

a. Prove the conditionalized version of the general product rule:

$$\mathbf{P}(X, Y|\mathbf{e}) = \mathbf{P}(X|Y, \mathbf{e})\mathbf{P}(Y|\mathbf{e}) .$$

b. Prove the conditionalized version of Bayes' rule in Equation (13.10).

13.10 Show that the statement

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

is equivalent to either of the statements

$$\mathbf{P}(A|B, C) = \mathbf{P}(A|C) \quad \text{and} \quad \mathbf{P}(B|A, C) = \mathbf{P}(B|C) .$$

13.11 Suppose you are given a bag containing n unbiased coins. You are told that $n - 1$ of these coins are normal, with heads on one side and tails on the other, whereas one coin is a fake, with heads on both sides.

a. Suppose you reach into the bag, pick out a coin uniformly at random, flip it, and get a head. What is the (conditional) probability that the coin you chose is the fake coin?

b. Suppose you continue flipping the coin for a total of k times after picking it and see k heads. Now what is the conditional probability that you picked the fake coin?

c. Suppose you wanted to decide whether the chosen coin was fake by flipping it k times. The decision procedure returns FAKE if all k flips come up heads, otherwise it returns NORMAL. What is the (unconditional) probability that this procedure makes an error?

13.12 In this exercise, you will complete the normalization calculation for the meningitis example. First, make up a suitable value for $P(S|\neg M)$, and use it to calculate unnormalized values for $P(M|S)$ and $P(\neg M|S)$ (i.e., ignoring the $P(S)$ term in the Bayes' rule expression). Now normalize these values so that they add to 1.

13.13 This exercise investigates the way in which conditional independence relationships affect the amount of information needed for probabilistic calculations.

a. Suppose we wish to calculate $P(h|e_1, e_2)$ and we have no conditional independence information. Which of the following sets of numbers are sufficient for the calculation?

(i) $\mathbf{P}(E_1, E_2), \mathbf{P}(H), \mathbf{P}(E_1|H), \mathbf{P}(E_2|H)$
(ii) $\mathbf{P}(E_1, E_2), \mathbf{P}(H), \mathbf{P}(E_1, E_2|H)$
(iii) $\mathbf{P}(H), \mathbf{P}(E_1|H), \mathbf{P}(E_2|H)$

b. Suppose we know that $\mathbf{P}(E_1|H, E_2) = \mathbf{P}(E_1|H)$ for all values of H, E_1, E_2. Now which of the three sets are sufficient?

13.14 Let X, Y, Z be Boolean random variables. Label the eight entries in the joint distribution $\mathbf{P}(X, Y, Z)$ as a through h. Express the statement that X and Y are conditionally

independent given Z as a set of equations relating a through h. How many *nonredundant* equations are there?

13.15 (Adapted from Pearl (1988).) Suppose you are a witness to a nighttime hit-and-run accident involving a taxi in Athens. All taxis in Athens are blue or green. You swear, under oath, that the taxi was blue. Extensive testing shows that, under the dim lighting conditions, discrimination between blue and green is 75% reliable. Is it possible to calculate the most likely color for the taxi? (*Hint:* distinguish carefully between the proposition that the taxi *is* blue and the proposition that it *appears* blue.)

What about now, given that 9 out of 10 Athenian taxis are green?

13.16 (Adapted from Pearl (1988).) Three prisoners, A, B, and C, are locked in their cells. It is common knowledge that one of them will be executed the next day and the others pardoned. Only the governor knows which one will be executed. Prisoner A asks the guard a favor: "Please ask the governor who will be executed, and then take a message to one of my friends B or C to let him know that he will be pardoned in the morning." The guard agrees, and comes back later and tells A that he gave the pardon message to B.

What are A's chances of being executed, given this information? (Answer this *mathematically*, not by energetic waving of hands.)

13.17 Write out a general algorithm for answering queries of the form $\mathbf{P}(Cause|\mathbf{e})$, using a naive Bayes distribution. You should assume that the evidence $\mathbf{e}$ may assign values to *any subset* of the effect variables.

13.18 Text categorization is the task of assigning a given document to one of a fixed set of categories, on the basis of the text it contains. Naive Bayes models are often used for this task. In these models, the query variable is the document category, and the "effect" variables are the presence or absence of each word in the language; the assumption is that words occur independently in documents, with frequencies determined by the document category.

a. Explain precisely how such a model can be constructed, given as "training data" a set of documents that have been assigned to categories.

b. Explain precisely how to categorize a new document.

c. Is the independence assumption reasonable? Discuss.

13.19 In our analysis of the wumpus world, we used the fact that each square contains a pit with probability 0.2, independently of the contents of the other squares. Suppose instead that exactly $N/5$ pits are scattered uniformly at random among the N squares other than [1,1]. Are the variables $P_{i,j}$ and $P_{k,l}$ still independent? What is the joint distribution $\mathbf{P}(P_{1,1}, \ldots, P_{4,4})$ now? Redo the calculation for the probabilities of pits in [1,3] and [2,2].

14 PROBABILISTIC REASONING

In which we explain how to build network models to reason under uncertainty according to the laws of probability theory.

Chapter 13 gave the syntax and semantics of probability theory. We remarked on the importance of independence and conditional independence relationships in simplifying probabilistic representations of the world. This chapter introduces a systematic way to represent such relationships explicitly in the form of **Bayesian networks**. We define the syntax and semantics of these networks and show how they can be used to capture uncertain knowledge in a natural and efficient way. We then show how probabilistic inference, although computationally intractable in the worst case, can be done efficiently in many practical situations. We also describe a variety of approximate inference algorithms that are often applicable when exact inference is infeasible. We explore ways in which probability theory can be applied to worlds with objects and relations—that is, to *first-order*, as opposed to *propositional*, representations. Finally, we survey alternative approaches to uncertain reasoning.

14.1 REPRESENTING KNOWLEDGE IN AN UNCERTAIN DOMAIN

In Chapter 13, we saw that the full joint probability distribution can answer any question about the domain, but can become intractably large as the number of variables grows. Furthermore, specifying probabilities for atomic events is rather unnatural and can be very difficult unless a large amount of data is available from which to gather statistical estimates.

We also saw that independence and conditional independence relationships among variables can greatly reduce the number of probabilities that need to be specified in order to define the full joint distribution. This section introduces a data structure called a **Bayesian network**[1] to represent the dependencies among variables and to give a concise specification of *any* full joint probability distribution.

BAYESIAN NETWORK

[1] This is the most common name, but there are many others, including **belief network**, **probabilistic network**, **causal network**, and **knowledge map**. In statistics, the term **graphical model** refers to a somewhat broader class that includes Bayesian networks. An extension of Bayesian networks called a **decision network** or **influence diagram** will be covered in Chapter 16.

A Bayesian network is a directed graph in which each *node* is annotated with quantitative probability information. The full specification is as follows:

1. A set of random variables makes up the nodes of the network. Variables may be discrete or continuous.
2. A set of directed links or arrows connects pairs of nodes. If there is an arrow from node X to node Y, X is said to be a *parent* of Y.
3. Each node X_i has a conditional probability distribution $P(X_i|Parents(X_i))$ that quantifies the effect of the parents on the node.
4. The graph has no directed cycles (and hence is a directed, acyclic graph, or DAG).

The topology of the network—the set of nodes and links—specifies the conditional independence relationships that hold in the domain, in a way that will be made precise shortly. The *intuitive* meaning of an arrow in a properly constructed network is usually that X has a *direct influence* on Y. It is usually easy for a domain expert to decide what direct influences exist in the domain—much easier, in fact, than actually specifying the probabilities themselves. Once the topology of the Bayesian network is laid out, we need only specify a conditional probability distribution for each variable, given its parents. We will see that the combination of the topology and the conditional distributions suffices to specify (implicitly) the full joint distribution for all the variables.

Recall the simple world described in Chapter 13, consisting of the variables *Toothache*, *Cavity*, *Catch*, and *Weather*. We argued that *Weather* is independent of the other variables; furthermore, we argued that *Toothache* and *Catch* are conditionally independent, given *Cavity*. These relationships are represented by the Bayesian network structure shown in Figure 14.1. Formally, the conditional independence of *Toothache* and *Catch* given *Cavity* is indicated by the *absence* of a link between *Toothache* and *Catch*. Intuitively, the network represents the fact that *Cavity* is a direct cause of *Toothache* and *Catch*, whereas no direct causal relationship exists between *Toothache* and *Catch*.

Now consider the following example, which is just a little more complex. You have a new burglar alarm installed at home. It is fairly reliable at detecting a burglary, but also responds on occasion to minor earthquakes. (This example is due to Judea Pearl, a resident of Los Angeles—hence the acute interest in earthquakes.) You also have two neighbors, John and Mary, who have promised to call you at work when they hear the alarm. John always calls

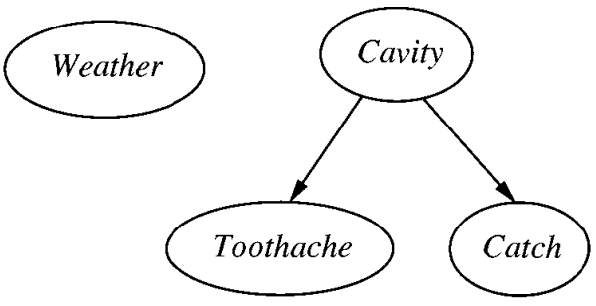


Figure 14.1 A simple Bayesian network in which *Weather* is independent of the other three variables and *Toothache* and *Catch* are conditionally independent, given *Cavity*.

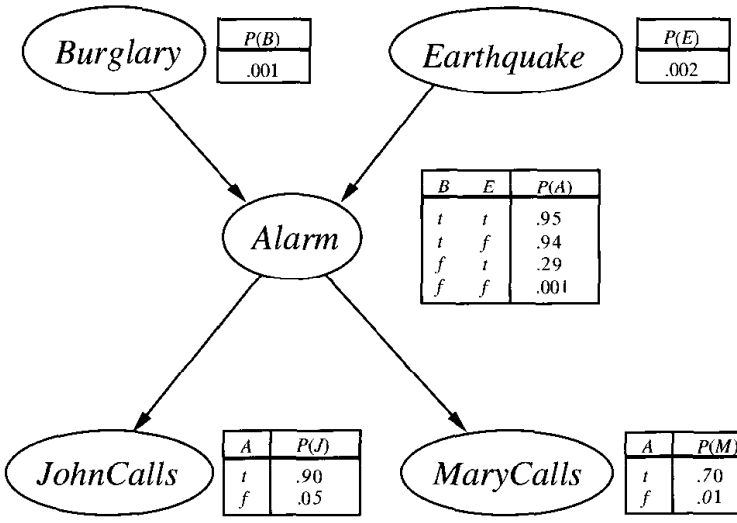


Figure 14.2 A typical Bayesian network, showing both the topology and the conditional probability tables (CPTs). In the CPTs, the letters B, E, A, J, and M stand for *Burglary*, *Earthquake*, *Alarm*, *JohnCalls*, and *MaryCalls*, respectively.

when he hears the alarm, but sometimes confuses the telephone ringing with the alarm and calls then, too. Mary, on the other hand, likes rather loud music and sometimes misses the alarm altogether. Given the evidence of who has or has not called, we would like to estimate the probability of a burglary. The Bayesian network for this domain appears in Figure 14.2.

For the moment, let us ignore the conditional distributions in the figure and concentrate on the topology of the network. In the case of the burglary network, the topology shows that burglary and earthquakes directly affect the probability of the alarm's going off, but whether John and Mary call depends only on the alarm. The network thus represents our assumptions that they do not perceive any burglaries directly, they do not notice the minor earthquakes, and they do not confer before calling.

Notice that the network does not have nodes corresponding to Mary's currently listening to loud music or to the telephone ringing and confusing John. These factors are summarized in the uncertainty associated with the links from *Alarm* to *JohnCalls* and *MaryCalls*. This shows both laziness and ignorance in operation: it would be a lot of work to find out why those factors would be more or less likely in any particular case, and we have no reasonable way to obtain the relevant information anyway. The probabilities actually summarize a *potentially infinite* set of circumstances in which the alarm might fail to go off (high humidity, power failure, dead battery, cut wires, a dead mouse stuck inside the bell, etc.) or John or Mary might fail to call and report it (out to lunch, on vacation, temporarily deaf, passing helicopter, etc.). In this way, a small agent can cope with a very large world, at least approximately. The degree of approximation can be improved if we introduce additional relevant information.

CONDITIONAL PROBABILITY TABLE

Now let us turn to the conditional distributions shown in Figure 14.2. In the figure, each distribution is shown as a **conditional probability table**, or CPT. (This form of table can be used for discrete variables; other representations, including those suitable for continuous variables, are described in Section 14.2.) Each row in a CPT contains the conditional probability of each node value for a **conditioning case**. A conditioning case is just a possible

CONDITIONING CASE

combination of values for the parent nodes—a miniature atomic event, if you like. Each row must sum to 1, because the entries represent an exhaustive set of cases for the variable. For Boolean variables, once you know that the probability of a true value is p, the probability of false must be $1 - p$, so we often omit the second number, as in Figure 14.2. In general, a table for a Boolean variable with k Boolean parents contains 2^k independently specifiable probabilities. A node with no parents has only one row, representing the prior probabilities of each possible value of the variable.

14.2 THE SEMANTICS OF BAYESIAN NETWORKS

The previous section described what a network is, but not what it means. There are two ways in which one can understand the semantics of Bayesian networks. The first is to see the network as a representation of the joint probability distribution. The second is to view it as an encoding of a collection of conditional independence statements. The two views are equivalent, but the first turns out to be helpful in understanding how to *construct* networks, whereas the second is helpful in designing inference procedures.

Representing the full joint distribution

A Bayesian network provides a complete description of the domain. Every entry in the full joint probability distribution (hereafter abbreviated as "joint") can be calculated from the information in the network. A generic entry in the joint distribution is the probability of a conjunction of particular assignments to each variable, such as $P(X_1 = x_1 \wedge \ldots \wedge X_n = x_n)$. We use the notation $P(x_1, \ldots, x_n)$ as an abbreviation for this. The value of this entry is given by the formula

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i | parents(X_i)) \,, \tag{14.1}$$

where $parents(X_i)$ denotes the specific values of the variables in $Parents(X_i)$. Thus, each entry in the joint distribution is represented by the product of the appropriate elements of the conditional probability tables (CPTs) in the Bayesian network. The CPTs therefore provide a decomposed representation of the joint distribution.

To illustrate this, we can calculate the probability that the alarm has sounded, but neither a burglary nor an earthquake has occurred, and both John and Mary call. We use single-letter names for the variables:

$$\begin{aligned} &P(j \wedge m \wedge a \wedge \neg b \wedge \neg e) \\ &\quad = P(j|a)P(m|a)P(a|\neg b \wedge \neg e)P(\neg b)P(\neg e) \\ &\quad = 0.90 \times 0.70 \times 0.001 \times 0.999 \times 0.998 = 0.00062 \,. \end{aligned}$$

Section 13.4 explained that the full joint distribution can be used to answer any query about the domain. If a Bayesian network is a representation of the joint distribution, then it too can be used to answer any query, by summing all the relevant joint entries. Section 14.4 explains how to do this, but also describes methods that are much more efficient.

A method for constructing Bayesian networks

Equation (14.1) defines what a given Bayesian network means. It does not, however, explain how to *construct* a Bayesian network in such a way that the resulting joint distribution is a good representation of a given domain. We will now show that Equation (14.1) implies certain conditional independence relationships that can be used to guide the knowledge engineer in constructing the topology of the network. First, we rewrite the joint distribution in terms of a conditional probability, using the product rule (see Chapter 13):

$$P(x_1, \ldots, x_n) = P(x_n|x_{n-1}, \ldots, x_1)P(x_{n-1}, \ldots, x_1) .$$

Then we repeat the process, reducing each conjunctive probability to a conditional probability and a smaller conjunction. We end up with one big product:

$$\begin{aligned} P(x_1, \ldots, x_n) &= P(x_n|x_{n-1}, \ldots, x_1)P(x_{n-1}|x_{n-2}, \ldots, x_1) \cdots P(x_2|x_1)P(x_1) \\ &= \prod_{i=1}^{n} P(x_i|x_{i-1}, \ldots, x_1) . \end{aligned}$$

CHAIN RULE

This identity holds true for any set of random variables and is called the **chain rule**. Comparing it with Equation (14.1), we see that the specification of the joint distribution is equivalent to the general assertion that, for every variable X_i in the network,

$$\mathbf{P}(X_i|X_{i-1}, \ldots, X_1) = \mathbf{P}(X_i|Parents(X_i)) , \tag{14.2}$$

provided that $Parents(X_i) \subseteq \{X_{i-1}, \ldots, X_1\}$. This last condition is satisfied by labeling the nodes in any order that is consistent with the partial order implicit in the graph structure.

What Equation (14.2) says is that the Bayesian network is a correct representation of the domain only if each node is conditionally independent of its predecessors in the node ordering, given its parents. Hence, in order to construct a Bayesian network with the correct structure for the domain, we need to choose parents for each node such that this property holds. Intuitively, the parents of node X_i should contain all those nodes in $X_1, \ldots, X_{i-1}$ that *directly influence* X_i. For example, suppose we have completed the network in Figure 14.2 except for the choice of parents for *MaryCalls*. *MaryCalls* is certainly influenced by whether there is a *Burglary* or an *Earthquake*, but not *directly* influenced. Intuitively, our knowledge of the domain tells us that these events influence Mary's calling behavior only through their effect on the alarm. Also, given the state of the alarm, whether John calls has no influence on Mary's calling. Formally speaking, we believe that the following conditional independence statement holds:

$$\mathbf{P}(MaryCalls|JohnCalls, Alarm, Earthquake, Burglary) = \mathbf{P}(MaryCalls|Alarm) .$$

Compactness and node ordering

As well as being a complete and nonredundant representation of the domain, a Bayesian network can often be far more *compact* than the full joint distribution. This property is what makes it feasible to handle domains with many variables. The compactness of Bayesian networks is an example of a very general property of **locally structured** (also called **sparse**) systems. In a locally structured system, each subcomponent interacts directly with only a bounded number of other components, regardless of the total number of components. Local

LOCALLY STRUCTURED
SPARSE

structure is usually associated with linear rather than exponential growth in complexity. In the case of Bayesian networks, it is reasonable to suppose that in most domains each random variable is directly influenced by at most k others, for some constant k. If we assume n Boolean variables for simplicity, then the amount of information needed to specify each conditional probability table will be at most 2^k numbers, and the complete network can be specified by $n2^k$ numbers. In contrast, the joint distribution contains 2^n numbers. To make this concrete, suppose we have $n = 30$ nodes, each with five parents ($k = 5$). Then the Bayesian network requires 960 numbers, but the full joint distribution requires over a billion.

There are domains in which each variable can be influenced directly by all the others, so that the network is fully connected. Then specifying the conditional probability tables requires the same amount of information as specifying the joint distribution. In some domains, there will be slight dependencies that should strictly be included by adding a new link. But if these dependencies are very tenuous, then it may not be worth the additional complexity in the network for the small gain in accuracy. For example, one might object to our burglary network on the grounds that if there is an earthquake, then John and Mary would not call even if they heard the alarm, because they assume that the earthquake is the cause. Whether to add the link from *Earthquake* to *JohnCalls* and *MaryCalls* (and thus enlarge the tables) depends on comparing the importance of getting more accurate probabilities with the cost of specifying the extra information.

Even in a locally structured domain, constructing a locally structured Bayesian network is not a trivial problem. We require not only that each variable be directly influenced by only a few others, but also that the network topology actually reflect those direct influences with the appropriate set of parents. Because of the way that the construction procedure works, the "direct influencers" will have to be added to the network first if they are to become parents of the node they influence. Therefore, *the correct order in which to add nodes is to add the "root causes" first, then the variables they influence,* and so on, until we reach the "leaves," which have no direct causal influence on the other variables.

What happens if we happen to choose the wrong order? Let us consider the burglary example again. Suppose we decide to add the nodes in the order *MaryCalls*, *JohnCalls*, *Alarm*, *Burglary*, *Earthquake*. Then we get the somewhat more complicated network shown in Figure 14.3(a). The process goes as follows:

- Adding *MaryCalls*: No parents.
- Adding *JohnCalls*: If Mary calls, that probably means the alarm has gone off, which of course would make it more likely that John calls. Therefore, *JohnCalls* needs *MaryCalls* as a parent
- Adding *Alarm*: Clearly, if both call, it is more likely that the alarm has gone off than if just one or neither call, so we need both *MaryCalls* and *JohnCalls* as parents.
- Adding *Burglary*: If we know the alarm state, then the call from John or Mary might give us information about our phone ringing or Mary's music, but not about burglary:

$$\mathbf{P}(Burglary|Alarm, JohnCalls, MaryCalls) = \mathbf{P}(Burglary|Alarm) \ .$$

Hence we need just *Alarm* as parent.

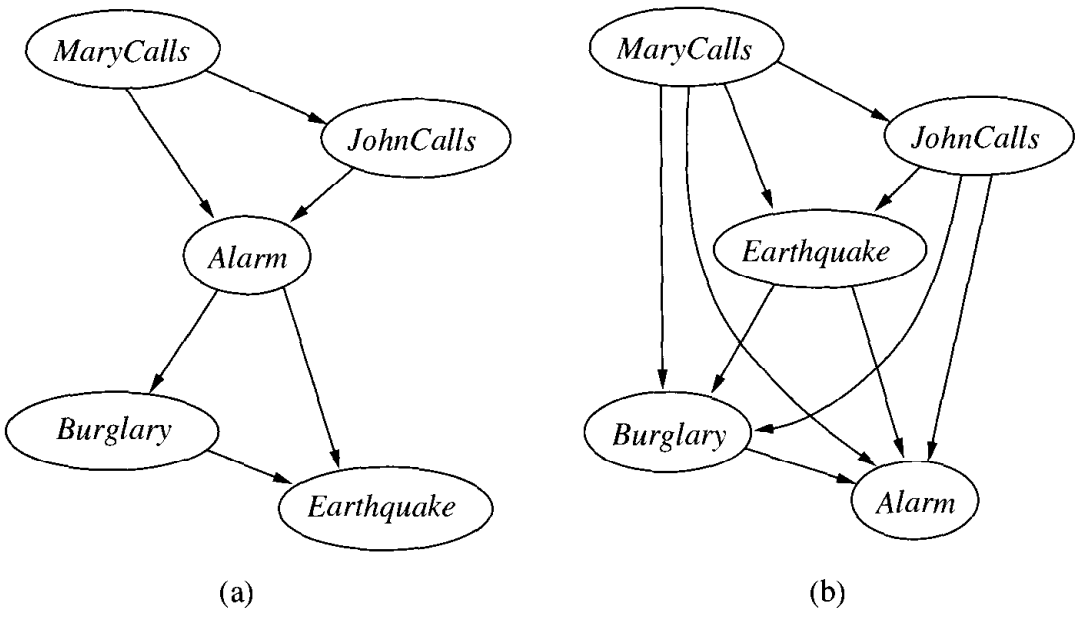


Figure 14.3 Network structure depends on order of introduction. In each network, we have introduced nodes in top-to-bottom order.

- Adding *Earthquake*: if the alarm is on, it is more likely that there has been an earthquake. (The alarm is an earthquake detector of sorts.) But if we know that there has been a burglary, then that explains the alarm, and the probability of an earthquake would be only slightly above normal. Hence, we need both *Alarm* and *Burglary* as parents.

The resulting network has two more links than the original network in Figure 14.2 and requires three more probabilities to be specified. What's worse, some of the links represent tenuous relationships that require difficult and unnatural probability judgments, such as assessing the probability of *Earthquake*, given *Burglary* and *Alarm*. This phenomenon is quite general and is related to the distinction between causal and diagnostic models introduced in Chapter 8. If we try to build a diagnostic model with links from symptoms to causes (as from *MaryCalls* to *Alarm* or *Alarm* to *Burglary*), we end up having to specify additional dependencies between otherwise independent causes (and often between separately occurring symptoms as well). *If we stick to a causal model, we end up having to specify fewer numbers, and the numbers will often be easier to come up with.* In the domain of medicine, for example, it has been shown by Tversky and Kahneman (1982) that expert physicians prefer to give probability judgments for causal rules rather than for diagnostic ones.

Figure 14.3(b) shows a very bad node ordering: *MaryCalls*, *JohnCalls*, *Earthquake*, *Burglary*, *Alarm*. This network requires 31 distinct probabilities to be specified—exactly the same as the full joint distribution. It is important to realize, however, that any of the three networks can represent *exactly the same joint distribution.* The last two versions simply fail to represent all the conditional independence relationships and hence end up specifying a lot of unnecessary numbers instead.

Conditional independence relations in Bayesian networks

We have provided a "numerical" semantics for Bayesian networks in terms of the representation of the full joint distribution, as in Equation (14.1). Using this semantics to derive a method for constructing Bayesian networks, we were led to the consequence that a node is conditionally independent of its predecessors, given its parents. It turns out that we can also go in the other direction. We can start from a "topological" semantics that specifies the conditional independence relationships encoded by the graph structure, and from these we can derive the "numerical" semantics. The topological semantics is given by either of the following specifications, which are equivalent:[2]

DESCENDANTS

1. A node is conditionally independent of its non-**descendants**, given its parents. For example, in Figure 14.2, *JohnCalls* is independent of *Burglary* and *Earthquake*, given the value of *Alarm*.

MARKOV BLANKET

2. A node is conditionally independent of all other nodes in the network, given its parents, children, and children's parents—that is, given its **Markov blanket**. For example, *Burglary* is independent of *JohnCalls* and *MaryCalls*, given *Alarm* and *Earthquake*.

These specifications are illustrated in Figure 14.4. From these conditional independence assertions and the CPTs, the full joint distribution can be reconstructed; thus, the "numerical" semantics and the "topological" semantics are equivalent.

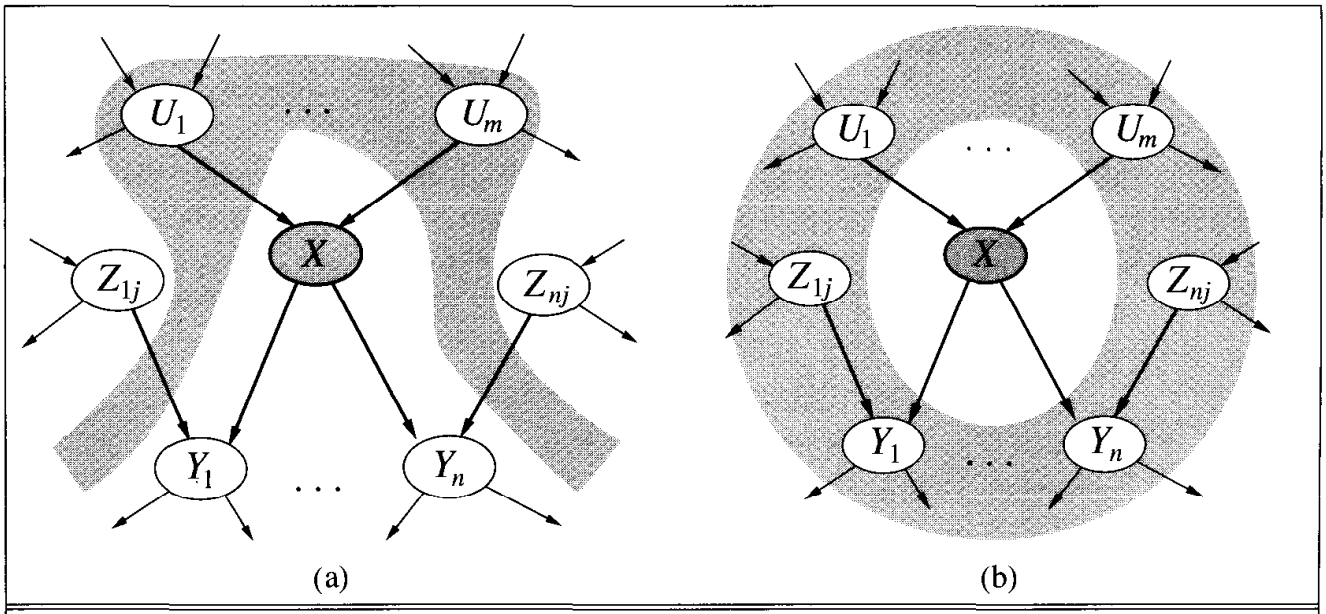


Figure 14.4 (a) A node X is conditionally independent of its non-descendants (e.g., the Z_{ij}s) given its parents (the U_is shown in the gray area). (b) A node X is conditionally independent of all other nodes in the network given its Markov blanket (the gray area).

[2] There is also a general topological criterion called **d-separation** for deciding whether a set of nodes **X** is independent of another set **Y**, given a third set **Z**. The criterion is rather complicated and is not needed for deriving the algorithms in this chapter, so we omit it. Details may be found in Russell and Norvig (1995) or Pearl (1988). Shachter (1998) gives a more intuitive method of ascertaining d-separation.

14.3 Efficient Representation of Conditional Distributions

Even if the maximum number of parents k is smallish, filling in the CPT for a node requires up to $O(2^k)$ numbers and perhaps a great deal of experience with all the possible conditioning cases. In fact, this is a worst-case scenario in which the relationship between the parents and the child is completely arbitrary. Usually, such relationships are describable by a **canonical distribution** that fits some standard pattern. In such cases, the complete table can be specified by naming the pattern and perhaps supplying a few parameters—much easier than supplying an exponential number of parameters.

CANONICAL DISTRIBUTION

DETERMINISTIC NODES

The simplest example is provided by **deterministic nodes**. A deterministic node has its value specified exactly by the values of its parents, with no uncertainty. The relationship can be a logical one: for example, the relationship between the parent nodes *Canadian*, *US*, *Mexican* and the child node *NorthAmerican* is simply that the child is the disjunction of the parents. The relationship can also be numerical: for example, if the parent nodes are the prices of a particular model of car at several dealers, and the child node is the price that a bargain hunter ends up paying, then the child node is the minimum of the parent values; or if the parent nodes are the inflows (rivers, runoff, precipitation) into a lake and the outflows (rivers, evaporation, seepage) from the lake and the child is the change in the water level of the lake, then the value of the child is the difference between the inflow parents and the outflow parents.

NOISY-OR

Uncertain relationships can often be characterized by so-called "noisy" logical relationships. The standard example is the **noisy-OR** relation, which is a generalization of the logical OR. In propositional logic, we might say that *Fever* is true if and only if *Cold*, *Flu*, or *Malaria* is true. The noisy-OR model allows for uncertainty about the ability of each parent to cause the child to be true—the causal relationship between parent and child may be *inhibited*, and so a patient could have a cold, but not exhibit a fever. The model makes two assumptions. First, it assumes that all the possible causes are listed. (This is not as strict as it seems, because we can always add a so-called **leak node** that covers "miscellaneous causes.") Second, it assumes that inhibition of each parent is independent of inhibition of any other parents: for example, whatever inhibits *Malaria* from causing a fever is independent of whatever inhibits *Flu* from causing a fever. Given these assumptions, *Fever* is *false* if and only if all its *true* parents are inhibited, and the probability of this is the product of the inhibition probabilities for each parent. Let us suppose these individual inhibition probabilities are as follows:

LEAK NODE

$$P(\neg fever|cold, \neg flu, \neg malaria) = 0.6\ ,$$
$$P(\neg fever|\neg cold, flu, \neg malaria) = 0.2\ ,$$
$$P(\neg fever|\neg cold, \neg flu, malaria) = 0.1\ .$$

Then, from this information and the noisy-OR assumptions, the entire CPT can be built. The following table shows how:

Cold	*Flu*	*Malaria*	$P(Fever)$	$P(\neg Fever)$
F	F	F	0.0	1.0
F	F	T	0.9	**0.1**
F	T	F	0.8	**0.2**
F	T	T	0.98	$0.02 = 0.2 \times 0.1$
T	F	F	0.4	**0.6**
T	F	T	0.94	$0.06 = 0.6 \times 0.1$
T	T	F	0.88	$0.12 = 0.6 \times 0.2$
T	T	T	0.988	$0.012 = 0.6 \times 0.2 \times 0.1$

In general, noisy logical relationships in which a variable depends on k parents can be described using $O(k)$ parameters instead of $O(2^k)$ for the full conditional probability table. This makes assessment and learning much easier. For example, the CPCS network (Pradhan *et al.*, 1994) uses noisy-OR and noisy-MAX distributions to model relationships among diseases and symptoms in internal medicine. With 448 nodes and 906 links, it requires only 8,254 values instead of 133,931,430 for a network with full CPTs.

Bayesian nets with continuous variables

Many real-world problems involve continuous quantities, such as height, mass, temperature, and money; in fact, much of statistics deals with random variables whose domains are continuous. By definition, continuous variables have an infinite number of possible values, so it is impossible to specify conditional probabilities explicitly for each value. One possible way to handle continuous variables is to avoid them by using **discretization**—that is, dividing up the possible values into a fixed set of intervals. For example, temperatures could be divided into (<0°C), (0°C-100°C), and (>100°C). Discretization is sometimes an adequate solution, but often results in a considerable loss of accuracy and very large CPTs. The other solution is to define standard families of probability density functions (see Appendix A) that are specified by a finite number of **parameters**. For example, a Gaussian (or normal) distribution $N(\mu, \sigma^2)(x)$ has the mean μ and the variance σ^2 as parameters.

DISCRETIZATION

PARAMETERS

A network with both discrete and continuous variables is called a **hybrid Bayesian network**. To specify a hybrid network, we have to specify two new kinds of distributions: the conditional distribution for a continuous variable given discrete or continuous parents; and the conditional distribution for a discrete variable given continuous parents. Consider the simple example in Figure 14.5, in which a customer buys some fruit depending on its cost, which depends in turn on the size of the harvest and whether the government's subsidy scheme is operating. The variable *Cost* is continuous and has continuous and discrete parents; the variable *Buys* is discrete and has a continuous parent.

HYBRID BAYESIAN NETWORK

For the *Cost* variable, we need to specify $\mathbf{P}(Cost|Harvest, Subsidy)$. The discrete parent is handled by explicit enumeration—that is, specifying both $P(Cost|Harvest, subsidy)$ and $P(Cost|Harvest, \neg subsidy)$. To handle *Harvest*, we specify how the distribution over the cost c depends on the continuous value h of *Harvest*. In other words, we specify the *parameters* of the cost distribution as a function of h. The most common choice is the **linear**

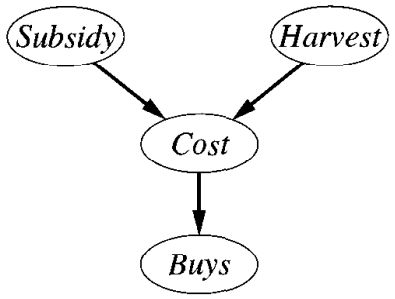


Figure 14.5 A simple network with discrete variables (*Subsidy* and *Buys*) and continuous variables (*Harvest* and *Cost*).

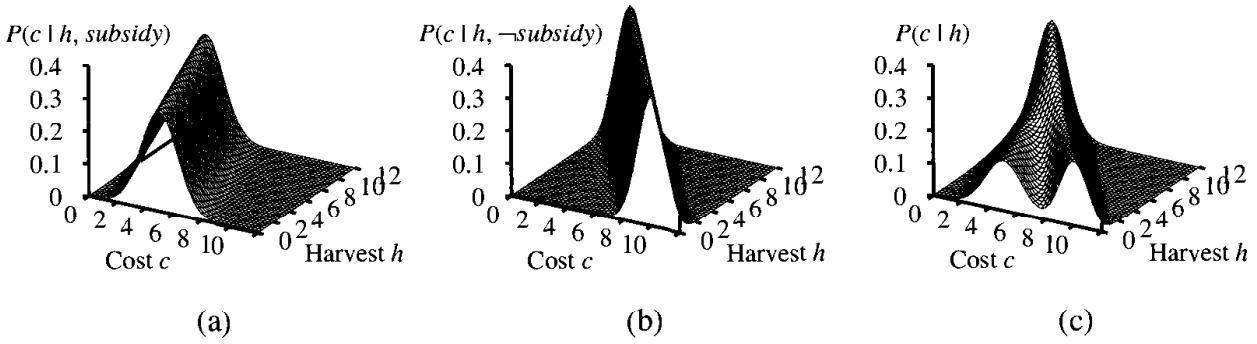


Figure 14.6 The graphs in (a) and (b) show the probability distribution over *Cost* as a function of *Harvest* size, with *Subsidy* true and false respectively. Graph (c) shows the distribution $P(Cost|Harvest)$, obtained by summing over the two subsidy cases.

LINEAR GAUSSIAN

Gaussian distribution, in which the child has a Gaussian distribution whose mean μ varies linearly with the value of the parent and whose standard deviation σ is fixed. We need two distributions, one for *subsidy* and one for $\neg subsidy$, with different parameters:

$$P(c|h, subsidy) = N(a_t h + b_t, \sigma_t^2)(c) = \frac{1}{\sigma_t\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{c-(a_t h+b_t)}{\sigma_t}\right)^2}$$

$$P(c|h, \neg subsidy) = N(a_f h + b_f, \sigma_f^2)(c) = \frac{1}{\sigma_f\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{c-(a_f h+b_f)}{\sigma_f}\right)^2}$$

For this example, then, the conditional distribution for *Cost* is specified by naming the linear Gaussian distribution and providing the parameters a_t, b_t, σ_t, a_f, b_f, and σ_f. Figures 14.6(a) and (b) show these two relationships. Notice that in each case the slope is negative, because price decreases as supply increases. (Of course, the assumption of linearity implies that the price becomes negative at some point; the linear model is reasonable only if the harvest size is limited to a narrow range.) Figure 14.6(c) shows the distribution $P(c|h)$, averaging over the two possible values of *Subsidy* and assuming that each has prior probability 0.5. This shows that even with very simple models, quite interesting distributions can be represented.

The linear Gaussian conditional distribution has some special properties. A network containing only continuous variables with linear Gaussian distributions has a joint distribu-

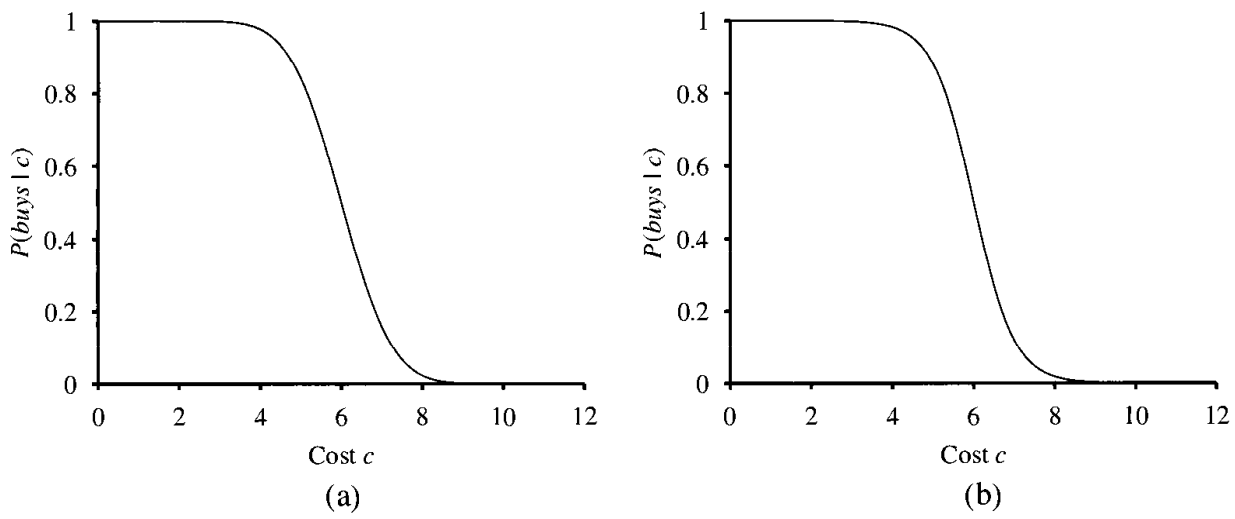


Figure 14.7 (a) A probit distribution for the probability of *Buys* given *Cost*, with $\mu = 6.0$ and $\sigma = 1.0$. (b) A logit distribution with the same parameters.

tion that is a multivariate Gaussian distribution over all the variables (Exercise 14.5).[3] (A multivariate Gaussian distribution is a surface in more than one dimension that has a peak at the mean (in n dimensions) and drops off on all sides.) When discrete variables are added (provided that no discrete variable is a child of a continuous variable), the network defines a **conditional Gaussian**, or CG, distribution: given any assignment to the discrete variables, the distribution over the continuous variables is a multivariate Gaussian.

CONDITIONAL GAUSSIAN

Now we turn to the distributions for discrete variables with continuous parents. Consider, for example, the *Buys* node in Figure 14.5. It seems reasonable to assume that the customer will buy if the cost is low and will not buy if it is high and that the probability of buying varies smoothly in some intermediate region. In other words, the conditional distribution is like a "soft" threshold function. One way to make soft thresholds is to use the *integral* of the standard normal distribution:

$$\Phi(x) = \int_{-\infty}^{x} N(0,1)(x)dx \ .$$

Then the probability of *Buys* given *Cost* might be

$$P(buys \mid Cost = c) = \Phi((-c + \mu)/\sigma)$$

which means that the cost threshold occurs around μ, the width of the threshold region is proportional to σ, and the probability of buying decreases as cost increases.

PROBIT DISTRIBUTION

This **probit distribution** is illustrated in Figure 14.7(a). The form can be justified by proposing that the underlying decision process has a hard threshold, but that the precise location of the threshold is subject to random Gaussian noise. An alternative to the probit model is the **logit distribution**, which uses the **sigmoid function** to produce a soft threshold:

LOGIT DISTRIBUTION

SIGMOID FUNCTION

$$P(buys \mid Cost = c) = \frac{1}{1 + exp(-2\frac{-c+\mu}{\sigma})} \ .$$

[3] It follows that inference in linear Gaussian networks takes only $O(n^3)$ time in the worst case, regardless of the network topology. In Section 14.4, we will see that inference for networks of discrete variables is NP-hard.

This is illustrated in Figure 14.7(b). The two distributions look similar, but the logit actually has much longer "tails." The probit is often a better fit to real situations, but the logit is sometimes easier to deal with mathematically. It is used widely in neural networks (Chapter 20). Both probit and logit can be generalized to handle multiple continuous parents by taking a linear combination of the parent values. Extensions for a multivalued discrete child are explored in Exercise 14.6.

14.4 Exact Inference in Bayesian Networks

EVENT

HIDDEN VARIABLES

The basic task for any probabilistic inference system is to compute the posterior probability distribution for a set of **query variables**, given some observed **event**—that is, some assignment of values to a set of **evidence variables**. We will use the notation introduced in Chapter 13: X denotes the query variable; $\mathbf{E}$ denotes the set of evidence variables $E_1, \ldots, E_m$, and $\mathbf{e}$ is a particular observed event; $\mathbf{Y}$ will denote the nonevidence variables $Y_1, \ldots, Y_l$ (sometimes called the **hidden variables**). Thus, the complete set of variables $\mathbf{X} = \{X\} \cup \mathbf{E} \cup \mathbf{Y}$. A typical query asks for the posterior probability distribution $\mathbf{P}(X|\mathbf{e})$.[4]

In the burglary network, we might observe the event in which $JohnCalls = true$ and $MaryCalls = true$. We could then ask for, say, the probability that a burglary has occurred:

$$\mathbf{P}(Burglary|JohnCalls = true, MaryCalls = true) = \langle 0.284, 0.716 \rangle \ .$$

In this section we will discuss exact algorithms for computing posterior probabilities and will consider the complexity of this task. It turns out that the general case is intractable, so Section 14.5 covers methods for approximate inference.

Inference by enumeration

Chapter 13 explained that any conditional probability can be computed by summing terms from the full joint distribution. More specifically, a query $\mathbf{P}(X|\mathbf{e})$ can be answered using Equation (13.6), which we repeat here for convenience:

$$\mathbf{P}(X|\mathbf{e}) = \alpha \mathbf{P}(X, \mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}(X, \mathbf{e}, \mathbf{y}) \ .$$

Now, a Bayesian network gives a complete representation of the full joint distribution. More specifically, Equation (14.1) shows that the terms $P(x, \mathbf{e}, \mathbf{y})$ in the joint distribution can be written as products of conditional probabilities from the network. Therefore, *a query can be answered using a Bayesian network by computing sums of products of conditional probabilities from the network.*

In Figure 13.4, an algorithm, ENUMERATE-JOINT-ASK, was given for inference by enumeration from the full joint distribution. The algorithm takes as input a full joint distribution $\mathbf{P}$ and looks up values therein. It is a simple matter to modify the algorithm so that it takes

[4] We will assume that the query variable is not among the evidence variables; if it is, then the posterior distribution for X simply gives probability 1 to the observed value. For simplicity, we have also assumed that the query is just a single variable. Our algorithms can be extended easily to handle a joint query over several variables.

as input a Bayesian network bn and "looks up" joint entries by multiplying the corresponding CPT entries from bn.

Consider the query $\mathbf{P}(Burglary|JohnCalls = true, MaryCalls = true)$. The hidden variables for this query are $Earthquake$ and $Alarm$. From Equation (13.6), using initial letters for the variables in order to shorten the expressions, we have[5]

$$\mathbf{P}(B|j,m) = \alpha\,\mathbf{P}(B,j,m) = \alpha \sum_e \sum_a \mathbf{P}(B,e,a,j,m) .$$

The semantics of Bayesian networks (Equation (14.1)) then gives us an expression in terms of CPT entries. For simplicity, we will do this just for $Burglary = true$:

$$P(b|j,m) = \alpha \sum_e \sum_a P(b)P(e)P(a|b,e)P(j|a)P(m|a) .$$

To compute this expression, we have to add four terms, each computed by multiplying five numbers. In the worst case, where we have to sum out almost all the variables, the complexity of the algorithm for a network with n Boolean variables is $O(n2^n)$.

An improvement can be obtained from the following simple observations: the $P(b)$ term is a constant and can be moved outside the summations over a and e, and the $P(e)$ term can be moved outside the summation over a. Hence, we have

$$P(b|j,m) = \alpha\, P(b) \sum_e P(e) \sum_a P(a|b,e)P(j|a)P(m|a) . \tag{14.3}$$

This expression can be evaluated by looping through the variables in order, multiplying CPT entries as we go. For each summation, we also need to loop over the variable's possible values. The structure of this computation is shown in Figure 14.8. Using the numbers from Figure 14.2, we obtain $P(b|j,m) = \alpha \times 0.00059224$. The corresponding computation for $\neg b$ yields $\alpha \times 0.0014919$; hence

$$\mathbf{P}(B|j,m) = \alpha\,\langle 0.00059224, 0.0014919\rangle \approx \langle 0.284, 0.716\rangle .$$

That is, the chance of a burglary, given calls from both neighbors, is about 28%.

The evaluation process for the expression in Equation (14.3) is shown as an expression tree in Figure 14.8. The ENUMERATION-ASK algorithm in Figure 14.9 evaluates such trees using depth-first recursion. Thus, the space complexity of ENUMERATION-ASK is only linear in the number of variables—effectively, the algorithm sums over the full joint distribution without ever constructing it explicitly. Unfortunately, its time complexity for a network with n Boolean variables is always $O(2^n)$—better than the $O(n2^n)$ for the simple approach described earlier, but still rather grim. One thing to note about the tree in Figure 14.8 is that it makes explicit the *repeated subexpressions* that are evaluated by the algorithm. The products $P(j|a)P(m|a)$ and $P(j|\neg a)P(m|\neg a)$ are computed twice, once for each value of e. The next section describes a general method that avoids such wasted computations.

[5] An expression such as $\sum_e P(a,e)$ means to sum $P(A = a, E = e)$ for all possible values of e. There is an ambiguity in that $P(e)$ is used to mean both $P(E = true)$ and $P(E = e)$, but it should be clear from context which is intended; in particular, in the context of a sum the latter is intended.

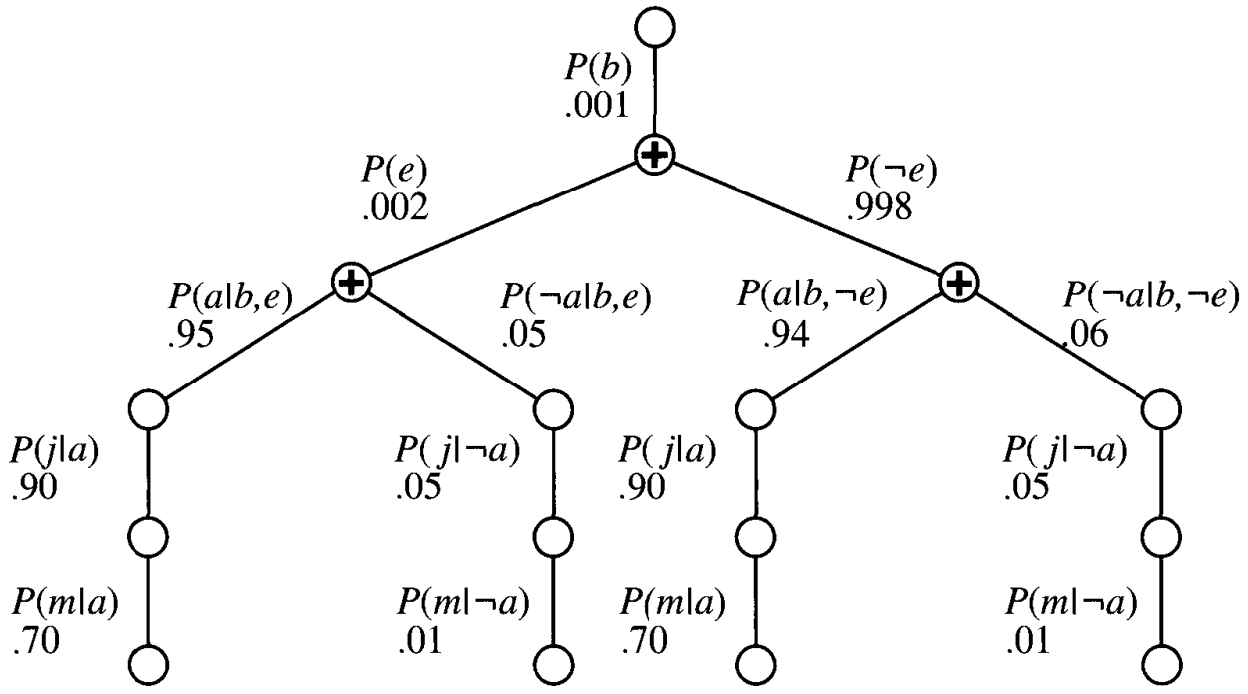


Figure 14.8 The structure of the expression shown in Equation (14.3). The evaluation proceeds top-down, multiplying values along each path and summing at the "+" nodes. Notice the repetition of the paths for j and m.

function ENUMERATION-ASK(X, **e**, bn) **returns** a distribution over X
  inputs: X, the query variable
    e, observed values for variables **E**
    bn, a Bayes net with variables $\{X\} \cup \mathbf{E} \cup \mathbf{Y}$ /* **Y** = *hidden variables* */

  $\mathbf{Q}(X) \leftarrow$ a distribution over X, initially empty
  for each value x_i of X **do**
    extend **e** with value x_i for X
    $\mathbf{Q}(x_i) \leftarrow$ ENUMERATE-ALL(VARS[bn], **e**)
  return NORMALIZE($\mathbf{Q}(X)$)

function ENUMERATE-ALL($vars$, **e**) **returns** a real number
  if EMPTY?($vars$) **then return** 1.0
  $Y \leftarrow$ FIRST($vars$)
  if Y has value y in **e**
    then return $P(y \mid parents(Y)) \times$ ENUMERATE-ALL(REST($vars$), **e**)
    else return $\sum_y P(y \mid parents(Y)) \times$ ENUMERATE-ALL(REST($vars$), $\mathbf{e}_y$)
      where $\mathbf{e}_y$ is **e** extended with $Y = y$

Figure 14.9 The enumeration algorithm for answering queries on Bayesian networks.

The variable elimination algorithm

The enumeration algorithm can be improved substantially by eliminating repeated calculations of the kind illustrated in Figure 14.8. The idea is simple: do the calculation once and save the results for later use. This is a form of dynamic programming. There are several versions of this approach; we present the **variable elimination** algorithm, which is the simplest. Variable elimination works by evaluating expressions such as Equation (14.3) in *right-to-left* order (that is, *bottom-up* in Figure 14.8). Intermediate results are stored, and summations over each variable are done only for those portions of the expression that depend on the variable.

VARIABLE ELIMINATION

Let us illustrate this process for the burglary network. We evaluate the expression

$$\mathbf{P}(B|j,m) = \alpha \underbrace{\mathbf{P}(B)}_{B} \sum_e \underbrace{P(e)}_{E} \sum_a \underbrace{\mathbf{P}(a|B,e)}_{A} \underbrace{P(j|a)}_{J} \underbrace{P(m|a)}_{M} .$$

Notice that we have annotated each part of the expression with the name of the associated variable; these parts are called **factors**. The steps are as follows:

FACTORS

- The factor for M, $P(m|a)$, does not require summing over M (because M's value is already fixed). We store the probability, given each value of a, in a two-element vector,
$$\mathbf{f}_M(A) = \begin{pmatrix} P(m|a) \\ P(m|\neg a) \end{pmatrix} .$$
(The $\mathbf{f}_M$ means that M was used to produce $\mathbf{f}$.)
- Similarly, we store the factor for J as the two-element vector $\mathbf{f}_J(A)$.
- The factor for A is $\mathbf{P}(a|B,e)$, which will be a $2 \times 2 \times 2$ matrix $\mathbf{f}_A(A,B,E)$.
- Now we must sum out A from the product of these three factors. This will give us a 2×2 matrix whose indices range over just B and E. We put a bar over A in the name of the matrix to indicate that A has been summed out:
$$\begin{aligned} \mathbf{f}_{\bar{A}JM}(B,E) &= \sum_a \mathbf{f}_A(a,B,E) \times \mathbf{f}_J(a) \times \mathbf{f}_M(a) \\ &= \mathbf{f}_A(a,B,E) \times \mathbf{f}_J(a) \times \mathbf{f}_M(a) \\ &+ \mathbf{f}_A(\neg a,B,E) \times \mathbf{f}_J(\neg a) \times \mathbf{f}_M(\neg a) . \end{aligned}$$
The multiplication process used here is called a **pointwise product** and will be described shortly.

POINTWISE PRODUCT

- We process E in the same way: sum out E from the product of $\mathbf{f}_E(E)$ and $\mathbf{f}_{\bar{A}JM}(B,E)$:
$$\begin{aligned} \mathbf{f}_{\bar{E}\bar{A}JM}(B) &= \mathbf{f}_E(e) \times \mathbf{f}_{\bar{A}JM}(B,e) \\ &+ \mathbf{f}_E(\neg e) \times \mathbf{f}_{\bar{A}JM}(B,\neg e) . \end{aligned}$$
- Now we can compute the answer simply by multiplying the factor for B (i.e., $\mathbf{f}_B(B) = \mathbf{P}(B)$), by the accumulated matrix $\mathbf{f}_{\bar{E}\bar{A}JM}(B)$:
$$\mathbf{P}(B|j,m) = \alpha \mathbf{f}_B(B) \times \mathbf{f}_{\bar{E}\bar{A}JM}(B) .$$

Exercise 14.7(a) asks you to check that this process yields the correct answer.

Examining this sequence of steps, we see that there are two basic computational operations required: pointwise product of a pair of factors, and summing out a variable from a product of factors.

The pointwise product is not matrix multiplication, nor is it element-by-element multiplication. The pointwise product of two factors $\mathbf{f}_1$ and $\mathbf{f}_2$ yields a new factor $\mathbf{f}$ whose variables are the *union* of the variables in $\mathbf{f}_1$ and $\mathbf{f}_2$. Suppose the two factors have variables $Y_1, \ldots, Y_k$ in common. Then we have

$$\mathbf{f}(X_1 \ldots X_j, Y_1 \ldots Y_k, Z_1 \ldots Z_l) = \mathbf{f}_1(X_1 \ldots X_j, Y_1 \ldots Y_k)\, \mathbf{f}_2(Y_1 \ldots Y_k, Z, \ldots Z_l).$$

If all the variables are binary, then $\mathbf{f}_1$ and $\mathbf{f}_2$ have 2^{j+k} and 2^{k+l} entries respectively, and the pointwise product has 2^{j+k+l} entries. For example, given two factors $\mathbf{f}_1(A, B)$ and $\mathbf{f}_2(B, C)$ with probability distributions shown below, the pointwise product $\mathbf{f}_1 \times \mathbf{f}_2$ is given as $\mathbf{f}_3(A, B, C)$:

A	B	$\mathbf{f}_1(A,B)$	B	C	$\mathbf{f}_2(B,C)$	A	B	C	$\mathbf{f}_3(A,B,C)$
T	T	.3	T	T	.2	T	T	T	$.3 \times .2$
T	F	.7	T	F	.8	T	T	F	$.3 \times .8$
F	T	.9	F	T	.6	T	F	T	$.7 \times .6$
F	F	.1	F	F	.4	T	F	F	$.7 \times .4$
						F	T	T	$.9 \times .2$
						F	T	F	$.9 \times .8$
						F	F	T	$.1 \times .6$
						F	F	F	$.1 \times .4$

Summing out a variable from a product of factors is also a straightforward computation. The only trick is to notice that any factor that does *not* depend on the variable to be summed out can be moved outside the summation process. For example,

$$\sum_e \mathbf{f}_E(e) \times \mathbf{f}_A(A, B, e) \times \mathbf{f}_J(A) \times \mathbf{f}_M(A) = \mathbf{f}_J(A) \times \mathbf{f}_M(A) \times \sum_e \mathbf{f}_E(e) \times \mathbf{f}_A(A, B, e)\ .$$

Now the pointwise product inside the summation is computed, and the variable is summed out of the resulting matrix:

$$\mathbf{f}_J(A) \times \mathbf{f}_M(A) \times \sum_e \mathbf{f}_E(e) \times \mathbf{f}_A(A, B, e) = \mathbf{f}_J(A) \times \mathbf{f}_M(A) \times \mathbf{f}_{\bar{E},A}(A, B)\ .$$

Notice that matrices are *not* multiplied until we need to sum out a variable from the accumulated product. At that point, we multiply just those matrices that include the variable to be summed out. Given routines for pointwise product and summing out, the variable elimination algorithm itself can be written quite simply, as shown in Figure 14.10.

Let us consider one more query: $P(JohnCalls|Burglary = true)$. As usual, the first step is to write out the nested summation:

$$P(J|b) = \alpha\, P(b) \sum_e P(e) \sum_a P(a|b, e) P(J|a) \sum_m P(m|a)\ .$$

If we evaluate this expression from right to left, we notice something interesting: $\sum_m P(m|a)$ is equal to 1 by definition! Hence, there was no need to include it in the first place; the variable M is *irrelevant* to this query. Another way of saying this is that the result of the query $P(JohnCalls|Burglary = true)$ is unchanged if we remove $MaryCalls$ from the network altogether. In general, we can remove any leaf node that is not a query variable or an evidence

```
function ELIMINATION-ASK(X, e, bn) returns a distribution over X
  inputs: X, the query variable
          e, evidence specified as an event
          bn, a Bayesian network specifying joint distribution P(X_1, ..., X_n)

  factors ← [ ]; vars ← REVERSE(VARS[bn])
  for each var in vars do
      factors ← [MAKE-FACTOR(var, e)|factors]
      if var is a hidden variable then factors ← SUM-OUT(var, factors)
  return NORMALIZE(POINTWISE-PRODUCT(factors))
```

Figure 14.10 The variable elimination algorithm for answering queries on Bayesian networks.

variable. After its removal, there may be some more leaf nodes, and these too may be irrelevant. Continuing this process, we eventually find that *every variable that is not an ancestor of a query variable or evidence variable is irrelevant to the query.* A variable elimination algorithm can therefore remove all these variables before evaluating the query.

The complexity of exact inference

We have argued that variable elimination is more efficient than enumeration because it avoids repeated computations (as well as dropping irrelevant variables). The time and space requirements of variable elimination are dominated by the size of the largest factor constructed during the operation of the algorithm. This in turn is determined by the order of elimination of variables and by the structure of the network.

The burglary network of Figure 14.2 belongs to the family of networks in which there is at most one undirected path between any two nodes in the network. These are called **singly connected** networks or **polytrees**, and they have a particularly nice property: *The time and space complexity of exact inference in polytrees is linear in the size of the network.* Here, the size is defined as the number of CPT entries; if the number of parents of each node is bounded by a constant, then the complexity will also be linear in the number of nodes. These results hold for any ordering consistent with the topological ordering of the network (Exercise 14.7).

SINGLY CONNECTED

POLYTREES

MULTIPLY CONNECTED

For **multiply connected** networks, such as that of Figure 14.11(a), variable elimination can have exponential time and space complexity in the worst case, even when the number of parents per node is bounded. This is not surprising when one considers that, *because it includes inference in propositional logic as a special case, inference in Bayesian networks is NP-hard.* In fact, it can be shown (Exercise 14.8) that the problem is as hard as that of computing the *number* of satisfying assignments for a propositional logic formula. This means that it is #P-hard ("number-P hard")—that is, strictly harder than NP-complete problems.

There is a close connection between the complexity of Bayesian network inference and the complexity of constraint satisfaction problems (CSPs). As we discussed in Chapter 5, the difficulty of solving a discrete CSP is related to how "tree-like" its constraint graph is.

Measures such as **hypertree width**, which bound the complexity of solving a CSP, can also be applied directly to Bayesian networks. Moreover, the variable elimination algorithm can be generalized to solve CSPs as well as Bayesian networks.

Clustering algorithms

The variable elimination algorithm is simple and efficient for answering individual queries. If we want to compute posterior probabilities for all the variables in a network, however, it can be less efficient. For example, in a polytree network, one would need to issue $O(n)$ queries costing $O(n)$ each, for a total of $O(n^2)$ time. Using **clustering** algorithms (also known as **join tree** algorithms), the time can be reduced to $O(n)$. For this reason, these algorithms are widely used in commercial Bayesian network tools.

CLUSTERING

JOIN TREE

The basic idea of clustering is to join individual nodes of the network to form cluster nodes in such a way that the resulting network is a polytree. For example, the multiply connected network shown in Figure 14.11(a) can be converted into a polytree by combining the *Sprinkler* and *Rain* node into a cluster node called *Sprinkler*+*Rain*, as shown in Figure 14.11(b). The two Boolean nodes are replaced by a meganode that takes on four possible values: TT, TF, FT, and FF. The meganode has only one parent, the Boolean variable *Cloudy*, so there are two conditioning cases.

Once the network is in polytree form, a special-purpose inference algorithm is applied. Essentially, the algorithm is a form of constraint propagation (see Chapter 5) where the constraints ensure that neighboring clusters agree on the posterior probability of any variables that they have in common. With careful bookkeeping, this algorithm is able to compute posterior probabilities for all the nonevidence nodes in the network in time $O(n)$, where n is now the size of the modified network. However, the NP-hardness of the problem has not disappeared: if a network requires exponential time and space with variable elimination, then the CPTs in the clustered network will require exponential time and space to construct.

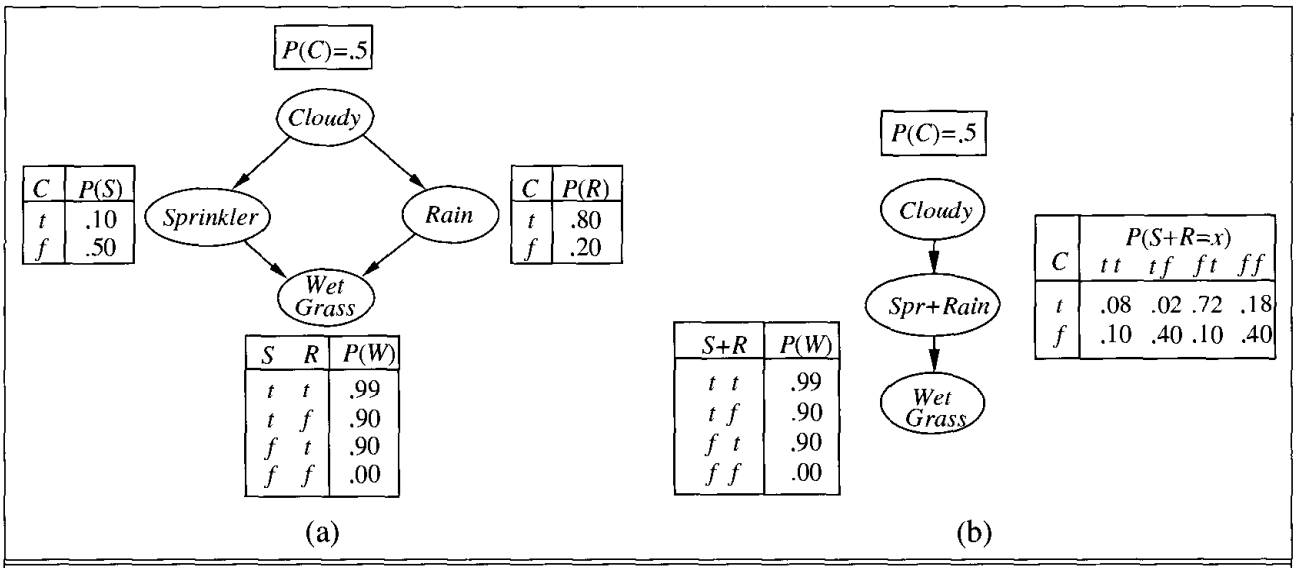


Figure 14.11 (a) A multiply connected network with conditional probability tables. (b) A clustered equivalent of the multiply connected network.

14.5 APPROXIMATE INFERENCE IN BAYESIAN NETWORKS

MONTE CARLO

Given the intractability of exact inference in large, multiply connected networks, it is essential to consider approximate inference methods. This section describes randomized sampling algorithms, also called **Monte Carlo** algorithms, that provide approximate answers whose accuracy depends on the number of samples generated. In recent years, Monte Carlo algorithms have become widely used in computer science to estimate quantities that are difficult to calculate exactly. For example, the simulated annealing algorithm described in Chapter 4 is a Monte Carlo method for optimization problems. In this section, we are interested in sampling applied to the computation of posterior probabilities. We describe two families of algorithms: direct sampling and Markov chain sampling. Two other approaches—variational methods and loopy propagation—are mentioned in the notes at the end of the chapter.

Direct sampling methods

The primitive element in any sampling algorithm is the generation of samples from a known probability distribution. For example, an unbiased coin can be thought of as a random variable $Coin$ with values $\langle heads, tails \rangle$ and a prior distribution $\mathbf{P}(Coin) = \langle 0.5, 0.5 \rangle$. Sampling from this distribution is exactly like flipping the coin: with probability 0.5 it will return $heads$, and with probability 0.5 it will return $tails$. Given a source of random numbers in the range $[0, 1]$, it is a simple matter to sample any distribution on a single variable. (See Exercise 14.9.)

The simplest kind of random sampling process for Bayesian networks generates events from a network that has no evidence associated with it. The idea is to sample each variable in turn, in topological order. The probability distribution from which the value is sampled is conditioned on the values already assigned to the variable's parents. This algorithm is shown in Figure 14.12. We can illustrate its operation on the network in Figure 14.11(a), assuming an ordering $[Cloudy, Sprinkler, Rain, WetGrass]$:

1. Sample from $\mathbf{P}(Cloudy) = \langle 0.5, 0.5 \rangle$; suppose this returns $true$.
2. Sample from $\mathbf{P}(Sprinkler | Cloudy = true) = \langle 0.1, 0.9 \rangle$; suppose this returns $false$.
3. Sample from $\mathbf{P}(Rain | Cloudy = true) = \langle 0.8, 0.2 \rangle$; suppose this returns $true$.
4. Sample from $\mathbf{P}(WetGrass | Sprinkler = false, Rain = true) = \langle 0.9, 0.1 \rangle$; suppose this returns $true$.

In this case, PRIOR-SAMPLE returns the event $[true, false, true, true]$.

It is easy to see that PRIOR-SAMPLE generates samples from the prior joint distribution specified by the network. First, let $S_{PS}(x_1, \ldots, x_n)$ be the probability that a specific event is generated by the PRIOR-SAMPLE algorithm. *Just looking at the sampling process*, we have

$$S_{PS}(x_1 \ldots x_n) = \prod_{i=1}^{n} P(x_i | parents(X_i))$$

because each sampling step depends only on the parent values. This expression should look

function PRIOR-SAMPLE(bn) **returns** an event sampled from the prior specified by bn
inputs: bn, a Bayesian network specifying joint distribution $\mathbf{P}(X_1, \ldots, X_n)$

$\mathbf{x} \leftarrow$ an event with n elements
for $i = 1$ **to** n **do**
  $x_i \leftarrow$ a random sample from $\mathbf{P}(X_i \mid parents(X_i))$
return $\mathbf{x}$

Figure 14.12 A sampling algorithm that generates events from a Bayesian network.

familiar, because it is also the probability of the event according to the Bayesian net's representation of the joint distribution, as stated in Equation (14.1). That is, we have

$$S_{PS}(x_1 \ldots x_n) = P(x_1 \ldots x_n) \ .$$

This simple fact makes it very easy to answer questions by using samples.

In any sampling algorithm, the answers are computed by counting the actual samples generated. Suppose there are N total samples, and let $N(x_1, \ldots, x_n)$ be the frequency of the specific event $x_1, \ldots, x_n$. We expect this frequency to converge, in the limit, to its expected value according to the sampling probability:

$$\lim_{N \to \infty} \frac{N_{PS}(x_1, \ldots, x_n)}{N} = S_{PS}(x_1, \ldots, x_n) = P(x_1, \ldots, x_n) \ . \tag{14.4}$$

For example, consider the event produced earlier: $[true, false, true, true]$. The sampling probability for this event is

$$S_{PS}(true, false, true, true) = 0.5 \times 0.9 \times 0.8 \times 0.9 = 0.324 \ .$$

Hence, in the limit of large N, we expect 32.4% of the samples to be of this event.

Whenever we use an approximate equality ("$\approx$") in what follows, we mean it in exactly this sense—that the estimated probability becomes exact in the large-sample limit. Such an estimate is called **consistent**. For example, one can produce a consistent estimate of the probability of any partially specified event $x_1, \ldots, x_m$, where $m \leq n$, as follows:

CONSISTENT

$$P(x_1, \ldots, x_m) \approx N_{PS}(x_1, \ldots, x_m)/N \ . \tag{14.5}$$

That is, the probability of the event can be estimated as the fraction of all complete events generated by the sampling process that match the partially specified event. For example, if we generate 1000 samples from the sprinkler network, and 511 of them have $Rain = true$, then the estimated probability of rain, written as $\hat{P}(Rain = true)$, is 0.511.

Rejection sampling in Bayesian networks

REJECTION SAMPLING

Rejection sampling is a general method for producing samples from a hard-to-sample distribution given an easy-to-sample distribution. In its simplest form, it can be used to compute conditional probabilities—that is, to determine $P(X|\mathbf{e})$. The REJECTION-SAMPLING algorithm is shown in Figure 14.13. First, it generates samples from the prior distribution specified

```
function REJECTION-SAMPLING(X, e, bn, N) returns an estimate of P(X|e)
  inputs: X, the query variable
          e, evidence specified as an event
          bn, a Bayesian network
          N, the total number of samples to be generated
  local variables: N, a vector of counts over X, initially zero

  for j = 1 to N do
    x ← PRIOR-SAMPLE(bn)
    if x is consistent with e then
      N[x] ← N[x]+1 where x is the value of X in x
  return NORMALIZE(N[X])
```

Figure 14.13 The rejection sampling algorithm for answering queries given evidence in a Bayesian network.

by the network. Then, it rejects all those that do not match the evidence. Finally, the estimate $\hat{P}(X = x|\mathbf{e})$ is obtained by counting how often $X = x$ occurs in the remaining samples.

Let $\hat{\mathbf{P}}(X|\mathbf{e})$ be the estimated distribution that the algorithm returns. From the definition of the algorithm, we have

$$\hat{\mathbf{P}}(X|\mathbf{e}) = \alpha \mathbf{N}_{PS}(X, \mathbf{e}) = \frac{\mathbf{N}_{PS}(X, \mathbf{e})}{N_{PS}(\mathbf{e})} .$$

From Equation (14.5), this becomes

$$\hat{\mathbf{P}}(X|\mathbf{e}) \approx \frac{\mathbf{P}(X, \mathbf{e})}{P(\mathbf{e})} = \mathbf{P}(X|\mathbf{e}) .$$

That is, rejection sampling produces a consistent estimate of the true probability.

Continuing with our example from Figure 14.11(a), let us assume that we wish to estimate $\mathbf{P}(Rain|Sprinkler = true)$, using 100 samples. Of the 100 that we generate, suppose that 73 have $Sprinkler = false$ and are rejected, while 27 have $Sprinkler = true$; of the 27, 8 have $Rain = true$ and 19 have $Rain = false$. Hence,

$$\mathbf{P}(Rain|Sprinkler = true) \approx \text{NORMALIZE}(\langle 8, 19\rangle) = \langle 0.296, 0.704\rangle .$$

The true answer is $\langle 0.3, 0.7\rangle$. As more samples are collected, the estimate will converge to the true answer. The standard deviation of the error in each probability will be proportional to $1/\sqrt{n}$, where n is the number of samples used in the estimate.

The biggest problem with rejection sampling is that it rejects so many samples! The fraction of samples consistent with the evidence $\mathbf{e}$ drops exponentially as the number of evidence variables grows, so the procedure is simply unusable for complex problems.

Notice that rejection sampling is very similar to the estimation of conditional probabilities directly from the real world. For example, to estimate $\mathbf{P}(Rain|RedSkyAtNight = true)$, one can simply count how often it rains after a red sky is observed the previous evening—ignoring those evenings when the sky is not red. (Here, the world itself plays the role of the

sample generation algorithm.) Obviously, this could take a long time if the sky is very seldom red, and that is the weakness of rejection sampling.

Likelihood weighting

LIKELIHOOD WEIGHTING

Likelihood weighting avoids the inefficiency of rejection sampling by generating only events that are consistent with the evidence **e**. We begin by describing how the algorithm works; then we show that it works correctly—that is, generates consistent probability estimates.

LIKELIHOOD-WEIGHTING (see Figure 14.14) fixes the values for the evidence variables **E** and samples only the remaining variables X and **Y**. This guarantees that each event generated is consistent with the evidence. Not all events are equal, however. Before tallying the counts in the distribution for the query variable, each event is weighted by the *likelihood* that the event accords to the evidence, as measured by the product of the conditional probabilities for each evidence variable, given its parents. Intuitively, events in which the actual evidence appears unlikely should be given less weight.

Let us apply the algorithm to the network shown in Figure 14.11(a), with the query $\mathbf{P}(Rain|Sprinkler = true, WetGrass = true)$. The process goes as follows: First, the weight w is set to 1.0. Then an event is generated:

1. Sample from $\mathbf{P}(Cloudy) = \langle 0.5, 0.5\rangle$; suppose this returns $true$.
2. $Sprinkler$ is an evidence variable with value $true$. Therefore, we set
$$w \leftarrow w \times P(Sprinkler = true|Cloudy = true) = 0.1 .$$
3. Sample from $\mathbf{P}(Rain|Cloudy = true) = \langle 0.8, 0.2\rangle$; suppose this returns $true$.
4. $WetGrass$ is an evidence variable with value $true$. Therefore, we set
$$w \leftarrow w \times P(WetGrass = true|Sprinkler = true, Rain = true) = 0.099 .$$

Here WEIGHTED-SAMPLE returns the event $[true, true, true, true]$ with weight 0.099, and this is tallied under $Rain = true$. The weight is low because the event describes a cloudy day, which makes the sprinkler unlikely to be on.

To understand why likelihood weighting works, we start by examining the sampling distribution S_{WS} for WEIGHTED-SAMPLE. Remember that the evidence variables **E** are fixed with values **e**. We will call the other variables **Z**, that is, $\mathbf{Z} = \{X\} \cup \mathbf{Y}$. The algorithm samples each variable in **Z** given its parent values:

$$S_{WS}(\mathbf{z}, \mathbf{e}) = \prod_{i=1}^{l} P(z_i|parents(Z_i)) . \tag{14.6}$$

Notice that $Parents(Z_i)$ can include both hidden variables and evidence variables. Unlike the prior distribution $P(\mathbf{z})$, the distribution S_{WS} pays some attention to the evidence: the sampled values for each Z_i will be influenced by evidence among Z_i's ancestors. On the other hand, S_{WS} pays less attention to the evidence than does the true posterior distribution $P(\mathbf{z}|\mathbf{e})$, because the sampled values for each Z_i *ignore* evidence among Z_i's non-ancestors.[6]

[6] Ideally, we would like use a sampling distribution equal to the true posterior $P(\mathbf{z}|\mathbf{e})$, to take all the evidence into account. This cannot be done efficiently, however. If it could, then we could approximate the desired probability to arbitrary accuracy with a polynomial number of samples. It can be shown that no such polynomial-time approximation scheme can exist.

function LIKELIHOOD-WEIGHTING($X, \mathbf{e}, bn, N$) **returns** an estimate of $P(X|\mathbf{e})$
  inputs: X, the query variable
    $\mathbf{e}$, evidence specified as an event
    bn, a Bayesian network
    N, the total number of samples to be generated
  local variables: $\mathbf{W}$, a vector of weighted counts over X, initially zero

  for j = 1 to N **do**
    $\mathbf{x}, w \leftarrow$ WEIGHTED-SAMPLE(bn)
    $\mathbf{W}[x] \leftarrow \mathbf{W}[x] + w$ where x is the value of X in $\mathbf{x}$
  return NORMALIZE($\mathbf{W}[X]$)

---

function WEIGHTED-SAMPLE($bn, \mathbf{e}$) **returns** an event and a weight

  $\mathbf{x} \leftarrow$ an event with n elements; $w \leftarrow 1$
  for i = 1 **to** n **do**
    if X_i has a value x_i in $\mathbf{e}$
      then $w \leftarrow w \times P(X_i = x_i \mid parents(X_i))$
      else $x_i \leftarrow$ a random sample from $\mathbf{P}(X_i \mid parents(X_i))$
  return $\mathbf{x}$, w

Figure 14.14 The likelihood weighting algorithm for inference in Bayesian networks.

The likelihood weight w makes up for the difference between the actual and desired sampling distributions. The weight for a given sample $\mathbf{x}$, composed from $\mathbf{z}$ and $\mathbf{e}$, is the product of the likelihoods for each evidence variable given its parents (some or all of which may be among the Z_is):

$$w(\mathbf{z}, \mathbf{e}) = \prod_{i=1}^{m} P(e_i|parents(E_i)) \ . \tag{14.7}$$

Multiplying Equations (14.6) and (14.7), we see that the *weighted* probability of a sample has the particularly convenient form

$$\begin{aligned} S_{WS}(\mathbf{z}, \mathbf{e})w(\mathbf{z}, \mathbf{e}) &= \prod_{i=1}^{l} P(z_i|parents(Z_i)) \prod_{i=1}^{m} P(e_i|parents(E_i)) \\ &= P(\mathbf{z}, \mathbf{e}) \ , \end{aligned} \tag{14.8}$$

because the two products cover all the variables in the network, allowing us to use Equation (14.1) for the joint probability.

Now it is easy to show that likelihood weighting estimates are consistent. For any particular value x of X, the estimated posterior probability can be calculated as follows:

$$\begin{aligned} \hat{P}(x|\mathbf{e}) &= \alpha \sum_{\mathbf{y}} N_{WS}(x, \mathbf{y}, \mathbf{e})w(x, \mathbf{y}, \mathbf{e}) && \text{from LIKELIHOOD-WEIGHTING} \\ &\approx \alpha' \sum_{\mathbf{y}} S_{WS}(x, \mathbf{y}, \mathbf{e})w(x, \mathbf{y}, \mathbf{e}) && \text{for large } N \end{aligned}$$

$$= \alpha' \sum_{\mathbf{y}} P(x, \mathbf{y}, \mathbf{e}) \qquad \text{by Equation (14.8)}$$
$$= \alpha' P(x, \mathbf{e}) = P(x|\mathbf{e}) \, .$$

Hence, likelihood weighting returns consistent estimates.

Because likelihood weighting uses all the samples generated, it can be much more efficient than rejection sampling. It will, however, suffer a degradation in performance as the number of evidence variables increases. This is because most samples will have very low weights and hence the weighted estimate will be dominated by the tiny fraction of samples that accord more than an infinitesimal likelihood to the evidence. The problem is exacerbated if the evidence variables occur late in the variable ordering, because then the samples will be simulations that bear little resemblance to the reality suggested by the evidence.

Inference by Markov chain simulation

MARKOV CHAIN MONTE CARLO

In this section, we describe the **Markov chain Monte Carlo** (MCMC) algorithm for inference in Bayesian networks. We will first describe what the algorithm does, then we will explain why it works and why it has such a complicated name.

The MCMC algorithm

Unlike the other two sampling algorithms, which generate each event from scratch, MCMC generates each event by making a random change to the preceding event. It is therefore helpful to think of the network as being in a particular *current state* specifying a value for every variable. The next state is generated by randomly sampling a value for one of the nonevidence variables X_i, *conditioned on the current values of the variables in the Markov blanket of* X_i. (Recall from page 499 that the Markov blanket of a variable consists of its parents, children, and children's parents.) MCMC therefore wanders randomly around the state space—the space of possible complete assignments—flipping one variable at a time, but keeping the evidence variables fixed.

Consider the query $\mathbf{P}(Rain|Sprinkler = true, WetGrass = true)$ applied to the network in Figure 14.11(a). The evidence variables $Sprinkler$ and $WetGrass$ are fixed to their observed values and the hidden variables $Cloudy$ and $Rain$ are initialized randomly—let us say to $true$ and $false$ respectively. Thus, the initial state is $[true, true, false, true]$. Now the following steps are executed repeatedly:

1. $Cloudy$ is sampled, given the current values of its Markov blanket variables: in this case, we sample from $\mathbf{P}(Cloudy|Sprinkler = true, Rain = false)$. (Shortly, we will show how to calculate this distribution.) Suppose the result is $Cloudy = false$. Then the new current state is $[false, true, false, true]$.
2. $Rain$ is sampled, given the current values of its Markov blanket variables: in this case, we sample from $\mathbf{P}(Rain|Cloudy = false, Sprinkler = true, WetGrass = true)$. Suppose this yields $Rain = true$. The new current state is $[false, true, true, true]$.

Each state visited during this process is a sample that contributes to the estimate for the query variable $Rain$. If the process visits 20 states where $Rain$ is true and 60 states where $Rain$ is

```
function MCMC-ASK(X, e, bn, N) returns an estimate of P(X|e)
  local variables: N[X], a vector of counts over X, initially zero
                   Z, the nonevidence variables in bn
                   x, the current state of the network, initially copied from e

  initialize x with random values for the variables in Z
  for j = 1 to N do
    for each Z_i in Z do
      sample the value of Z_i in x from P(Z_i|mb(Z_i)) given the values of MB(Z_i) in x
      N[x] ← N[x] + 1 where x is the value of X in x
  return NORMALIZE(N[X])
```

Figure 14.15 The MCMC algorithm for approximate inference in Bayesian networks.

false, then the answer to the query is NORMALIZE($\langle 20, 60\rangle$) = $\langle 0.25, 0.75\rangle$. The complete algorithm is shown in Figure 14.15.

Why MCMC works

We will now show that MCMC returns consistent estimates for posterior probabilities. The material in this section is quite technical, but the basic claim is straightforward: *the sampling process settles into a "dynamic equilibrium" in which the long-run fraction of time spent in each state is exactly proportional to its posterior probability.* This remarkable property follows from the specific **transition probability** with which the process moves from one state to another, as defined by the conditional distribution given the Markov blanket of the variable being sampled.

TRANSITION PROBABILITY

Let $q(\mathbf{x} \rightarrow \mathbf{x}')$ be the probability that the process makes a transition from state $\mathbf{x}$ to state $\mathbf{x}'$. This transition probability defines what is called a **Markov chain** on the state space. (Markov chains will also figure prominently in Chapters 15 and 17.) Now suppose that we run the Markov chain for t steps, and let $\pi_t(\mathbf{x})$ be the probability that the system is in state $\mathbf{x}$ at time t. Similarly, let $\pi_{t+1}(\mathbf{x}')$ be the probability of being in state $\mathbf{x}'$ at time $t+1$. Given $\pi_t(\mathbf{x})$, we can calculate $\pi_{t+1}(\mathbf{x}')$ by summing, for all states the system could be in at time t, the probability of being in that state times the probability of making the transition to $\mathbf{x}'$:

MARKOV CHAIN

$$\pi_{t+1}(\mathbf{x}') = \sum_{\mathbf{x}} \pi_t(\mathbf{x}) q(\mathbf{x} \rightarrow \mathbf{x}') \ .$$

STATIONARY DISTRIBUTION

We will say that the chain has reached its **stationary distribution** if $\pi_t = \pi_{t+1}$. Let us call this stationary distribution π; its defining equation is therefore

$$\pi(\mathbf{x}') = \sum_{\mathbf{x}} \pi(\mathbf{x}) q(\mathbf{x} \rightarrow \mathbf{x}') \qquad \text{for all } \mathbf{x}' \ . \tag{14.9}$$

Under certain standard assumptions about the transition probability distribution q,[7] there is exactly one distribution π satisfying this equation for any given q.

[7] The Markov chain defined by q must be **ergodic**—that is, essentially, every state must be reachable from every other, and there can be no strictly periodic cycles.

Equation (14.9) can be read as saying that the expected "outflow" from each state (i.e., its current "population") is equal to the expected "inflow" from all the states. One obvious way to satisfy this relationship is if the expected flow between any pair of states is the same in both directions. This is the property of **detailed balance**:

DETAILED BALANCE

$$\pi(\mathbf{x})q(\mathbf{x} \rightarrow \mathbf{x}') = \pi(\mathbf{x}')q(\mathbf{x}' \rightarrow \mathbf{x}) \qquad \text{for all } \mathbf{x},\ \mathbf{x}' . \tag{14.10}$$

We can show that detailed balance implies stationarity simply by summing over $\mathbf{x}$ in Equation (14.10). We have

$$\sum_{\mathbf{x}} \pi(\mathbf{x})q(\mathbf{x} \rightarrow \mathbf{x}') = \sum_{\mathbf{x}} \pi(\mathbf{x}')q(\mathbf{x}' \rightarrow \mathbf{x}) = \pi(\mathbf{x}') \sum_{\mathbf{x}} q(\mathbf{x}' \rightarrow \mathbf{x}) = \pi(\mathbf{x}')$$

where the last step follows because a transition from $\mathbf{x}'$ is guaranteed to occur.

Now we will show that the transition probability $q(\mathbf{x} \rightarrow \mathbf{x}')$ defined by the sampling step in MCMC-ASK satisfies the detailed balance equation with a stationary distribution equal to $P(\mathbf{x}|\mathbf{e})$, (the true posterior distribution on the hidden variables). We will do this in two steps. First, we will define a Markov chain in which each variable is sampled conditionally on the current values of *all* the other variables, and we will show that this satisfies detailed balance. Then, we will simply observe that, for Bayesian networks, doing that is equivalent to sampling conditionally on the variable's Markov blanket (see page 499).

Let X_i be the variable to be sampled, and let $\overline{\mathbf{X}_i}$ be all the hidden variables *other than* X_i. Their values in the current state are x_i and $\overline{\mathbf{x}_i}$. If we sample a new value x_i' for X_i conditionally on all the other variables, including the evidence, we have

$$q(\mathbf{x} \rightarrow \mathbf{x}') = q((x_i, \overline{\mathbf{x}_i}) \rightarrow (x_i', \overline{\mathbf{x}_i})) = P(x_i'|\overline{\mathbf{x}_i}, \mathbf{e}) .$$

GIBBS SAMPLER

This transition probability is called the **Gibbs sampler** and is a particularly convenient form of MCMC. Now we show that the Gibbs sampler is in detailed balance with the true posterior:

$$\begin{aligned}
&\pi(\mathbf{x})q(\mathbf{x} \rightarrow \mathbf{x}') = P(\mathbf{x}|\mathbf{e})P(x_i'|\overline{\mathbf{x}_i}, \mathbf{e}) = P(x_i, \overline{\mathbf{x}_i}|\mathbf{e})P(x_i'|\overline{\mathbf{x}_i}, \mathbf{e}) \\
&= P(x_i|\overline{\mathbf{x}_i}, \mathbf{e})P(\overline{\mathbf{x}_i}|\mathbf{e})P(x_i'|\overline{\mathbf{x}_i}, \mathbf{e}) \qquad \text{(using the chain rule on the first term)} \\
&= P(x_i|\overline{\mathbf{x}_i}, \mathbf{e})P(x_i', \overline{\mathbf{x}_i}|\mathbf{e}) \qquad \text{(using the chain rule backwards)} \\
&= \pi(\mathbf{x}')q(\mathbf{x}' \rightarrow \mathbf{x}) .
\end{aligned}$$

As stated on page 499, a variable is independent of all other variables given its Markov blanket; hence,

$$P(x_i'|\overline{\mathbf{x}_i}, \mathbf{e}) = P(x_i'|mb(X_i))$$

where $mb(X_i)$ denotes the values of the variables in X_i's Markov blanket, $MB(X_i)$. As shown in Exercise 14.10, the probability of a variable given its Markov blanket is proportional to the probability of the variable given its parents times the probability of each child given its respective parents:

$$P(x_i'|mb(X_i)) = \alpha\, P(x_i'|parents(X_i)) \times \prod_{Y_j \in Children(X_i)} P(y_j|parents(Y_j)) . \tag{14.11}$$

Hence, to flip each variable X_i, the number of multiplications required is equal to the number of X_i's children.

We have discussed here only one simple variant of MCMC, namely the Gibbs sampler. In its most general form, MCMC is a powerful method for computing with probability models and many variants have been developed, including the simulated annealing algorithm presented in Chapter 4, the stochastic satisfiability algorithms in Chapter 7, and the Metropolis–Hastings sampler in Chapter 15.

14.6 Extending Probability to First-Order Representations

In Chapter 8, we explained the representational advantages possessed by first-order logic in comparison to propositional logic. First-order logic commits to the existence of objects and relations among them and can express facts about *some* or *all* of the objects in a domain. This often results in representations that are vastly more concise than the equivalent propositional descriptions. Now, Bayesian networks are essentially propositional: the set of variables is fixed and finite, and each has a fixed domain of possible values. This fact limits the applicability of Bayesian networks. *If we can find a way to combine probability theory with the expressive power of first-order representations, we expect to be able to increase dramatically the range of problems that can be handled.*

The basic insight required to achieve this goal is the following: In the propositional context, a Bayesian network specifies probabilities over atomic events, each of which specifies a value for each variable in the network. Thus, an atomic event is a **model** or **possible world**, in the terminology of propositional logic. In the first-order context, a model (with its interpretation) specifies a domain of objects, the relations that hold among those objects, and a mapping from the constants and predicates of the knowledge base to the objects and relations in the model. Therefore, *a first-order probabilistic knowledge base should specify probabilities for all possible first-order models.* Let $\mu(M)$ be the probability assigned to model M by the knowledge base. For any first-order sentence ϕ, the probability $P(\phi)$ is given in the usual way by summing over the possible worlds where ϕ is true:

$$P(\phi) = \sum_{M:\phi \text{ is true in } M} \mu(M) \; . \tag{14.12}$$

So far, so good. There is, however, a problem: the set of first-order models is infinite. This means that (1) the summation could be infeasible, and (2) specifying a complete, consistent distribution over an infinite set of worlds could be very difficult.

Let us scale back our ambition, at least temporarily. In particular, let us devise a restricted language for which there are only finitely many models of interest. There are several ways to do this. Here, we present **relational probability models**, or RPMs, which borrow ideas from semantic networks (Chapter 10) and from object-relational databases. Other approaches are discussed in the bibliographical and historical notes.

RELATIONAL PROBABILITY MODEL

RPMs allow constant symbols that name objects. For example, let *ProfSmith* be the name of a professor, and let *Jones* be the name of a student. Each object is an instance of a class; for example, *ProfSmith* is a *Professor* and *Jones* is a *Student*. We assume that the class of every constant symbol is known.

SIMPLE FUNCTION

Our function symbols will be divided into two kinds. The first kind, **simple functions**, maps an object not to another structured object, but to a value from a fixed domain of values, just like a random variable. For example, $Intelligence(Jones)$ and $Funding(ProfSmith)$ might be hi or lo; $Success(Jones)$ and $Fame(ProfSmith)$ may be $true$ or $false$. Function symbols must not be applied to values such as $true$ and $false$, so it is not possible to have nesting of simple functions. In this way, we avoid one source of infinities. The value of a simple function applied to a given object may be observed or unknown; these will be the basic random variables of our representation.[8]

COMPLEX FUNCTION

We also allow **complex functions**, which map objects to other objects. For example, $Advisor(Jones)$ may be $ProfSmith$. Each complex function has a specified domain and range, which are classes. For example, the domain of $Advisor$ is $Student$ and the range is $Professor$. Functions apply only to objects of the right class; for instance, the $Advisor$ of $ProfSmith$ is undefined. Complex functions may be nested: $DeptHead(Advisor(Jones))$ could be $ProfMoore$. We will assume (for now) that the values of all complex functions are known for all constant symbols. Because the KB is finite, this implies that every chain of complex function applications leads to one of a finite number of objects.[9]

The last element we need is the probabilistic information. For each simple function, we specify a set of parents, just as in Bayesian networks. The parents can be other simple functions of the same object; for example, the $Funding$ of a $Professor$ might depend on his or her $Fame$. The parents can also be simple functions of *related* objects—for example, the $Success$ of a student could depend on the $Intelligence$ of the student and the $Fame$ of the student's advisor. These are really *universally quantified* assertions about the parents of all the objects in a class. Thus, we could write

$$\forall x \;\; x \in Student \;\Rightarrow$$
$$Parents(Success(x)) = \{Intelligence(x), Fame(Advisor(x))\} .$$

(Less formally, we can draw diagrams like Figure 14.16(a).) Now we specify the conditional probability distribution for the child, given its parents. For example, we might say that

$$\forall x \;\; x \in Student \;\Rightarrow$$
$$P(Success(x) = true | Intelligence(x) = hi, Fame(Advisor(x)) = true) = 0.95 .$$

Just as in semantic networks, we can attach the conditional distribution to the class itself, so that the instances **inherit** the dependencies and conditional probabilities from the class.

The semantics for the RPM language assumes that every constant symbol refers to a distinct object—the **unique names assumption** described in Chapter 10. Given this assumption and the restrictions listed previously, it can be shown that every RPM generates a fixed, finite set of random variables, each of which is a simple function applied to a constant symbol. Then, provided that the parent–child dependencies are *acyclic*, we can construct an equivalent Bayesian network. That is, the RPM and the Bayesian network specify identical probabili-

[8] They play a role very similar to that of the ground atomic sentences generated in the propositionalization process described in Section 9.1.

[9] This restriction means that we cannot use complex functions such as $Father$ and $Mother$, which lead to potentially infinite chains that would have to end with an unknown object. We revisit this restriction later.

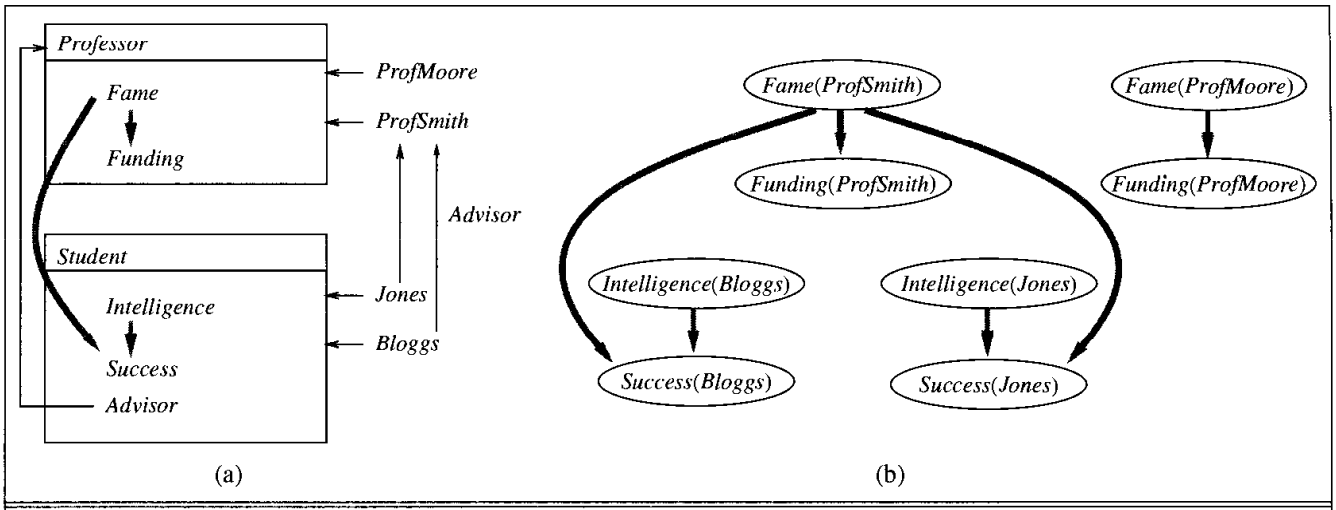


Figure 14.16 (a) An RPM describing two classes: *Professor* and *Student*. There are two professors and two students, and *ProfSmith* is the advisor of both students. (b) The Bayesian network equivalent to the RPM in (a).

ties for each possible world. Figure 14.16(b) shows the Bayesian network corresponding to the RPM in Figure 14.16(a). Notice that the *Advisor* links in the RPM are absent in the Bayesian network. This is because they are fixed and known. They appear *implicitly* in the network topology, however; for example, $Success(Jones)$ has $Fame(ProfSmith)$ as a parent because $Advisor(Jones)$ is $ProfSmith$. In general, the relations that hold among the objects determine the pattern of dependencies among the properties of those objects.

RECURSIVE DEPENDENCY

There are several ways to increase the expressive power of RPMs. We can allow **recursive dependencies** among variables to capture certain kinds of recurring relationships. For example, suppose that addiction to fast food is caused by the $McGene$. Then, for any x, $McGene(x)$ depends on $McGene(Father(x))$ and $McGene(Mother(x))$, which depend in turn on $McGene(Father(Father(x)))$, $McGene(Mother(Father(x)))$, and so on. Even though such knowledge bases correspond to Bayesian networks with infinitely many random variables, solutions can sometimes be obtained from fixed-point equations. For example, the equilibrium distribution of the $McGene$ can be calculated, given the conditional probability of inheritance. Another very important family of recursive knowledge bases consists of the **temporal probability models** described in Chapter 15. In these models, properties of the state at time t depend on properties of the state at time $t - 1$, and so on.

RELATIONAL UNCERTAINTY

RPMs can also be extended to allow for **relational uncertainty**—that is, uncertainty about the values of complex functions. For example, we may not know who $Advisor(Jones)$ is. $Advisor(Jones)$ then becomes a random variable, with possible values $ProfSmith$ and $ProfMoore$. The corresponding network is shown in Figure 14.17.

IDENTITY UNCERTAINTY

There can also be **identity uncertainty**; for example, we might not know whether $Mary$ and $ProfSmith$ are the same person. With identity uncertainty, the number of objects and propositions can vary across possible worlds. A world where $Mary$ and $ProfSmith$ are the same person has one fewer object than a world in which they are different people. This makes the inference process more complicated, but the basic principle established in Equation (14.12) still holds: the probability of any sentence is well defined and can be calculated.

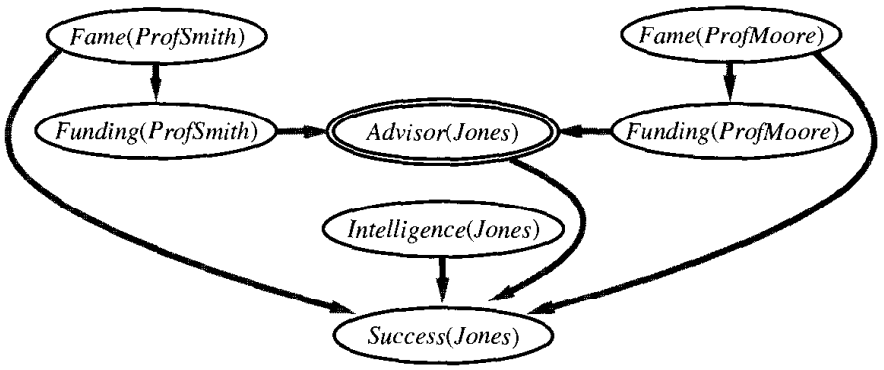


Figure 14.17 Part of the Bayesian network corresponding to an RPM in which $Advisor(Jones)$ is unknown, but is either $ProfSmith$ or $ProfMoore$. The choice of advisor depends on how much funding each professor has. Notice that $Success(Jones)$ will now depend on the $Fame$ of *both* professors, although the value of $Advisor(Jones)$ determines which one actually has an influence.

Identity uncertainty is particularly important for robots and for embedded sensor systems that must keep track of multiple objects. We return to this problem in Chapter 15.

Let us now examine the question of inference. Clearly, inference can be done in the equivalent Bayesian network, provided that we restrict the RPM language so that the equivalent network is finite and has a fixed structure. This is analogous to the way in which first-order logical inference can be done via propositional inference on the equivalent propositional knowledge base. (See Section 9.1.) As in the logical case, the equivalent network could be too large to construct, let alone evaluate. Dense interconnections are also a problem. (See Exercise 14.12.) Approximation algorithms. such as MCMC (Section 14.5), are therefore very useful for RPM inference.

When MCMC is applied to the equivalent Bayesian network for a simple RPM knowledge base with no relational or identity uncertainty, the algorithm samples from the space of possible worlds defined by the values of simple functions of the objects. It is easy to see that this approach can be extended to handle relational and identity uncertainty as well. In that case, a transition between possible worlds might change the value of a simple function or it might change a complex function, and so lead to a change in the dependency structure. Transitions might also change the identity relations among the constant symbols. Thus, MCMC seems to be an elegant way to handle inference for quite expressive first-order probabilistic knowledge bases.

Research in this area is still at an early stage, but already it is becoming clear that first-order probabilistic reasoning yields a tremendous increase in the effectiveness of AI systems at handling uncertain information. Potential applications include computer vision, natural language understanding, information retrieval, and situation assessment. In all of these areas, the set of objects—and hence the set of random variables—is not known in advance, so purely "propositional" methods, such as Bayesian networks, are incapable of representing the situation completely. They have been augmented by search over the space of model, but RPMs allow reasoning about this uncertainty in a single model.

14.7 Other Approaches to Uncertain Reasoning

Other sciences (e.g., physics, genetics, and economics) have long favored probability as a model for uncertainty. In 1819, Pierre Laplace said "Probability theory is nothing but common sense reduced to calculation." In 1850, James Maxwell said "the true logic for this world is the calculus of Probabilities, which takes account of the magnitude of the probability which is, or ought to be, in a reasonable man's mind."

Given this long tradition, it is perhaps surprising that AI has considered many alternatives to probability. The earliest expert systems of the 1970s ignored uncertainty and used strict logical reasoning, but it soon became clear that this was impractical for most real-world domains. The next generation of expert systems (especially in medical domains) used probabilistic techniques. Initial results were promising, but they did not scale up because of the exponential number of probabilities required in the full joint distribution. (Efficient Bayesian network algorithms were unknown then.) As a result, probabilistic approaches fell out of favor from roughly 1975 to 1988, and a variety of alternatives to probability were tried for a variety of reasons:

- One common view is that probability theory is essentially numerical, whereas human judgmental reasoning is more "qualitative." Certainly, we are not consciously aware of doing numerical calculations of degrees of belief. (Neither are we aware of doing unification, yet we seem to be capable of some kind of logical reasoning.) It might be that we have some kind of numerical degrees of belief encoded directly in strengths of connections and activations in our neurons. In that case, the difficulty of conscious access to those strengths is not surprising. One should also note that qualitative reasoning mechanisms can be built directly on top of probability theory, so that the "no numbers" argument against probability has little force. Nonetheless, some qualitative schemes have a good deal of appeal in their own right. One of the best studied is **default reasoning**, which treats conclusions not as "believed to a certain degree," but as "believed until a better reason is found to believe something else." Default reasoning is covered in Chapter 10.
- **Rule-based** approaches to uncertainty also have been tried. Such approaches hope to build on the success of logical rule-based systems, but add a sort of "fudge factor" to each rule to accommodate uncertainty. These methods were developed in the mid-1970s and formed the basis for a large number of expert systems in medicine and other areas.
- One area that we have not addressed so far is the question of **ignorance**, as opposed to uncertainty. Consider the flipping of a coin. If we know that the coin is fair, then a probability of 0.5 for heads is reasonable. If we know that the coin is biased, but we do not know which way, then 0.5 is the only reasonable probability. Obviously, the two cases are different, yet probability seems not to distinguish them. The **Dempster–Shafer theory** uses **interval-valued** degrees of belief to represent an agent's knowledge of the probability of a proposition. Other methods using second-order probabilities are also discussed.

- Probability makes the same ontological commitment as logic: that events are true or false in the world, even if the agent is uncertain as to which is the case. Researchers in **fuzzy logic** have proposed an ontology that allows **vagueness**: that an event can be "sort of" true. Vagueness and uncertainty are in fact orthogonal issues, as we will see.

The next three subsections treat some of these approaches in slightly more depth. We will not provide detailed technical material, but we cite references for further study.

Rule-based methods for uncertain reasoning

Rule-based systems emerged from early work on practical and intuitive systems for logical inference. Logical systems in general, and logical rule-based systems in particular, have three desirable properties:

LOCALITY

◇ **Locality**: In logical systems, whenever we have a rule of the form $A \Rightarrow B$, we can conclude B, given evidence A, *without worrying about any other rules.* In probabilistic systems, we need to consider all the evidence in the Markov blanket.

DETACHMENT

◇ **Detachment**: Once a logical proof is found for a proposition B, the proposition can be used regardless of how it was derived. That is, it can be **detached** from its justification. In dealing with probabilities, on the other hand, the source of the evidence for a belief is important for subsequent reasoning.

TRUTH-FUNCTIONALITY

◇ **Truth-functionality**: In logic, the truth of complex sentences can be computed from the truth of the components. Probability combination does not work this way, except under strong global independence assumptions.

There have been several attempts to devise uncertain reasoning schemes that retain these advantages. The idea is to attach degrees of belief to propositions and rules and to devise purely local schemes for combining and propagating those degrees of belief. The schemes are also truth-functional; for example, the degree of belief in $A \vee B$ is a function of the belief in A and the belief in B.

The bad news for rule-based systems is that the properties of *locality, detachment, and truth-functionality are simply not appropriate for uncertain reasoning.* Let us look at truth-functionality first. Let H_1 be the event that a fair coin flip comes up heads, let T_1 be the event that the coin comes up tails on that same flip, and let H_2 be the event that the coin comes up heads on a second flip. Clearly, all three events have the same probability, 0.5, and so a truth-functional system must assign the same belief to the disjunction of any two of them. But we can see that the probability of the disjunction depends on the events themselves and not just on their probabilities:

$P(A)$	$P(B)$	$P(A \vee B)$
$P(H_1) = 0.5$	$P(H_1) = 0.5$	$P(H_1 \vee H_1) = 0.50$
	$P(T_1) = 0.5$	$P(H_1 \vee T_1) = 1.00$
	$P(H_2) = 0.5$	$P(H_1 \vee H_2) = 0.75$

It gets worse when we chain evidence together. Truth-functional systems have **rules** of the form $A \mapsto B$ that allow us to compute the belief in B as a function of the belief in the rule

and the belief in A. Both forward- and backward-chaining systems can be devised. The belief in the rule is assumed to be constant and is usually specified by the knowledge engineer—for example, as $A \mapsto_{0.9} B$.

Consider the wet-grass situation from Figure 14.11(a). If we wanted to be able to do both causal and diagnostic reasoning, we would need the two rules

$$Rain \mapsto WetGrass \qquad \text{and} \qquad WetGrass \mapsto Rain\ .$$

These two rules form a feedback loop: evidence for *Rain* increases the belief in *WetGrass*, which in turn increases the belief in *Rain* even more. Clearly, uncertain reasoning systems need to keep track of the paths along which evidence is propagated.

Intercausal reasoning (or explaining away) is also tricky. Consider what happens when we have the two rules

$$Sprinkler \mapsto WetGrass \qquad \text{and} \qquad WetGrass \mapsto Rain\ .$$

Suppose we see that the sprinkler is on. Chaining forward through our rules, this increases the belief that the grass will be wet, which in turn increases the belief that it is raining. But this is ridiculous: the fact that the sprinkler is on explains away the wet grass and should *reduce* the belief in rain. A truth-functional system acts as if it also believes $Sprinkler \mapsto Rain$.

Given these difficulties, how is it possible that truth-functional systems were ever considered useful? The answer lies in restricting the task and in carefully engineering the rule base so that undesirable interactions do not occur. The most famous example of a truth-functional system for uncertain reasoning is the **certainty factors** model, which was developed for the MYCIN medical diagnosis program and was widely used in expert systems of the late 1970s and 1980s. Almost all uses of certainty factors involved rule sets that were either purely diagnostic (as in MYCIN) or purely causal. Furthermore, evidence was entered only at the "roots" of the rule set, and most rule sets were singly connected. Heckerman (1986) has shown that, under these circumstances, a minor variation on certainty-factor inference was exactly equivalent to Bayesian inference on polytrees. In other circumstances, certainty factors could yield disastrously incorrect degrees of belief through overcounting of evidence. As rule sets became larger, undesirable interactions between rules became more common, and practitioners found that the certainty factors of many other rules had to be "tweaked" when new rules were added. Needless to say, the approach is no longer recommended.

CERTAINTY FACTORS

Representing ignorance: Dempster–Shafer theory

DEMPSTER–SHAFER

The **Dempster–Shafer** theory is designed to deal with the distinction between **uncertainty** and **ignorance**. Rather than computing the probability of a proposition, it computes the probability that the evidence supports the proposition. This measure of belief is called a **belief function**, written $Bel(X)$.

BELIEF FUNCTION

We return to coin flipping for an example of belief functions. Suppose a shady character comes up to you and offers to bet you \$10 that his coin will come up heads on the next flip. Given that the coin might or might not be fair, what belief should you ascribe to the event that it comes up heads? Dempster–Shafer theory says that because you have no evidence either way, you have to say that the belief $Bel(Heads) = 0$ and also that $Bel(\neg Heads) = 0$.

This makes Dempster–Shafer reasoning systems skeptical in a way that has some intuitive appeal. Now suppose you have an expert at your disposal who testifies with 90% certainty that the coin is fair (i.e., he is 90% sure that $P(Heads) = 0.5$). Then Dempster–Shafer theory gives $Bel(Heads) = 0.9 \times 0.5 = 0.45$ and likewise $Bel(\neg Heads) = 0.45$. There is still a 10 percentage point "gap" that is not accounted for by the evidence. "Dempster's rule" (Dempster, 1968) shows how to combine evidence to give new values for Bel, and Shafer's work extends this into a complete computational model.

As with default reasoning, there is a problem in connecting beliefs to actions. With probabilities, decision theory says that if $P(Heads) = P(\neg Heads) = 0.5$, then (assuming that winning \$10 and losing \$10 are considered equal magnitude opposites) the reasoner will be indifferent between the action of accepting and declining the bet. A Dempster–Shafer reasoner has $Bel(\neg Heads) = 0$ and thus no reason to accept the bet, but then it also has $Bel(Heads) = 0$ and thus no reason to decline it. Thus, it seems that the Dempster–Shafer reasoner comes to the same conclusion about how to act in this case. Unfortunately, Dempster–Shafer theory allows no definite decision in many other cases where probabilistic inference does yield a specific choice. In fact, the notion of utility in the Dempster–Shafer model is not yet well understood.

One interpretation of Dempster—Shafer theory is that it defines a probability interval: the interval for *Heads* is $[0, 1]$ before our expert testimony and $[0.45, 0.55]$ after. The width of the interval might be an aid in deciding when we need to acquire more evidence: it can tell you that the expert's testimony will help you if you do not know whether the coin is fair, but will not help you if you have already learned that the coin is fair. However, there are no clear guidelines for how to do this, because there is no clear meaning for what the width of an interval means. In the Bayesian approach, this kind of reasoning can be done easily by examining how much one's belief would change if one were to acquire more evidence. For example, knowing whether the coin is fair would have a significant impact on the belief that it will come up heads, and detecting an asymmetric weight would have an impact on the belief that the coin is fair. A complete Bayesian model would include probability estimates for factors such as these, allowing us to express our "ignorance" in terms of how our beliefs would change in the face of future information gathering.

Representing vagueness: Fuzzy sets and fuzzy logic

FUZZY SET THEORY

Fuzzy set theory is a means of specifying how well an object satisfies a vague description. For example, consider the proposition "Nate is tall." Is this true, if Nate is $5'\ 10''$? Most people would hesitate to answer "true" or "false," preferring to say, "sort of." Note that this is not a question of uncertainty about the external world—we are sure of Nate's height. The issue is that the linguistic term "tall" does not refer to a sharp demarcation of objects into two classes—there are *degrees* of tallness. For this reason, *fuzzy set theory is not a method for uncertain reasoning at all.* Rather, fuzzy set theory treats *Tall* as a fuzzy predicate and says that the truth value of $Tall(Nate)$ is a number between 0 and 1, rather than being just *true* or *false*. The name "fuzzy set" derives from the interpretation of the predicate as implicitly defining a set of its members—a set that does not have sharp boundaries.

FUZZY LOGIC

Fuzzy logic is a method for reasoning with logical expressions describing membership in fuzzy sets. For example, the complex sentence $Tall(Nate) \wedge Heavy(Nate)$ has a fuzzy truth value that is a function of the truth values of its components. The standard rules for evaluating the fuzzy truth, T, of a complex sentence are

$$T(A \wedge B) = \min(T(A), T(B))$$
$$T(A \vee B) = \max(T(A), T(B))$$
$$T(\neg A) = 1 - T(A) .$$

Fuzzy logic is therefore a truth-functional system—a fact that causes serious difficulties. For example, suppose that $T(Tall(Nate)) = 0.6$ and $T(Heavy(Nate)) = 0.4$. Then we have $T(Tall(Nate) \wedge T(Heavy(Nate)) = 0.4$, which seems reasonable, but we also get the result $T(Tall(Nate) \wedge \neg Tall(Nate)) = 0.4$, which does not. Clearly, the problem arises from the inability of a truth-functional approach to take into account the correlations or anticorrelations among the component propositions.

FUZZY CONTROL

Fuzzy control is a methodology for constructing control systems in which the mapping between real-valued input and output parameters is represented by fuzzy rules. Fuzzy control has been very successful in commercial products such as automatic transmissions, video cameras, and electric shavers. Critics (see, e.g., Elkan, 1993) argue that these applications are successful because they have small rule bases, no chaining of inferences, and tunable parameters that can be adjusted to improve the system's performance. The fact that they are implemented with fuzzy operators might be incidental to their success; the key is simply to provide a concise and intuitive way to specify a smoothly interpolated, real-valued function.

There have been attempts to provide an explanation of fuzzy logic in terms of probability theory. One idea is to view assertions such as "Nate is Tall" as discrete observations made concerning a continuous hidden variable, Nate's actual $Height$. The probability model specifies $P(\text{Observer says Nate is tall} \mid Height)$, perhaps using a **probit distribution** as described on page 503. A posterior distribution over Nate's height can then be calculated in the usual way, for example if the model is part of a hybrid Bayesian network. Such an approach is not truth-functional, of course. For example, the conditional distribution

$$P(\text{Observer says Nate is tall and heavy} \mid Height, Weight)$$

allows for interactions between height and weight in the causing of the observation. Thus, someone who is eight feet tall and weighs 190 pounds is very unlikely to be called "tall and heavy," even though "eight feet" counts as "tall" and "190 pounds" counts as "heavy."

Fuzzy predicates can also be given a probabilistic interpretation in terms of **random sets**—that is, random variables whose possible values are sets of objects. For example, $Tall$ is a random set whose possible values are sets of people. The probability $P(Tall = S_1)$, where S_1 is some particular set of people, is the probability that exactly that set would be identified as "tall" by an observer. Then the probability that "Nate is tall" is the sum of the probabilities of all the sets of which Nate is a member.

RANDOM SETS

Both the hybrid Bayesian network approach and the random sets approach appear to capture aspects of fuzziness without introducing degrees of truth. Nonetheless, there remain many open issues concerning the proper representation of linguistic observations and continuous quantities—issues that have been neglected by most outside the fuzzy community.

14.8 SUMMARY

This chapter has described **Bayesian networks**, a well-developed representation for uncertain knowledge. Bayesian networks play a role roughly analogous to that of propositional logic for definite knowledge.

- A Bayesian network is a directed acyclic graph whose nodes correspond to random variables; each node has a conditional distribution for the node, given its parents.
- Bayesian networks provide a concise way to represent **conditional independence** relationships in the domain.
- A Bayesian network specifies a full joint distribution; each joint entry is defined as the product of the corresponding entries in the local conditional distributions. A Bayesian network is often exponentially smaller than the full joint distribution.
- Many conditional distributions can be represented compactly by canonical families of distributions. **Hybrid Bayesian networks**, which include both discrete and continuous variables, use a variety of canonical distributions.
- Inference in Bayesian networks means computing the probability distribution of a set of query variables, given a set of evidence variables. Exact inference algorithms, such as **variable elimination**, evaluate sums of products of conditional probabilities as efficiently as possible.
- In **polytrees** (singly connected networks), exact inference takes time linear in the size of the network. In the general case, the problem is intractable.
- Stochastic approximation techniques such as **likelihood weighting** and **Markov chain Monte Carlo** can give reasonable estimates of the true posterior probabilities in a network and can cope with much larger networks than can exact algorithms.
- Probability theory can be combined with representational ideas from first-order logic to produce very powerful systems for reasoning under uncertainty. **Relational probability models** (RPMs) include representational restrictions that guarantee a well-defined probability distribution that can be expressed as an equivalent Bayesian network.
- Various alternative systems for reasoning under uncertainty have been suggested. Generally speaking, **truth-functional** systems are not well suited for such reasoning.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The use of networks to represent probabilistic information began early in the 20th century, with the work of Sewall Wright on the probabilistic analysis of genetic inheritance and animal growth factors (Wright, 1921, 1934). One of his networks appears on the cover of this book. I. J. Good (1961), in collaboration with Alan Turing, developed probabilistic representations and Bayesian inference methods that could be regarded as a forerunner of modern Bayesian

networks—although the paper is not often cited in this context.[10] The same paper is the original source for the noisy-OR model.

The **influence diagram** representation for decision problems, which incorporated a DAG representation for random variables, was used in decision analysis in the late 1970s (see Chapter 16), but only enumeration was used for evaluation. Judea Pearl developed the message-passing method for carrying out inference in tree networks (Pearl, 1982a) and polytree networks (Kim and Pearl, 1983) and explained the importance of constructing causal rather than diagnostic probability models, in contrast to the certainty-factor systems then in vogue. The first expert system using Bayesian networks was CONVINCE (Kim, 1983; Kim and Pearl, 1987). More recent systems include the MUNIN system for diagnosing neuromuscular disorders (Andersen *et al.*, 1989) and the PATHFINDER system for pathology (Heckerman, 1991). By far the most widely used Bayesian network systems have been the diagnosis-and-repair modules (e.g., the Printer Wizard) in Microsoft Windows (Breese and Heckerman, 1996) and the Office Assistant in Microsoft Office (Horvitz *et al.*, 1998).

Pearl (1986) developed a clustering algorithm for exact inference in general Bayesian networks, utilizing a conversion to a directed polytree of clusters in which message passing was used to achieve consistency over variables shared between clusters. A similar approach, developed by the statisticians David Spiegelhalter and Steffen Lauritzen (Spiegelhalter, 1986; Lauritzen and Spiegelhalter, 1988), is based on conversion to an undirected (Markov) network. This approach is implemented in the HUGIN system, an efficient and widely used tool for uncertain reasoning (Andersen *et al.*, 1989). Ross Shachter, working in the influence diagram community, developed an exact method based on goal-directed reduction of the network, using posterior-preserving transformations (Shachter, 1986).

The variable elimination method described in the chapter is closest in spirit to Shachter's method, from which emerged the symbolic probabilistic inference (SPI) algorithm (Shachter *et al.*, 1990). SPI attempts to optimize the evaluation of expression trees such as that shown in Figure 14.8. The algorithm we describe is closest to that developed by Zhang and Poole (1994, 1996). Criteria for pruning irrelevant variables were developed by Geiger *et al.* (1990) and by Lauritzen *et al.* (1990); the criterion we give is a simple special case of these. Rina Dechter (1999) shows how the variable elimination idea is essentially identical to **nonserial dynamic programming** (Bertele and Brioschi, 1972), an algorithmic approach that can be applied to solve a range of inference problems in Bayesian networks—for example, finding the **most probable explanation** for a set of observations. This connects Bayesian network algorithms to related methods for solving CSPs and gives a direct measure of the complexity of exact inference in terms of the **hypertree width** of the network.

NONSERIAL DYNAMIC PROGRAMMING

The inclusion of continuous random variables in Bayesian networks was considered by Pearl (1988) and Shachter and Kenley (1989); these papers discussed networks containing only continuous variables with linear Gaussian distributions. The inclusion of discrete variables has been investigated by Lauritzen and Wermuth (1989) and implemented in the

[10] I. J. Good was chief statistician for Turing's code-breaking team in World War II. In *2001: A Space Odyssey* (Clarke, 1968a), Good and Minsky are credited with making the breakthrough that led to the development of the HAL 9000 computer.

cHUGIN system (Olesen, 1993). The probit distribution was studied first by Finney (1947), who called it the sigmoid distribution. It has been used widely for modeling discrete choice phenomena and can be extended to handle more than two choices (Daganzo, 1979). Bishop (1995) gives a justification for the use of the logit distribution.

Cooper (1990) showed that the general problem of inference in unconstrained Bayesian networks is NP-hard, and Paul Dagum and Mike Luby (1993) showed the corresponding approximation problem to be NP-hard. Space complexity is also a serious problem in both clustering and variable elimination methods. The method of **cutset conditioning**, which was developed for CSPs in Chapter 5, avoids the construction of exponentially large tables. In a Bayesian network, a cutset is a set of nodes that, when instantiated, reduces the remaining nodes to a polytree that can be solved in linear time and space. The query is answered by summing over all the instantiations of the cutset, so the overall space requirement is still linear (Pearl, 1988). Darwiche (2001) describes a recursive conditioning algorithm that allows a complete range of space/time tradeoffs.

The development of fast approximation algorithms for Bayesian network inference is a very active area, with contributions from statistics, computer science, and physics. The rejection sampling method is a general technique that is long known to statisticians; it was first applied to Bayesian networks by Max Henrion (1988), who called it **logic sampling**. Likelihood weighting, which was developed by Fung and Chang (1989) and Shachter and Peot (1989), is an example of the well-known statistical method of **importance sampling**. A large-scale application of likelihood weighting to medical diagnosis appears in Shwe and Cooper (1991). Cheng and Druzdzel (2000) describe an adaptive version of likelihood weighting that works well even when the evidence has very low prior likelihood.

Markov chain Monte Carlo (MCMC) algorithms began with the Metropolis algorithm, due to Metropolis *et al.* (1953), which was also the source of the simulated annealing algorithm described in Chapter 4. The Gibbs sampler was devised by Geman and Geman (1984) for inference in undirected Markov networks. The application of MCMC to Bayesian networks is due to Pearl (1987). The papers collected by Gilks *et al.* (1996) cover a wide variety of applications of MCMC, several of which were developed in the well-known BUGS package (Gilks *et al.*, 1994).

VARIATIONAL APPROXIMATION

There are two very important families of approximation methods that we did not cover in the chapter. The first is the family of **variational approximation** methods, which can be used to simplify complex calculations of all kinds. The basic idea is to propose a reduced version of the original problem that is simple to work with, but that resembles the original problem as closely as possible. The reduced problem is described by some **variational parameters** $\boldsymbol{\lambda}$ that are adjusted to minimize a distance function D between the original and the reduced problem, often by solving the system of equations $\partial D/\partial \boldsymbol{\lambda} = 0$. In many cases, strict upper and lower bounds can be obtained. Variational methods have long been used in statistics (Rustagi, 1976). In statistical physics, the **mean field** method is a particular variational approximation in which the individual variables making up the model are assumed to be completely independent. This idea was applied to solve large undirected Markov networks (Peterson and Anderson, 1987; Parisi, 1988). Saul *et al.* (1996) developed the mathematical foundations for applying variational methods to Bayesian networks and obtained

VARIATIONAL PARAMETERS

MEAN FIELD

accurate lower-bound approximations for sigmoid networks with the use of mean-field methods. Jaakkola and Jordan (1996) extended the methodology to obtain both lower and upper bounds. Variational approaches are surveyed by Jordan *et al.* (1999).

A second important family of approximation algorithms is based on Pearl's polytree message-passing algorithm (1982a). This algorithm can be applied to general networks, as suggested by Pearl (1988). The results might be incorrect, or the algorithm might fail to terminate, but in many cases, the values obtained are close to the true values. Little attention was paid to this so-called **belief propagation** (or **loopy propagation**) approach until McEliece *et al.* (1998) observed that message passing in a multiply-connected Bayesian network was exactly the computation performed by the **turbo decoding** algorithm (Berrou *et al.*, 1993), which provided a major breakthrough in the design of efficient error-correcting codes. The implication is that loopy propagation is both fast and accurate on the very large and very highly connected networks used for decoding and might therefore be useful more generally. Murphy *et al.* (1999) present an empirical study of where it does work. Yedidia *et al.* (2001) make further connections between loopy propagation and ideas from statistical physics.

BELIEF PROPAGATION

TURBO DECODING

The connection between probability and first-order languages was first studied by Carnap (1950). Gaifman (1964) and Scott and Krauss (1966) defined a language in which probabilities could be associated with first-order sentences and for which models were probability measures on possible worlds. Within AI, this idea was developed for propositional logic by Nilsson (1986) and for first-order logic by Halpern (1990). The first extensive investigation of knowledge representation issues in such languages was carried out by Bacchus (1990), and the paper by Wellman *et al.* (1992) surveys early implementation approaches based on the construction of equivalent propositional Bayesian networks. More recently, researchers have come to understand the importance of *complete* knowledge bases—that is, knowledge bases that, like Bayesian networks, define a unique joint distribution over all possible worlds. Methods for doing this have been based on probabilistic versions of logic programming (Poole, 1993; Sato and Kameya, 1997) or semantic networks (Koller and Pfeffer, 1998). Relational probability models of the kind described in this chapter are investigated in depth by Pfeffer (2000). Pasula and Russell (2001) examine both issues of relational and identity uncertainty within RPMs and the use of MCMC inference.

As explained in Chapter 13, early probabilistic systems fell out of favor in the early 1970s, leaving a partial vacuum to be filled by alternative methods. Certainty factors were invented for use in the medical expert system MYCIN (Shortliffe, 1976), which was intended both as an engineering solution and as a model of human judgment under uncertainty. The collection *Rule-Based Expert Systems* (Buchanan and Shortliffe, 1984) provides a complete overview of MYCIN and its descendants (see also Stefik, 1995). David Heckerman (1986) showed that a slightly modified version of certainty factor calculations gives correct probabilistic results in some cases, but results in serious overcounting of evidence in other cases. The PROSPECTOR expert system (Duda *et al.*, 1979) used a rule-based approach in which the rules were justified by a (seldom tenable) global independence assumption.

Dempster–Shafer theory originates with a paper by Arthur Dempster (1968) proposing a generalization of probability to interval values and a combination rule for using them. Later work by Glenn Shafer (1976) led to the Dempster-Shafer theory's being viewed as a

competing approach to probability. Ruspini *et al.* (1992) analyze the relationship between the Dempster-Shafer theory and standard probability theory. Shenoy (1989) has proposed a method for decision making with Dempster–Shafer belief functions.

Fuzzy sets were developed by Lotfi Zadeh (1965) in response to the perceived difficulty of providing exact inputs to intelligent systems. The text by Zimmermann (2001) provides a thorough introduction to fuzzy set theory; papers on fuzzy applications are collected in Zimmermann (1999). As we mentioned in the text, fuzzy logic has often been perceived incorrectly as a direct competitor to probability theory, whereas in fact it addresses a different set of issues. **Possibility theory** (Zadeh, 1978) was introduced to handle uncertainty in fuzzy systems and has much in common with probability. Dubois and Prade (1994) provide a thorough survey of the connections between possibility theory and probability theory.

POSSIBILITY THEORY

The resurgence of probability depended mainly on the discovery of Bayesian networks as a method for representing and using conditional independence information. This resurgence did not come without a fight; Peter Cheeseman's (1985) pugnacious "In Defense of Probability," and his later article "An Inquiry into Computer Understanding" (Cheeseman, 1988, with commentaries) give something of the flavor of the debate. One of the principal objections of the logicists was that the numerical calculations that probability theory was thought to require were not apparent to introspection and presumed an unrealistic level of precision in our uncertain knowledge. The development of **qualitative probabilistic networks** (Wellman, 1990a) provided a purely qualitative abstraction of Bayesian networks, using the notion of positive and negative influences between variables. Wellman shows that in many cases such information is sufficient for optimal decision making without the need for the precise specification of probability values. Work by Adnan Darwiche and Matt Ginsberg (1992) extracts the basic properties of conditioning and evidence combination from probability theory and shows that they can also be applied in logical and default reasoning.

The heart disease treatment system described in the chapter is due to Lucas (1996). Other fielded applications of Bayesian networks include the work at Microsoft on inferring computer user goals from their actions (Horvitz *et al.*, 1998) and on filtering junk email (Sahami *et al.*, 1998), the Electric Power Research Institute's work on monitoring power generators (Morjaria *et al.*, 1995), and NASA's work on displaying time-critical information at Mission Control in Houston (Horvitz and Barry, 1995).

Some important early papers on uncertain reasoning methods in AI are collected in the anthologies *Readings in Uncertain Reasoning* (Shafer and Pearl, 1990) and *Uncertainty in Artificial Intelligence* (Kanal and Lemmer, 1986). The most important single publication in the growth of Bayesian networks was undoubtedly the text *Probabilistic Reasoning in Intelligent Systems* (Pearl, 1988). Several excellent texts, including Lauritzen (1996), Jensen (2001) and Jordan (2003), contain more recent material. New research on probabilistic reasoning appears both in mainstream AI journals such as *Artificial Intelligence* and the *Journal of AI Research*, and in more specialized journals, such as the *International Journal of Approximate Reasoning*. Many papers on **graphical models**, which include Bayesian networks, appear in statistical journals. The proceedings of the conferences on Uncertainty in Artificial Intelligence (UAI), Neural Information Processing Systems (NIPS), and Artificial Intelligence and Statistics (AISTATS) are excellent sources for current research.

EXERCISES

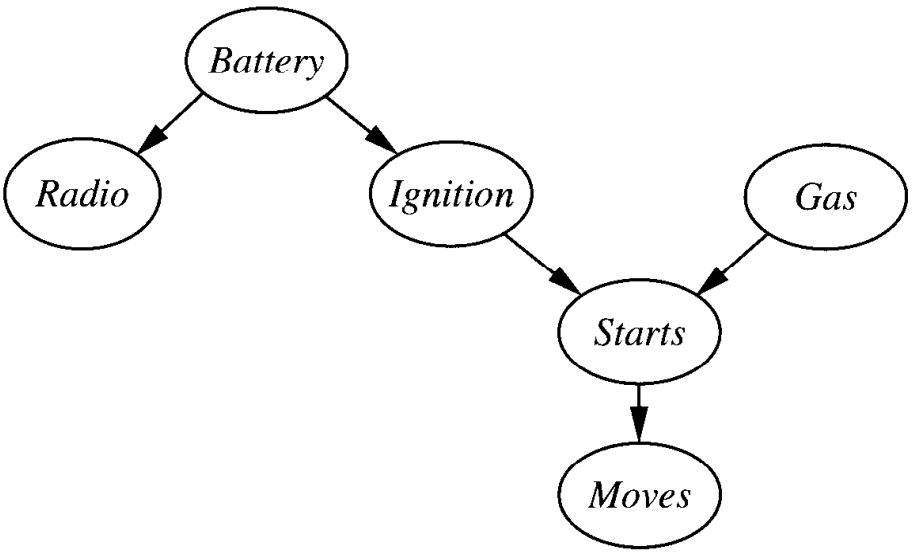


Figure 14.18 A Bayesian network describing some features of a car's electrical system and engine. Each variable is Boolean, and the *true* value indicates that the corresponding aspect of the vehicle is in working order.

14.1 Consider the network for car diagnosis shown in Figure 14.18.

a. Extend the network with the Boolean variables *IcyWeather* and *StarterMotor*.

b. Give reasonable conditional probability tables for all the nodes.

c. How many independent values are contained in the joint probability distribution for eight Boolean nodes, assuming that no conditional independence relations are known to hold among them?

d. How many independent probability values do your network tables contain?

e. The conditional distribution for *Starts* could be described as a **noisy-AND** distribution. Define this family in general and relate it to the noisy-OR distribution.

14.2 In your local nuclear power station, there is an alarm that senses when a temperature gauge exceeds a given threshold. The gauge measures the temperature of the core. Consider the Boolean variables A (alarm sounds), F_A (alarm is faulty), and F_G (gauge is faulty) and the multivalued nodes G (gauge reading) and T (actual core temperature).

a. Draw a Bayesian network for this domain, given that the gauge is more likely to fail when the core temperature gets too high.

b. Is your network a polytree?

c. Suppose there are just two possible actual and measured temperatures, normal and high; the probability that the gauge gives the correct temperature is x when it is working, but y when it is faulty. Give the conditional probability table associated with G.

d. Suppose the alarm works correctly unless it is faulty, in which case it never sounds. Give the conditional probability table associated with A.

e. Suppose the alarm and gauge are working and the alarm sounds. Calculate an expression for the probability that the temperature of the core is too high, in terms of the various conditional probabilities in the network.

14.3 Two astronomers in different parts of the world make measurements M_1 and M_2 of the number of stars N in some small region of the sky, using their telescopes. Normally, there is a small possibility e of error by up to one star in each direction. Each telescope can also (with a much smaller probability f) be badly out of focus (events F_1 and F_2), in which case the scientist will undercount by three or more stars (or, if N is less than 3, fail to detect any stars at all). Consider the three networks shown in Figure 14.19.

a. Which of these Bayesian networks are correct (but not necessarily efficient) representations of the preceding information?

b. Which is the best network? Explain.

c. Write out a conditional distribution for $\mathbf{P}(M_1|N)$, for the case where $N \in \{1, 2, 3\}$ and $M_1 \in \{0, 1, 2, 3, 4\}$. Each entry in the conditional distribution should be expressed as a function of the parameters e and/or f.

d. Suppose $M_1 = 1$ and $M_2 = 3$. What are the *possible* numbers of stars if we assume no prior constraint on the values of N?

e. What is the *most likely* number of stars, given these observations? Explain how to compute this, or, if it is not possible to compute, explain what additional information is needed and how it would affect the result.

14.4 Consider the network shown in Figure 14.19(ii), and assume that the two telescopes work identically. $N \in \{1, 2, 3\}$ and $M_1, M_2 \in \{0, 1, 2, 3, 4\}$, with the symbolic CPTs as described in Exercise 14.3. Using the enumeration algorithm, calculate the probability distribution $\mathbf{P}(N|M_1 = 2, M_2 = 2)$.

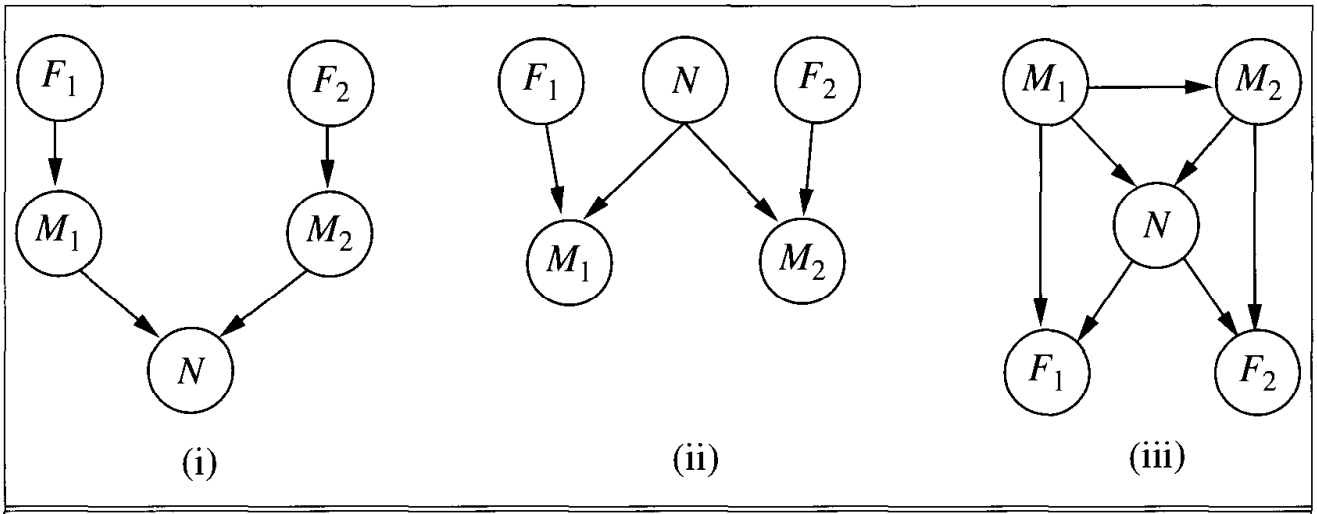


Figure 14.19 Three possible networks for the telescope problem.

14.5 Consider the family of linear Gaussian networks, as illustrated on page 502.

a. In a two-variable network, let X_1 be the parent of X_2, let X_1 have a Gaussian prior, and let $\mathbf{P}(X_2|X_1)$ be a linear Gaussian distribution. Show that the joint distribution $P(X_1, X_2)$ is a multivariate Gaussian, and calculate its covariance matrix.

b. Prove by induction that the joint distribution for a general linear Gaussian network on $X_1, \ldots, X_n$ is also a multivariate Gaussian.

14.6 The probit distribution defined on page 503 describes the probability distribution for a Boolean child, given a single continuous parent.

a. How might the definition be extended to cover multiple continuous parents?

b. How might it be extended to handle a *multivalued* child variable? Consider both cases where the child's values are ordered (as in selecting a gear while driving, depending on speed, slope, desired acceleration, etc.) and cases where they are unordered (as in selecting bus, train, or car to get to work). [*Hint*: Consider ways to divide the possible values into two sets, to mimic a Boolean variable.]

14.7 This exercise is concerned with the variable elimination algorithm in Figure 14.10.

a. Section 14.4 applies variable elimination to the query

$$\mathbf{P}(Burglary|JohnCalls = true, MaryCalls = true) .$$

Perform the calculations indicated and check that the answer is correct.

b. Count the number of arithmetic operations performed , and compare it with the number performed by the enumeration algorithm.

c. Suppose a network has the form of a *chain*: a sequence of Boolean variables $X_1, \ldots, X_n$ where $Parents(X_i) = \{X_{i-1}\}$ for $i = 2, \ldots, n$. What is the complexity of computing $\mathbf{P}(X_1|X_n = true)$ using enumeration? Using variable elimination?

d. Prove that the complexity of running variable elimination on a polytree network is linear in the size of the tree for any variable ordering consistent with the network structure.

14.8 Investigate the complexity of exact inference in general Bayesian networks:

a. Prove that any 3-SAT problem can be reduced to exact inference in a Bayesian network constructed to represent the particular problem and hence that exact inference is NP-hard. [*Hint*: Consider a network with one variable for each proposition symbol, one for each clause, and one for the conjunction of clauses.]

b. The problem of counting the number of satisfying assignments for a 3-SAT problem is #P-complete. Show that exact inference is at least as hard as this.

14.9 Consider the problem of generating a random sample from a specified distribution on a single variable. You can assume that a random number generator is available that returns a random number uniformly distributed between 0 and 1.

a. Let X be a discrete variable with $P(X = x_i) = p_i$ for $i \in \{1, \ldots, k\}$. The **cumulative distribution** of X gives the probability that $X \in \{x_1, \ldots, x_j\}$ for each possible j. Ex-

CUMULATIVE DISTRIBUTION

plain how to calculate the cumulative distribution in $O(k)$ time and how to generate a single sample of X from it. Can the latter be done in less than $O(k)$ time?

b. Now suppose we want to generate N samples of X, where $N \gg k$. Explain how to do this with an expected runtime per sample that is *constant* (i.e., independent of k).

c. Now consider a continuous-valued variable with a parametrized distribution (e.g., Gaussian). How can samples be generated from such a distribution?

d. Suppose you want to query a continuous-valued variable and you are using a sampling algorithm such as LIKELIHOODWEIGHTING to do the inference. How would you have to modify the query-answering process?

14.10 The **Markov blanket** of a variable is defined on page 499.

a. Prove that a variable is independent of all other variables in the network, given its Markov blanket.

b. Derive Equation (14.11).

14.11 Consider the query $\mathbf{P}(Rain|Sprinkler = true, WetGrass = true)$ in Figure 14.11(a) and how MCMC can answer it.

a. How many states does the Markov chain have?

b. Calculate the **transition matrix Q** containing $q(\mathbf{y} \rightarrow \mathbf{y}')$ for all $\mathbf{y}, \mathbf{y}'$.

c. What does $\mathbf{Q}^2$, the square of the transition matrix, represent?

d. What about $\mathbf{Q}^n$ as $n \rightarrow \infty$?

e. Explain how to do probabilistic inference in Bayesian networks, assuming that $\mathbf{Q}^n$ is available. Is this a practical way to do inference?

14.12 Three soccer teams A, B, and C, play each other once. Each match is between two teams, and can be won, drawn, or lost. Each team has a fixed, unknown degree of quality—an integer ranging from 0 to 3—and the outcome of a match depends probabilistically on the difference in quality between the two teams.

a. Construct a relational probability model to describe this domain, and suggest numerical values for all the necessary probability distributions.

b. Construct the equivalent Bayesian network.

c. Suppose that in the first two matches A beats B and draws with C. Using an exact inference algorithm of your choice, compute the posterior distribution for the outcome of the third match.

d. Suppose there are n teams in the league and we have the results for all but the last match. How does the complexity of predicting the last game vary with n?

e. Investigate the application of MCMC to this problem. How quickly does it converge in practice and how well does it scale?

15 PROBABILISTIC REASONING OVER TIME

In which we try to interpret the present, understand the past, and perhaps predict the future, even when very little is crystal clear.

Agents in uncertain environments must be able to keep track of the current state of the environment, just as logical agents must. The task is made more difficult by partial and noisy percepts and uncertainty about how the environment changes over time. At best, the agent will be able to obtain only a probabilistic assessment of the current situation. This chapter describes the representations and inference algorithms that make that assessment possible, building on the ideas introduced in Chapter 14.

The basic approach is described in Section 15.1: a changing world is modeled using a random variable for each aspect of the world state *at each point in time*. The relations among these variables describe how the state evolves. Section 15.2 defines the basic inference tasks and describes the general structure of inference algorithms for temporal models. Then we describe three specific kinds of models: **hidden Markov models**, **Kalman filters**, and **dynamic Bayesian networks** (which include hidden Markov models and Kalman filters as special cases). Finally, Section 15.6 explains how temporal probability models form the core of modern speech recognition systems. Learning plays a central role in the construction of all these models, but a detailed investigation of learning algorithms is left until Part VI.

15.1 TIME AND UNCERTAINTY

We have developed our techniques for probabilistic reasoning in the context of **static** worlds, in which each random variable has a single fixed value. For example, when repairing a car, we assume that whatever is broken remains broken during the process of diagnosis; our job is to infer the state of the car from observed evidence, which also remains fixed.

Now consider a slightly different problem: treating a diabetic patient. As in the case of car repair, we have evidence such as recent insulin doses, food intake, blood sugar measurements, and other physical signs. The task is to assess the current state of the patient, including the actual blood sugar level and insulin level. Given this information, the doctor (or patient) makes a decision about the patient's food intake and insulin dose. Unlike the case

of car repair, here the *dynamic* aspects of the problem are essential. Blood sugar levels and measurements thereof can change rapidly over time, depending on one's recent food intake and insulin doses, one's metabolic activity, the time of day, and so on. To assess the current state from the history of evidence and to predict the outcomes of treatment actions, we must model these changes.

The same considerations arise in many other contexts, ranging from tracking the economic activity of a nation, given approximate and partial statistics, to understanding a sequence of spoken words, given noisy and ambiguous acoustic measurements. How can dynamic situations like these be modeled?

States and observations

The basic approach we will adopt is similar to the idea underlying situation calculus, as described in Chapter 10: the process of change can be viewed as a series of snapshots, each of which describes the state of the world at a particular time. Each snapshot, or **time slice**, contains a set of random variables, some of which are observable and some of which are not. For simplicity, we will assume that the same subset of variables is observable in each slice (although this is not strictly necessary in anything that follows). We will use $\mathbf{X}_t$ to denote the set of unobservable state variables at time t and $\mathbf{E}_t$ to denote the set of observable evidence variables. The observation at time t is $\mathbf{E}_t = \mathbf{e}_t$ for some set of values $\mathbf{e}_t$.

TIME SLICE

Consider the following oversimplified example: Suppose you are the security guard at some secret underground installation. You want to know whether it's raining today, but your only access to the outside world occurs each morning when you see the director coming in with, or without, an umbrella. For each day t, the set $\mathbf{E}_t$ thus contains a single evidence variable U_t (whether the umbrella appears), and the set $\mathbf{X}_t$ contains a single state variable R_t (whether it is raining). Other problems can involve larger sets of variables. In the diabetes example, we might have evidence variables such as $MeasuredBloodSugar_t$ and $PulseRate_t$, and state variables such as $BloodSugar_t$ and $StomachContents_t$.[1]

The interval between time slices also depends on the problem. For diabetes monitoring, a suitable interval might be an hour rather than a day. In this chapter, we will generally assume a fixed, finite interval; this means that times can be labeled by integers. We will assume that the state sequence starts at $t = 0$; for various uninteresting reasons, we will assume that evidence starts arriving at $t = 1$ rather than $t = 0$. Hence, our umbrella world is represented by state variables R_0, R_1, $R_2, \ldots$ and evidence variables U_1, $U_2, \ldots$. We will use the notation $a{:}b$ to denote the sequence of integers from a to b (inclusive), and the notation $\mathbf{X}_{a:b}$ to denote the corresponding set of variables from $\mathbf{X}_a$ to $\mathbf{X}_b$. For example, $U_{1:3}$ corresponds to the variables U_1, U_2, U_3.

Stationary processes and the Markov assumption

With the set of state and evidence variables for a given problem decided on, the next step is to specify the dependencies among the variables. We could follow the procedure laid down

[1] Notice that $BloodSugar_t$ and $MeasuredBloodSugar_t$ are not the same variable; this is how we deal with noisy measurements of actual quantities.

in Chapter 14, placing the variables in some order and asking questions about conditional independence of predecessors, given some set of parents. One obvious choice is to order the variables in their natural temporal order, since cause usually precedes effect and we prefer to add the variables in causal order.

We would quickly run into an obstacle, however: the set of variables is unbounded, because it includes the state and evidence variables for every time slice. This actually creates two problems: first, we might have to specify an unbounded number of conditional probability tables, one for each variable in each slice; second, each one might involve an unbounded number of parents.

STATIONARY PROCESS

The first problem is solved by assuming that changes in the world state are caused by a **stationary process**—that is, a process of change that is governed by laws that do not themselves change over time. (Don't confuse *stationary* with *static*: in a *static* process, the state itself does not change.) In the umbrella world, then, the conditional probability that the umbrella appears, $\mathbf{P}(U_t|Parents(U_t))$, is the same for all t. Given the assumption of stationarity, therefore, we need specify conditional distributions only for the variables within a "representative" time slice.

MARKOV ASSUMPTION

The second problem, that of handling the potentially infinite number of parents, is solved by making what is called a **Markov assumption**—that is, that the current state depends on only a *finite* history of previous states. Processes satisfying this assumption were first studied in depth by the Russian statistician Andrei Markov and are called **Markov processes** or **Markov chains**. They come in various flavors; the simplest is the **first-order Markov process**, in which the current state depends only on the previous state and not on any earlier states. In other words, a state is the information you need to make the future independent of the past given the state. Using our notation, the corresponding conditional independence assertion states that, for all t,

MARKOV PROCESSES

FIRST-ORDER MARKOV PROCESS

$$\mathbf{P}(\mathbf{X}_t|\mathbf{X}_{0:t-1}) = \mathbf{P}(\mathbf{X}_t|\mathbf{X}_{t-1}) \ . \tag{15.1}$$

Hence, in a first-order Markov process, the laws describing how the state evolves over time are contained entirely within the conditional distribution $\mathbf{P}(\mathbf{X}_t|\mathbf{X}_{t-1})$, which we call the **transition model** for first-order processes.[2] The transition model for a second-order Markov process is the conditional distribution $\mathbf{P}(\mathbf{X}_t|\mathbf{X}_{t-2}, \mathbf{X}_{t-1})$. Figure 15.1 shows the Bayesian network structures corresponding to first-order and second-order Markov processes.

In addition to restricting the parents of the state variables $\mathbf{X}_t$, we must restrict the parents of the evidence variables $\mathbf{E}_t$. Typically, we will assume that the evidence variables at time t depend only on the current state:

$$\mathbf{P}(\mathbf{E}_t|\mathbf{X}_{0:t}, \mathbf{E}_{0:t-1}) = \mathbf{P}(\mathbf{E}_t|\mathbf{X}_t) \ . \tag{15.2}$$

SENSOR MODEL

The conditional distribution $\mathbf{P}(\mathbf{E}_t|\mathbf{X}_t)$ is called the **sensor model** (or sometimes the **observation model**), because it describes how the "sensors"—that is, the evidence variables—are affected by the actual state of the world. Notice the direction of the dependence: the "arrow" goes from state to sensor values because the state of the world *causes* the sensors to take on

[2] The transition model is the probabilistic analog of the Boolean update circuits in Chapter 7 and the successor-state axioms in Chapter 10.

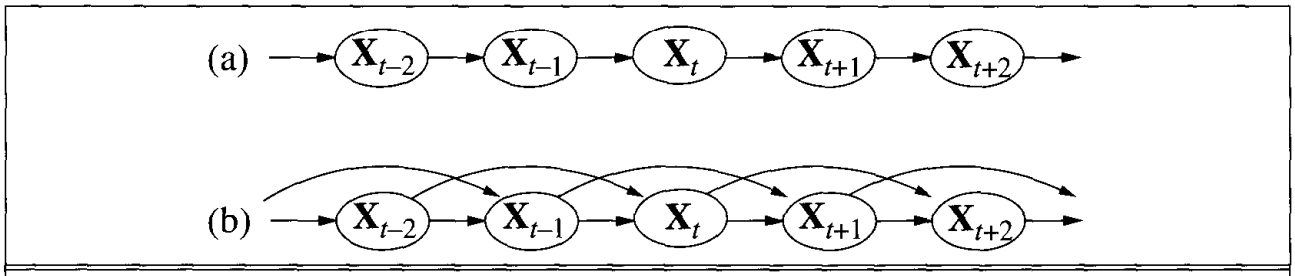


Figure 15.1 (a) Bayesian network structure corresponding to a first-order Markov process with state defined by the variables $\mathbf{X}_t$. (b) A second-order Markov process.

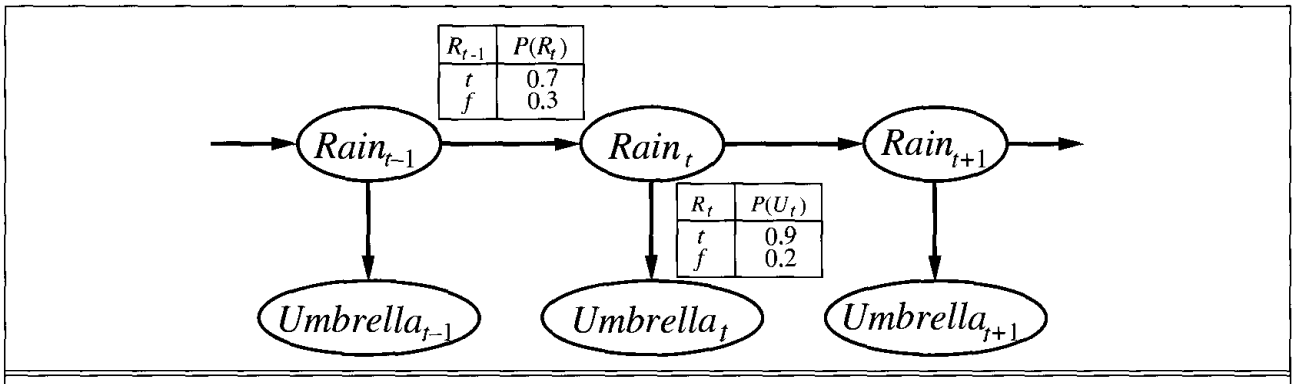


Figure 15.2 Bayesian network structure and conditional distributions describing the umbrella world. The transition model is $P(Rain_t|Rain_{t-1})$ and the sensor model is $P(Umbrella_t|Rain_t)$.

particular values. In the umbrella world, for example, the rain *causes* the umbrella to appear. (The inference process, of course, goes in the other direction; the distinction between the direction of modeled dependencies and the direction of inference is one of the principal advantages of Bayesian networks.)

In addition to the transition model and sensor model, we need to specify a prior probability $\mathbf{P}(\mathbf{X}_0)$ over the states at time 0. These three distributions, combined with the conditional independence assertions in Equations (15.1) and (15.2), give us a specification of the complete joint distribution over all the variables. For any finite t, we have

$$\mathbf{P}(\mathbf{X}_0, \mathbf{X}_1, \ldots, \mathbf{X}_t, \mathbf{E}_1, \ldots, \mathbf{E}_t) = \mathbf{P}(\mathbf{X}_0) \prod_{i=1}^{t} \mathbf{P}(\mathbf{X}_i|\mathbf{X}_{i-1})\mathbf{P}(\mathbf{E}_i|\mathbf{X}_i) \ .$$

The independence assumptions correspond to a very simple structure for the Bayesian network describing the whole system. Figure 15.2 shows the network structure for the umbrella example, including the conditional distributions for the transition and sensor models.

The structure in the figure assumes a first-order Markov process, because the probability of rain is assumed to depend only on whether it rained the previous day. Whether such an assumption is reasonable depends on the domain itself. The first-order Markov assumption says that the state variables contain *all* the information needed to characterize the probability distribution for the next time slice. Sometimes the assumption is exactly true—for example,

RANDOM WALK

if a particle is executing a **random walk** along the x-axis, changing its position by ± 1 at each time step, then using the x-coordinate as the state gives a first-order Markov process.

Sometimes the assumption is only approximate, as in the case of predicting rain only on the basis of whether it rained the previous day. There are two possible fixes if the approximation proves too inaccurate:

1. Increasing the order of the Markov process model. For example, we could make a second-order model by adding $Rain_{t-2}$ as a parent of $Rain_t$, which might give slightly more accurate predictions (for example, in Palo Alto it very rarely rains more than two days in a row).
2. Increasing the set of state variables. For example, we could add $Season_t$ to allow us to incorporate historical records of rainy seasons, or we could add $Temperature_t$, $Humidity_t$ and $Pressure_t$ to allow us to use a physical model of rainy conditions.

Exercise 15.1 asks you to show that the first solution—increasing the order—can always be reformulated as an increase in the set of state variables, keeping the order fixed. Notice that adding state variables might improve the system's predictive power but also increases the prediction *requirements*: we now have to predict the new variables as well. Thus, we are looking for a "self-sufficient" set of variables, which really means that we have to understand the "physics" of the process being modeled. The requirement for accurate modeling of the process is obviously lessened if we can add new sensors (e.g., measurements of temperature and pressure) that provide information directly about the new state variables.

Consider, for example, the problem of tracking a robot wandering randomly on the X–Y plane. One might propose that the position and velocity are a sufficient set of state variables: one can simply use Newton's laws to calculate the new position, and the velocity may change unpredictably. If the robot is battery-powered, however, then battery exhaustion would tend to have a systematic effect on the change in velocity. Because this in turn depends on how much power was used by all previous maneuvers, the Markov property is violated. We can restore the Markov property by including the charge level $Battery_t$ as one of the state variables that make up $\mathbf{X}_t$. This helps in predicting the motion of the robot, but in turn requires a model for predicting $Battery_t$ from $Battery_{t-1}$ and the velocity. In some cases, that can be done reliably; accuracy would be improved by *adding a new sensor* for the battery level

15.2 Inference in Temporal Models

Having set up the structure of a generic temporal model, we can formulate the basic inference tasks that must be solved:

FILTERING
MONITORING
BELIEF STATE

◇ **Filtering** or **monitoring**: This is the task of computing the **belief state**—the posterior distribution over the current state, given all evidence to date. That is, we wish to compute $\mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t})$, assuming that evidence arrives in a continuous stream beginning at $t = 1$. In the umbrella example, this would mean computing the probability of rain today, given all the observations of the umbrella carrier made so far. Filtering is what a rational agent needs to do in order to keep track of the current state so that rational decisions can be made. (See Chapter 17.) It turns out that an almost identical calculation provides the **likelihood** of the evidence sequence, $P(\mathbf{e}_{1:t})$.

PREDICTION

◇ **Prediction**: This is the task of computing the posterior distribution over the *future* state, given all evidence to date. That is, we wish to compute $\mathbf{P}(\mathbf{X}_{t+k}|\mathbf{e}_{1:t})$ for some $k > 0$. In the umbrella example, this might mean computing the probability of rain three days from now, given all the observations of the umbrella-carrier made so far. Prediction is useful for evaluating possible courses of action.

SMOOTHING
HINDSIGHT

◇ **Smoothing**, or **hindsight**: This is the task of computing the posterior distribution over a *past* state, given all evidence up to the present. That is, we wish to compute $\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t})$ for some k such that $0 \leq k < t$. In the umbrella example, it might mean computing the probability that it rained last Wednesday, given all the observations of the umbrella carrier made up to today. Hindsight provides a better estimate of the state than was available at the time, because it incorporates more evidence.

◇ **Most likely explanation**: Given a sequence of observations, we might wish to find the sequence of states that is most likely to have generated those observations. That is, we wish to compute $\text{argmax}_{\mathbf{x}_{1:t}} P(\mathbf{x}_{1:t}|\mathbf{e}_{1:t})$. For example, if the umbrella appears on each of the first three days and is absent on the fourth, then the most likely explanation is that it rained on the first three days and did not rain on the fourth. Algorithms for this task are useful in many applications, including speech recognition—where the aim is to find the most likely sequence of words, given a series of sounds—and the reconstruction of bit strings transmitted over a noisy channel.

In addition to these tasks, methods are needed for *learning* the transition and sensor models from observations. Just as with static Bayesian networks, dynamic Bayes net learning can be done as a by-product of inference. Inference provides an estimate of what transitions actually occurred and of what states generated the sensor readings, and these estimates can be used to update the models. The updated models provide new estimates, and the process iterates to convergence. The overall process is an instance of the expectation-maximization or **EM algorithm**. (See Section 20.3.) One point to note is that learning requires the full smoothing inference, rather than filtering, because it provides better estimates of the states of the process. Learning with filtering can fail to converge correctly; consider, for example, the problem of learning to solve murders: hindsight is *always* required to infer what happened at the murder scene from the observable variables.

Algorithms for the four inference tasks listed in the preceding paragraph can be described first at a generic level, independently of the particular kind of model employed. Improvements specific to each model will be described in the corresponding sections.

Filtering and prediction

Let us begin with filtering. We will show that this can be done in a simple online fashion: given the result of filtering up to time t, one can easily compute the result for $t+1$ from the new evidence $\mathbf{e}_{t+1}$. That is,

$$\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) = f(\mathbf{e}_{t+1}, \mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t}))\ .$$

RECURSIVE ESTIMATION

for some function f. This process is often called **recursive estimation**. We can view the calculation as actually being composed of two parts: first, the current state distribution is

projected forward from t to $t+1$; then it is updated using the new evidence $\mathbf{e}_{t+1}$. This two-part process emerges quite simply:

$$\begin{aligned}\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) &= \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}, \mathbf{e}_{t+1}) \quad \text{(dividing up the evidence)}\\ &= \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1}, \mathbf{e}_{1:t})\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}) \quad \text{(using Bayes' rule)}\\ &= \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}) \quad \text{(by the Markov property of evidence).}\end{aligned}$$

Here and throughout this chapter, α is a normalizing constant used to make probabilities sum up to 1. The second term, $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t})$ represents a one-step prediction of the next state, and the first term updates this with the new evidence; notice that $\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})$ is obtainable directly from the sensor model. Now we obtain the one-step prediction for the next state by conditioning on the current state $\mathbf{X}_t$:

$$\begin{aligned}\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) &= \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})\sum_{\mathbf{x}_t}\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t, \mathbf{e}_{1:t})P(\mathbf{x}_t|\mathbf{e}_{1:t})\\ &= \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})\sum_{\mathbf{x}_t}\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t)P(\mathbf{x}_t|\mathbf{e}_{1:t}) \quad \text{(using the Markov property).} \qquad (15.3)\end{aligned}$$

Within the summation, the first factor is simply the transition model and the second is the current state distribution. Hence, we have the desired recursive formulation. We can think of the filtered estimate $\mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t})$ as a "message" $\mathbf{f}_{1:t}$ that is propagated forward along the sequence, modified by each transition and updated by each new observation. The process is

$$\mathbf{f}_{1:t+1} = \alpha\,\text{FORWARD}(\mathbf{f}_{1:t}, \mathbf{e}_{t+1})$$

where FORWARD implements the update described in Equation (15.3).

When all the state variables are discrete, the time for each update is constant (i.e., independent of t), and the space required is also constant. (The constants depend, of course, on the size of the state space and the specific type of the temporal model in question.) *The time and space requirements for updating must be constant if an agent with limited memory is to keep track of the current state distribution over an unbounded sequence of observations.*

Let us illustrate the filtering process for two steps in the basic umbrella example. (See Figure 15.2.) We assume that our security guard has some prior belief about whether it rained on day 0, just before the observation sequence begins. Let's suppose this is $\mathbf{P}(R_0) = \langle 0.5, 0.5\rangle$. Now we process the two observations as follows:

- On day 1, the umbrella appears, so $U_1 = \mathit{true}$. The prediction from $t=0$ to $t=1$ is

$$\begin{aligned}\mathbf{P}(R_1) &= \sum_{r_0}\mathbf{P}(R_1|r_0)P(r_0)\\ &= \langle 0.7, 0.3\rangle \times 0.5 + \langle 0.3, 0.7\rangle \times 0.5 = \langle 0.5, 0.5\rangle\ ,\end{aligned}$$

  and updating it with the evidence for $t=1$ gives

$$\begin{aligned}\mathbf{P}(R_1|u_1) &= \alpha\mathbf{P}(u_1|R_1)\mathbf{P}(R_1) = \alpha\,\langle 0.9, 0.2\rangle\langle 0.5, 0.5\rangle\\ &= \alpha\,\langle 0.45, 0.1\rangle \approx \langle 0.818, 0.182\rangle\ .\end{aligned}$$

- On day 2, the umbrella appears, so $U_2 = \mathit{true}$. The prediction from $t=1$ to $t=2$ is

$$\begin{aligned}\mathbf{P}(R_2|u_1) &= \sum_{r_1}\mathbf{P}(R_2|r_1)P(r_1|u_1)\\ &= \langle 0.7, 0.3\rangle \times 0.818 + \langle 0.3, 0.7\rangle \times 0.182 \approx \langle 0.627, 0.373\rangle\ ,\end{aligned}$$

and updating it with the evidence for $t = 2$ gives

$$\begin{aligned}\mathbf{P}(R_2|u_1, u_2) &= \alpha\mathbf{P}(u_2|R_2)\mathbf{P}(R_2|u_1) = \alpha\langle 0.9, 0.2\rangle\langle 0.627, 0.373\rangle \\ &= \alpha\langle 0.565, 0.075\rangle \approx \langle 0.883, 0.117\rangle\ .\end{aligned}$$

Intuitively, the probability of rain increases from day 1 to day 2 because rain persists. Exercise 15.2(a) asks you to investigate this tendency further.

The task of **prediction** can be seen simply as filtering without the addition of new evidence. In fact, the filtering process already incorporates a one-step prediction, and it is easy to derive the following recursive computation for predicting the state at $t + k + 1$ from a prediction for $t + k$:

$$\mathbf{P}(\mathbf{X}_{t+k+1}|\mathbf{e}_{1:t}) = \sum_{\mathbf{x}_{t+k}} \mathbf{P}(\mathbf{X}_{t+k+1}|\mathbf{x}_{t+k})P(\mathbf{x}_{t+k}|\mathbf{e}_{1:t})\ . \tag{15.4}$$

Naturally, this computation involves only the transition model and not the sensor model.

It is interesting to consider what happens as we try to predict further and further into the future. As Exercise 15.2(b) shows, the predicted distribution for rain converges to a fixed point $\langle 0.5, 0.5\rangle$, after which it remains constant for all time. This is the **stationary distribution** of the Markov process defined by the transition model. (See also page 517.) A great deal is known about the properties of such distributions and about the **mixing time**—roughly, the time taken to reach the fixed point. In practical terms, this dooms to failure any attempt to predict the *actual* state for a number of steps that is more than a small fraction of the mixing time. The more uncertainty there is in the transition model, the shorter will be the mixing time and the more the future is obscured.

MIXING TIME

In addition to filtering and prediction, we can use a forward recursion to compute the **likelihood** of the evidence sequence, $P(\mathbf{e}_{1:t})$. This is a useful quantity if we want to compare different temporal models that might have produced the same evidence sequence; for example, in Section 15.6, we compare different words that might have produced the same sound sequence. For this recursion, we use a likelihood message $\ell_{1:t} = \mathbf{P}(\mathbf{X}_t, \mathbf{e}_{1:t})$. It is a simple exercise to show that

$$\ell_{1:t+1} = \text{FORWARD}(\ell_{1:t}, \mathbf{e}_{t+1})\ .$$

Having computed $\ell_{1:t}$, we obtain the actual likelihood by summing out $\mathbf{X}_t$:

$$L_{1:t} = P(\mathbf{e}_{1:t}) = \sum_{\mathbf{x}_t} \ell_{1:t}(\mathbf{x}_t)\ . \tag{15.5}$$

Smoothing

As we said earlier, **smoothing** is the process of computing the distribution over past states given evidence up to the present; that is, $\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t})$ for $1 \leq k < t$. (See Figure 15.3.) This is done most conveniently in two parts—the evidence up to k and the evidence from $k + 1$ to t,

$$\begin{aligned}\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t}) &= \mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:k}, \mathbf{e}_{k+1:t}) \\ &= \alpha\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:k})\mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k, \mathbf{e}_{1:k}) \quad \text{(using Bayes' rule)} \\ &= \alpha\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:k})\mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k) \quad \text{(using conditional independence)} \\ &= \alpha\mathbf{f}_{1:k}\mathbf{b}_{k+1:t}\ ,\end{aligned} \tag{15.6}$$

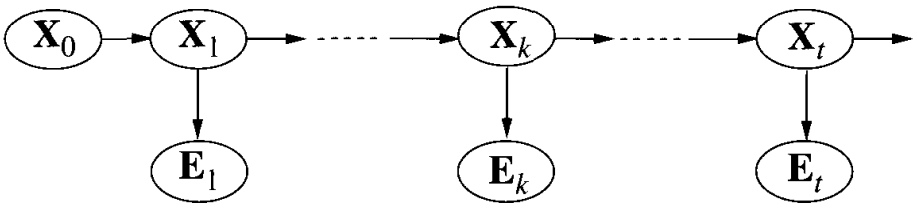


Figure 15.3 Smoothing computes $\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t})$, the posterior distribution of the state at some past time k given a complete sequence of observations from 1 to t.

where we have defined a "backward" message $\mathbf{b}_{k+1:t} = \mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k)$, analogous to the forward message $\mathbf{f}_{1:k}$. The forward message $\mathbf{f}_{1:k}$ can be computed by filtering forward from 1 to k, as given by Equation (15.3). It turns out that the backward message $\mathbf{b}_{k+1:t}$ can be computed by a recursive process that runs *backwards* from t:

$$\begin{aligned}
\mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k) &= \sum_{\mathbf{x}_{k+1}} \mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k, \mathbf{x}_{k+1})\mathbf{P}(\mathbf{x}_{k+1}|\mathbf{X}_k) \quad \text{(conditioning on } \mathbf{X}_{k+1}\text{)} \\
&= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1:t}|\mathbf{x}_{k+1})\mathbf{P}(\mathbf{x}_{k+1}|\mathbf{X}_k) \quad \text{(by conditional independence)} \\
&= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1}, \mathbf{e}_{k+2:t}|\mathbf{x}_{k+1})\mathbf{P}(\mathbf{x}_{k+1}|\mathbf{X}_k) \\
&= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1}|\mathbf{x}_{k+1})P(\mathbf{e}_{k+2:t}|\mathbf{x}_{k+1})\mathbf{P}(\mathbf{x}_{k+1}|\mathbf{X}_k)\,, \qquad (15.7)
\end{aligned}$$

where the last step follows by the conditional independence of $\mathbf{e}_{k+1}$ and $\mathbf{e}_{k+2:t}$, given $\mathbf{X}_{k+1}$. Of the three factors in this summation, the first and third are obtained directly from the model, and the second is the "recursive call." Using the message notation, we have

$$\mathbf{b}_{k+1:t} = \text{BACKWARD}(\mathbf{b}_{k+2:t}, \mathbf{e}_{k+1:t})$$

where BACKWARD implements the update described in Equation (15.7). As with the forward recursion, the time and space needed for each update are constant and thus independent of t.

We can now see that the two terms in Equation (15.6) can both be computed by recursions through time, one running forward from 1 to k and using the filtering equation (15.3) and the other running backward from t to $k+1$ and using Equation (15.7). Note that the backward phase is initialized with $\mathbf{b}_{t+1:t} = \mathbf{P}(\mathbf{e}_{t+1:t}|\mathbf{X}_t) = \mathbf{1}$, where $\mathbf{1}$ is a vector of ones. (Because $\mathbf{e}_{t+1:t}$ is an empty sequence, the probability of observing it is 1.)

Let us now apply this algorithm to the umbrella example, computing the smoothed estimate for the probability of rain at $t = 1$, given the umbrella observations on days 1 and 2. From Equation (15.6), this is given by

$$\mathbf{P}(R_1|u_1, u_2) = \alpha\mathbf{P}(R_1|u_1)\mathbf{P}(u_2|R_1)\,. \qquad (15.8)$$

The first term we already know to be $\langle .818, .182\rangle$, from the forward filtering process described earlier. The second term can be computed by applying the backward recursion in Equation (15.7):

$$\begin{aligned}
\mathbf{P}(u_2|R_1) &= \sum_{r_2} P(u_2|r_2)P(\,|r_2)\mathbf{P}(r_2|R_1) \\
&= (0.9 \times 1 \times \langle 0.7, 0.3\rangle) + (0.2 \times 1 \times \langle 0.3, 0.7\rangle) = \langle 0.69, 0.41\rangle\,.
\end{aligned}$$

Plugging this into Equation (15.8), we find that the smoothed estimate for rain on day 1 is

$$\mathbf{P}(R_1|u_1, u_2) = \alpha \langle 0.818, 0.182 \rangle \times \langle 0.69, 0.41 \rangle \approx \langle 0.883, 0.117 \rangle .$$

Thus, the smoothed estimate is *higher* than the filtered estimate (0.818) in this case. This is because the umbrella on day 2 makes it more likely to have rained on day 2; in turn, because rain tends to persist, that makes it more likely to have rained on day 1.

Both the forward and backward recursions take a constant amount of time per step; hence, the time complexity of smoothing with respect to evidence $\mathbf{e}_{1:t}$ is $O(t)$. This is the complexity for smoothing at a particular time step k. If we want to smooth the whole sequence, one obvious method is simply to run the whole smoothing process once for each time step to be smoothed. This results in a time complexity of $O(t^2)$. A better approach uses a very simple application of dynamic programming to reduce the complexity to $O(t)$. A clue appears in the preceding analysis of the umbrella example, where we were able to reuse the results of the forward filtering phase. The key to the linear-time algorithm is to *record the results* of forward filtering over the whole sequence. Then we run the backward recursion from t down to 1, computing the smoothed estimate at each step k from the computed backward message $\mathbf{b}_{k+1:t}$ and the stored forward message $\mathbf{f}_{1:k}$. The algorithm, aptly called the **forward–backward algorithm**, is shown in Figure 15.4.

FORWARD–BACKWARD ALGORITHM

The alert reader will have spotted that the Bayesian network structure shown in Figure 15.3 is a **polytree** in the terminology of Chapter 14. This means that a straightforward application of the clustering algorithm also yields a linear-time algorithm that computes smoothed estimates for the entire sequence. It is now understood that the forward–backward algorithm is in fact a special case of the polytree propagation algorithm used with clustering methods (although the two were developed independently).

```
function FORWARD-BACKWARD(ev, prior) returns a vector of probability distributions
  inputs: ev, a vector of evidence values for steps 1, ..., t
          prior, the prior distribution on the initial state, P(X_0)
  local variables: fv, a vector of forward messages for steps 0, ..., t
                   b, a representation of the backward message, initially all 1s
                   sv, a vector of smoothed estimates for steps 1, ..., t

  fv[0] ← prior
  for i = 1 to t do
      fv[i] ← FORWARD(fv[i − 1], ev[i])
  for i = t downto 1 do
      sv[i] ← NORMALIZE(fv[i] × b)
      b ← BACKWARD(b, ev[i])
  return sv
```

Figure 15.4 The forward–backward algorithm for computing posterior probabilities of a sequence of states given a sequence of observations. The FORWARD and BACKWARD operators are defined by Equations (15.3) and (15.7), respectively.

The forward–backward algorithm forms the backbone of the computational methods employed in many applications that deal with sequences of noisy observations, ranging from speech recognition to radar tracking of aircraft. As described, it has two practical drawbacks. The first is that its space complexity can be too high for applications where the state space is large and the sequences are long. It uses $O(|\mathbf{f}|t)$ space where $|\mathbf{f}|$ is the size of the representation of the forward message. The space requirement can be reduced to $O(|\mathbf{f}| \log t)$ with a concomitant increase in the time complexity by a factor of $\log t$, as shown in Exercise 15.3. In some cases (see Section 15.3), a constant-space algorithm can be used with no time penalty.

The second drawback of the basic algorithm is that it needs to be modified to work in an *online* setting where smoothed estimates must be computed for earlier time slices as new observations are continuously added to the end of the sequence. The most common requirement is for **fixed-lag smoothing**, which requires computing the smoothed estimate $\mathbf{P}(\mathbf{X}_{t-d}|\mathbf{e}_{1:t})$ for fixed d. That is, smoothing is done for the time slice d steps behind the current time t; as t increases, the smoothing has to keep up. Obviously, we can run the forward–backward algorithm over the d-step "window" as each new observation is added, but this seems inefficient. In Section 15.3, we will see that fixed-lag smoothing can, in some cases, be done in constant time per update, independently of the lag d.

FIXED-LAG SMOOTHING

Finding the most likely sequence

Suppose that $[true, true, false, true, true]$ is the umbrella sequence for the security guard's first five days on the job. What is the weather sequence most likely to explain this? Does the absence of the umbrella on day 3 mean that it wasn't raining, or did the director forget to bring it? If it didn't rain on day 3, perhaps (because weather tends to persist) it didn't rain on day 4 either, but the director brought the umbrella just in case. In all, there are 2^5 possible weather sequences we could pick. Is there a way to find the most likely one, short of enumerating all of them?

One approach we could try is the following linear-time procedure: use the smoothing algorithm to find the posterior distribution for the weather at each time step; then construct the sequence, using at each step the weather most likely according to the posterior. Such an approach should set off alarm bells in the reader's head, because the posteriors computed by smoothing are distributions over *single* time steps, whereas to find the most likely *sequence* we must consider *joint* probabilities over all the time steps. The results can in fact be quite different. (See Exercise 15.4.)

There *is* a linear-time algorithm for finding the most likely sequence, but it requires a little more thought. It relies on the same Markov property that yielded efficient algorithms for filtering and smoothing. The easiest way to think about the problem is to view each sequence as a *path* through a graph whose nodes are the possible *states* at each time step. Such a graph is shown for the umbrella world in Figure 15.5(a). Now consider the task of finding the most likely path through this graph, where the likelihood of any path is the product of the transition probabilities along the path and the probabilities of the given observations at each state. Let's focus in particular on paths that reach the state $Rain_5 = true$. Because of the Markov property, it follows that the most likely path to the state $Rain_5 = true$ consists of

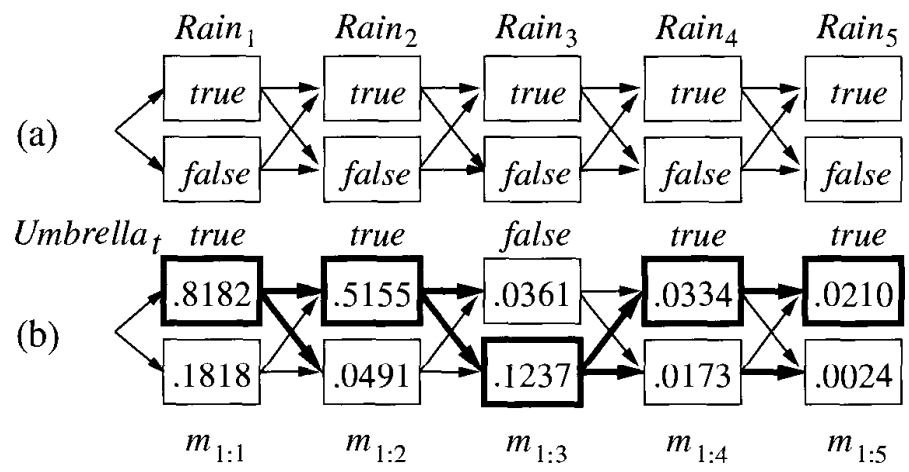


Figure 15.5 (a) Possible state sequences for $Rain_t$ can be viewed as paths through a graph of the possible states at each time step. (States are shown as rectangles to avoid confusion with nodes in a Bayes net.) (b) Operation of the Viterbi algorithm for the umbrella observation sequence $[true, true, false, true, true]$. For each t, we have shown the values of the message $\mathbf{m}_{1:t}$, which gives the probability of the best sequence reaching each state at time t. Also, for each state, the bold arrow leading into it indicates its best predecessor as measured by the product of the preceding sequence probability and the transition probability. Following the bold arrows back from the most likely state in $\mathbf{m}_{1:5}$ gives the most likely sequence.

the most likely path to *some* state at time 4 followed by a transition to $Rain_5 = true$; and the state at time 4 that will become part of the path to $Rain_5 = true$ is whichever maximizes the likelihood of that path. In other words, *there is a recursive relationship between most likely paths to each state* $\mathbf{x}_{t+1}$ *and most likely paths to each state* $\mathbf{x}_t$. We can write this relationship as an equation connecting the probabilities of the paths:

$$\max_{\mathbf{x}_1 \ldots \mathbf{x}_t} \mathbf{P}(\mathbf{x}_1, \ldots, \mathbf{x}_t, \mathbf{X}_{t+1} | \mathbf{e}_{1:t+1})$$
$$= \alpha \mathbf{P}(\mathbf{e}_{t+1} | \mathbf{X}_{t+1}) \max_{\mathbf{x}_t} \left(\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{x}_t) \max_{\mathbf{x}_1 \ldots \mathbf{x}_{t-1}} P(\mathbf{x}_1, \ldots, \mathbf{x}_{t-1}, \mathbf{x}_t | \mathbf{e}_{1:t}) \right) . \quad (15.9)$$

Equation (15.9) is *identical* to the filtering equation (15.3) except that

1. The forward message $\mathbf{f}_{1:t} = \mathbf{P}(\mathbf{X}_t | \mathbf{e}_{1:t})$ is replaced by the message
$$\mathbf{m}_{1:t} = \max_{\mathbf{x}_1 \ldots \mathbf{x}_{t-1}} \mathbf{P}(\mathbf{x}_1, \ldots, \mathbf{x}_{t-1}, \mathbf{X}_t | \mathbf{e}_{1:t}) ,$$
that is, the probabilities of the most likely path to each state $\mathbf{x}_t$; and
2. the summation over $\mathbf{x}_t$ in Equation (15.3) is replaced by the maximization over $\mathbf{x}_t$ in Equation (15.9).

Thus, the algorithm for computing the most likely sequence is similar to filtering: it runs forward along the sequence, computing the $\mathbf{m}$ message at each time step, using Equation (15.9). The progress of this computation is shown in Figure 15.5(b). At the end, it will have the probability for the most likely sequence reaching *each* of the final states. One can thus easily select the most likely sequence overall (the state outlined in bold). In order to identify the actual sequence, as opposed to just computing its probability, the algorithm will also need

to keep pointers from each state back to the best state that leads to it (shown in bold); the sequence is identified by following the pointers back from the best final state.

VITERBI ALGORITHM

The algorithm we have just described is called the **Viterbi algorithm**, after its inventor. Like the filtering algorithm, its complexity is linear in t, the length of the sequence. Unlike filtering, however, its space requirement is also linear in t. This is because the Viterbi algorithm needs to keep the pointers that identify the best sequence leading to each state.

15.3 HIDDEN MARKOV MODELS

The preceding section developed algorithms for temporal probabilistic reasoning using a general framework that was independent of the specific form of the transition and sensor models. In this and the next two sections, we discuss more concrete models and applications that illustrate the power of the basic algorithms and in some cases allow further improvements.

HIDDEN MARKOV MODEL

We begin with the **hidden Markov model**, or **HMM**. An HMM is a temporal probabilistic model in which the state of the process is described by a *single discrete* random variable. The possible values of the variable are the possible states of the world. The umbrella example described in the preceding section is therefore an HMM, since it has just one state variable: $Rain_t$. Additional state variables can be added to a temporal model while staying within the HMM framework, but only by combining all the state variables into a single "megavariable" whose values are all possible tuples of values of the individual state variables. We will see that the restricted structure of HMMs allows for a very simple and elegant matrix implementation of all the basic algorithms.[3] Section 15.6 shows how HMMs are used for speech recognition.

Simplified matrix algorithms

With a single, discrete state variable X_t, we can give concrete form to the representations of the transition model, the sensor model, and the forward and backward messages. Let the state variable X_t have values denoted by integers $1, \ldots, S$, where S is the number of possible states. The transition model $\mathbf{P}(X_t|X_{t-1})$ becomes an $S \times S$ matrix $\mathbf{T}$, where

$$\mathbf{T}_{ij} = P(X_t = j|X_{t-1} = i) .$$

That is, $\mathbf{T}_{ij}$ is the probability of a transition from state i to state j. For example, the transition matrix for the umbrella world is

$$\mathbf{T} = \mathbf{P}(X_t|X_{t-1}) = \begin{pmatrix} 0.7 & 0.3 \\ 0.3 & 0.7 \end{pmatrix} .$$

We also put the sensor model in matrix form. In this case, because the value of the evidence variable E_t is known to be, say, e_t, we need use only that part of the model specifying the probability that e_t appears. For each time step t, we construct a diagonal matrix $\mathbf{O}_t$ whose

[3] The reader unfamiliar with basic operations on vectors and matrices might wish to consult Appendix A before proceeding with this section.

diagonal entries are given by the values $P(e_t|X_t = i)$ and whose other entries are 0. For example, on day 1 in the umbrella world, $U_1 = true$, so, from Figure 15.2, we have

$$\mathbf{O}_1 = \begin{pmatrix} 0.9 & 0 \\ 0 & 0.2 \end{pmatrix} .$$

Now, if we use column vectors to represent the forward and backward messages, the computations become simple matrix–vector operations. The forward equation (15.3) becomes

$$\mathbf{f}_{1:t+1} = \alpha \mathbf{O}_{t+1}\mathbf{T}^\top \mathbf{f}_{1:t} \tag{15.10}$$

and the backward equation (15.7) becomes

$$\mathbf{b}_{k+1:t} = \mathbf{T}\mathbf{O}_{k+1}\mathbf{b}_{k+2:t} . \tag{15.11}$$

From these equations, we can see that the time complexity of the forward–backward algorithm (Figure 15.4) applied to a sequence of length t is $O(S^2 t)$, because each step requires multiplying an S-element vector by an $S \times S$ matrix. The space requirement is $O(St)$, because the forward pass stores t vectors of size S.

Besides providing an elegant description of the filtering and smoothing algorithms for HMMs, the matrix formulation reveals opportunities for improved algorithms. The first is a simple variation on the forward–backward algorithm that allows smoothing to be carried out in *constant* space, independently of the length of the sequence. The idea is that smoothing for any particular time slice k requires the simultaneous presence of both the forward and backward messages, $\mathbf{f}_{1:k}$ and $\mathbf{b}_{k+1:t}$, according to Equation (15.6). The forward–backward algorithm achieves this by storing the **f**s computed on the forward pass so that they are available during the backward pass. Another way to achieve this is with a single pass that propagates both **f** and **b** in the same direction. For example, the "forward" message **f** can be propagated backwards if we manipulate Equation (15.10) to work in the other direction:

$$\mathbf{f}_{1:t} = \alpha'(\mathbf{T}^\top)^{-1}\mathbf{O}_{t+1}^{-1}\mathbf{f}_{1:t+1} .$$

The modified smoothing algorithm works by first running the standard forward pass to compute $\mathbf{f}_{t:t}$ (forgetting all the intermediate results) and then running the backward pass for both **b** and **f** together, using them to compute the smoothed estimate at each step. Since only one copy of each message is needed, the storage requirements are constant (i.e. independent of t, the length of the sequence). There is one significant restriction on this algorithm: it requires that the transition matrix be invertible and that the sensor model have no zeroes—that is, every observation is possible in every state.

A second area in which the matrix formulation reveals an improvement is in *online* smoothing with a fixed lag. The fact that smoothing can be done in constant space suggests that there should exist an efficient recursive algorithm for online smoothing—that is, an algorithm whose time complexity is independent of the length of the lag. Let us suppose that the lag is d; that is, we are smoothing at time slice $t - d$, where the current time is t. By Equation (15.6), we need to compute

$$\alpha \mathbf{f}_{1:t-d}\mathbf{b}_{t-d+1:t}$$

for slice $t - d$. Then, when a new observation arrives, we need to compute

$$\alpha \mathbf{f}_{1:t-d+1}\mathbf{b}_{t-d+2:t+1}$$

for slice $t-d+1$. How can this be done incrementally? First, we can compute $\mathbf{f}_{1:t-d+1}$ from $\mathbf{f}_{1:t-d}$, using the standard filtering process, Equation (15.3).

Computing the backward message incrementally is more tricky, because there is no simple relationship between the old backward message $\mathbf{b}_{t-d+1:t}$ and the new backward message $\mathbf{b}_{t-d+2:t+1}$. Instead, we will examine the relationship between the old backward message $\mathbf{b}_{t-d+1:t}$ and the backward message at the front of the sequence, $\mathbf{b}_{t+1:t}$. To do this, we apply Equation (15.11) d times to get

$$\mathbf{b}_{t-d+1:t} = \left(\prod_{i=t-d+1}^{t} \mathbf{TO}_i\right) \mathbf{b}_{t+1:t} = \mathbf{B}_{t-d+1:t}\mathbf{1} \,, \tag{15.12}$$

where the matrix $\mathbf{B}_{t-d+1:t}$ is the product of the sequence of $\mathbf{T}$ and $\mathbf{O}$ matrices. $\mathbf{B}$ can be thought of as a "transformation operator" that transforms a later backward message into an earlier one. A similar equation holds for the new backward messages *after* the next observation arrives:

$$\mathbf{b}_{t-d+2:t+1} = \left(\prod_{i=t-d+2}^{t+1} \mathbf{TO}_i\right) \mathbf{b}_{t+2:t+1} = \mathbf{B}_{t-d+2:t+1}\mathbf{1} \,. \tag{15.13}$$

Examining the product expressions in Equations (15.12) and (15.13), we see that they have a simple relationship: to get the second product, "divide" the first product by the first element $\mathbf{TO}_{t-d+1}$, and multiply by the new last element $\mathbf{TO}_{t+1}$. In matrix language, then, there is a simple relationship between the old and new $\mathbf{B}$ matrices:

$$\mathbf{B}_{t-d+2:t+1} = \mathbf{O}_{t-d+1}^{-1}\mathbf{T}^{-1}\mathbf{B}_{t-d+1:t}\mathbf{T}\mathbf{O}_{t+1} \,. \tag{15.14}$$

This equation provides an incremental update for the $\mathbf{B}$ matrix, which in turn (through Equation (15.13)) allows us to compute the new backward message $\mathbf{b}_{t-d+2:t+1}$. The complete algorithm, which requires storing and updating $\mathbf{f}$ and $\mathbf{B}$, is shown in Figure 15.6.

15.4 Kalman Filters

Imagine watching a small bird flying through dense jungle foliage at dusk: you glimpse brief, intermittent flashes of motion; you try hard to guess where the bird is and where it will appear next so that you don't lose it. Or imagine that you are a World War II radar operator peering at a faint, wandering blip that appears once every 10 seconds on the screen. Or, going back further still, imagine you are Kepler trying to reconstruct the motions of the planets from a collection of highly inaccurate angular observations taken at irregular and imprecisely measured intervals. In all these cases, you are trying to estimate the state (position and velocity, for example) of a physical system from noisy observations over time. The problem can be formulated as inference in a temporal probability model, where the transition model describes the physics of motion and the sensor model describes the measurement process. This section examines the special representations and inference algorithms that have been developed to solve these sorts of problems; the method we will cover is called **Kalman filtering**, after its inventor, Rudolf E. Kalman.

KALMAN FILTERING

function FIXED-LAG-SMOOTHING(e_t, hmm, d) **returns** a distribution over $\mathbf{X}_{t-d}$
  inputs: e_t, the current evidence for time step t
    hmm, a hidden Markov model with $S \times S$ transition matrix $\mathbf{T}$
    d, the length of the lag for smoothing
  static: t, the current time, initially 1
    $\mathbf{f}$, a probability distribution, the forward message $\mathbf{P}(X_t|e_{1:t})$, initially PRIOR[hmm]
    $\mathbf{B}$, the d-step backward transformation matrix, initially the identity matrix
    $e_{t-d:t}$, double-ended list of evidence from $t-d$ to t, initially empty
  local variables: $\mathbf{O}_{t-d}, \mathbf{O}_t$, diagonal matrices containing the sensor model information

  add e_t to the end of $e_{t-d:t}$
  $\mathbf{O}_t \leftarrow$ diagonal matrix containing $\mathbf{P}(e_t|X_t)$
  if $t > d$ **then**
    $\mathbf{f} \leftarrow$ FORWARD($\mathbf{f}, e_t$)
    remove e_{t-d-1} from the beginning of $e_{t-d:t}$
    $\mathbf{O}_{t-d} \leftarrow$ diagonal matrix containing $\mathbf{P}(e_{t-d}|X_{t-d})$
    $\mathbf{B} \leftarrow \mathbf{O}_{t-d}^{-1}\mathbf{T}^{-1}\mathbf{B}\mathbf{T}\mathbf{O}_t$
  else $\mathbf{B} \leftarrow \mathbf{B}\mathbf{T}\mathbf{O}_t$
  $t \leftarrow t+1$
  if $t > d$ **then return** NORMALIZE($\mathbf{f} \times \mathbf{B1}$) **else return** null

Figure 15.6 An algorithm for smoothing with a fixed time lag of d steps, implemented as an online algorithm that outputs the new smoothed estimate given the observation for a new time step.

Clearly, we will need several *continuous* variables to specify the state of the system. For example, the bird's flight might be specified by position (X, Y, Z) and velocity $(\dot{X}, \dot{Y}, \dot{Z})$ at each point in time. We will also need suitable conditional densities to represent the transition and sensor models; as in Chapter 14, we will use **linear Gaussian** distributions. This means that the next state $\mathbf{X}_{t+1}$ must be a linear function of the current state $\mathbf{X}_t$, plus some Gaussian noise, a condition that turns out to be quite reasonable in practice. Consider, for example, the X-coordinate of the bird, ignoring the other coordinates for now. Let the interval between observations be Δ, and let us assume constant velocity; then the position update is given by

$$X_{t+\Delta} = X_t + \dot{X}\,\Delta\ .$$

If we add Gaussian noise then we have a linear Gaussian transition model:

$$P(X_{t+\Delta} = x_{t+\Delta}|X_t = x_t, \dot{X}_t = \dot{x}_t) = N(x_t + \dot{x}_t\,\Delta, \sigma)(x_{t+\Delta})\ .$$

The Bayesian network structure for a system with position $\mathbf{X}_t$ and velocity $\dot{\mathbf{X}}_t$ is shown in Figure 15.7. Note that this is a very specific form of linear Gaussian model; the general form will be described later in this section and covers a vast array of applications beyond the simple motion examples of the first paragraph. The reader might wish to consult Appendix A for some of the mathematical properties of Gaussian distributions; for our immediate purposes, the most important is that a **multivariate Gaussian** distribution for d variables is specified by a d-element mean $\boldsymbol{\mu}$ and a $d \times d$ covariance matrix $\boldsymbol{\Sigma}$.

MULTIVARIATE GAUSSIAN

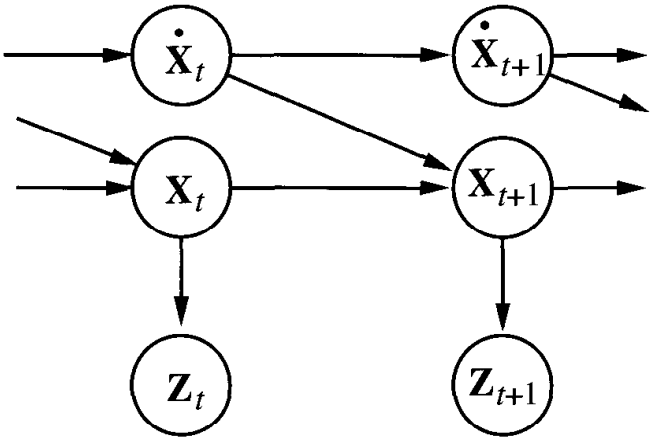


Figure 15.7 Bayesian network structure for a linear dynamical system with position $\mathbf{X}_t$, velocity $\dot{\mathbf{X}}_t$, and position measurement $\mathbf{Z}_t$.

Updating Gaussian distributions

In Chapter 14, we alluded to a key property of the linear Gaussian family of distributions: it remains closed under the standard Bayesian network operations. Here, we make this claim precise in the context of filtering in a temporal probability model. The required properties correspond to the two-step filtering calculation in Equation (15.3):

1. If the current distribution $\mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t})$ is Gaussian and the transition model $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t)$ is linear Gaussian, then the one-step predicted distribution given by

$$\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}) = \int_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t) P(\mathbf{x}_t|\mathbf{e}_{1:t})\, d\mathbf{x}_t \tag{15.15}$$

   is also a Gaussian distribution.

2. If the predicted distribution $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t})$ is Gaussian and sensor model $\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})$ is linear Gaussian, then, after conditioning on the new evidence, the updated distribution

$$\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) = \alpha\, \mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}) \tag{15.16}$$

   is also a Gaussian distribution.

Thus, the FORWARD operator for Kalman filtering takes a Gaussian forward message $\mathbf{f}_{1:t}$, specified by a mean $\boldsymbol{\mu}_t$ and covariance matrix $\boldsymbol{\Sigma}_t$, and produces a new multivariate Gaussian forward message $\mathbf{f}_{1:t+1}$, specified by a mean $\boldsymbol{\mu}_{t+1}$ and covariance matrix $\boldsymbol{\Sigma}_{t+1}$. So, if we start with a Gaussian prior $\mathbf{f}_{1:0} = \mathbf{P}(\mathbf{X}_0) = N(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)$, filtering with a linear Gaussian model produces a Gaussian state distribution for all time.

This seems to be a nice, elegant result, but why is it so important? The reason is that, except for a few special cases such as this, *filtering with continuous or hybrid (discrete and continuous) networks generates state distributions whose representation grows without bound over time.* This statement is not easy to prove in general, but Exercise 15.5 shows what happens for a simple example.

A simple one-dimensional example

We have said that the FORWARD operator for the Kalman filter maps a Gaussian into a new Gaussian. This translates into computing a new mean and covariance matrix from the previous mean and covariance matrix. Deriving the update rule in the general (multivariate) case requires rather a lot of linear algebra, so we will stick to a very simple univariate case for now; later we will give the results for the general case. Even for the univariate case, the calculations are somewhat tedious, but we feel that they are worth seeing because the usefulness of the Kalman filter is tied so intimately to the mathematical properties of Gaussian distributions.

The temporal model we will consider describes a **random walk** of a single continuous state variable X_t with a noisy observation Z_t. An example might be the "consumer confidence" index, which can be modeled as undergoing a random Gaussian-distributed change each month and is measured by a random consumer survey that also introduces Gaussian sampling noise. The prior distribution is assumed to be Gaussian with variance σ_0^2:

$$P(x_0) = \alpha\, e^{-\frac{1}{2}\left(\frac{(x_0-\mu_0)^2}{\sigma_0^2}\right)} .$$

(For simplicity, we will use the same symbol α for all normalizing constants in this section.) The transition model simply adds a Gaussian perturbation of constant variance σ_x^2 to the current state:

$$P(x_{t+1}|x_t) = \alpha\, e^{-\frac{1}{2}\left(\frac{(x_{t+1}-x_t)^2}{\sigma_x^2}\right)} .$$

The sensor model then assumes Gaussian noise with variance σ_z^2:

$$P(z_t|x_t) = \alpha\, e^{-\frac{1}{2}\left(\frac{(z_t-x_t)^2}{\sigma_z^2}\right)} .$$

Now, given the prior $P(X_0)$, we can compute the one-step predicted distribution using Equation (15.15):

$$\begin{aligned} P(x_1) &= \int_{-\infty}^{\infty} P(x_1|x_0)P(x_0)\,dx_0 = \alpha \int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(\frac{(x_1-x_0)^2}{\sigma_x^2}\right)} e^{-\frac{1}{2}\left(\frac{(x_0-\mu_0)^2}{\sigma_0^2}\right)} dx_0 \\ &= \alpha \int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(\frac{\sigma_0^2(x_1-x_0)^2+\sigma_x^2(x_0-\mu_0)^2}{\sigma_0^2\sigma_x^2}\right)} dx_0 . \end{aligned}$$

This integral looks rather complicated. The key to progress is to notice that the exponent is the sum of two expressions that are *quadratic* in x_0 and hence is itself a quadratic in x_0. A simple trick known as **completing the square** allows the rewriting of any quadratic $ax_0^2 + bx_0 + c$ as the sum of a squared term $a(x_0 - \frac{-b}{2a})^2$ and a residual term $c - \frac{b^2}{4a}$ that is independent of x_0. The residual term can be taken outside the integral, giving us

COMPLETING THE SQUARE

$$P(x_1) = \alpha\, e^{-\frac{1}{2}\left(c-\frac{b^2}{4a}\right)} \int_{-\infty}^{\infty} e^{-\frac{1}{2}(a(x_0-\frac{-b}{2a})^2)}\, dx_0 .$$

Now the integral is just the integral of a Gaussian over its full range, which is simply 1. Thus, we are left with only the residual term from the quadratic.